R453Plus1Toolbox
A package for importing and analyzing data from
Roche’s Genome Sequencer System

Hans-Ulrich Klein, Christoph Bartenhagen, Christian Ruckert

December 05, 2013

Contents

1 Introduction 3

2 Analysis of PCR amplicon projects 3
   2.1 Importing a Roche Amplicon Variant Analyzer project . . . . . . 3
      2.1.1 Import from AVA without AVA-CLI (version 2.5 and lower) 3
      2.1.2 Import from AVA with AVA-CLI . . . . . . . . . . . . . . 3
      2.1.3 Import from AVA with projects exported via AVA-CLI . . 4
      2.2 The AVASet class . . . . . . . . . . . . . . . . . . . . . . . . . 4
      2.3 Subsetting an AVASet . . . . . . . . . . . . . . . . . . . . . . . 9
      2.4 Setting filters on an AVASet . . . . . . . . . . . . . . . . . . . 10
      2.5 Variant coverage . . . . . . . . . . . . . . . . . . . . . . . . . 10
      2.6 Annotations and Variant Reports . . . . . . . . . . . . . . . . . 11
      2.7 Plotting . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
         2.7.1 Plot amplicon coverage . . . . . . . . . . . . . . . . . . 12
         2.7.2 Plot variation frequency . . . . . . . . . . . . . . . . . . 13
         2.7.3 Plot variant locations . . . . . . . . . . . . . . . . . . . 13
      2.8 VCF export . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3 Analysis of GS Mapper projects 14
   3.1 Importing a GS Reference Mapper project . . . . . . . . . . . . . 14
   3.2 The MapperSet class . . . . . . . . . . . . . . . . . . . . . . . . 14
   3.3 Setting filters and subsetting a MapperSet . . . . . . . . . . . . . 16
   3.4 Annotations and Variant Reports . . . . . . . . . . . . . . . . . . 16

4 Detection of structural variants 17
   4.1 Data preparation . . . . . . . . . . . . . . . . . . . . . . . . . . 17
   4.2 Computing and assessing putative structural variants . . . . . . . 19
   4.3 Visualization of breakpoints . . . . . . . . . . . . . . . . . . . . 21
## 5 Analysis and manipulation of SFF files

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Importing SFF files</td>
<td>22</td>
</tr>
<tr>
<td>5.2 The SFF container</td>
<td>22</td>
</tr>
<tr>
<td>5.3 Writing SFF files</td>
<td>23</td>
</tr>
<tr>
<td>5.4 Quality control of SFF files</td>
<td>23</td>
</tr>
</tbody>
</table>
1 Introduction

The R453Plus1 Toolbox comprises useful functions for the analysis of data generated by Roche’s 454 sequencing platform. It adds functions for quality assurance as well as for annotation and visualization of detected variants, complementing the software tools shipped by Roche with their product. Further, a pipeline for the detection of structural variants is provided.

> library(R453Plus1Toolbox)

2 Analysis of PCR amplicon projects

This section deals with the analysis of projects investigating massively parallel data generated from specifically designed PCR products.

2.1 Importing a Roche Amplicon Variant Analyzer project

The function `AVASet` imports data from Roche’s Amplicon Variant Analyzer (AVA). This can be done in three ways, depending on the version of your AVA software:

2.1.1 Import from AVA without AVA-CLI (version 2.5 and lower)

For projects created with the AVA software version \( \leq 2.5 \), `AVASet` expects only a `dirname` pointing to the project data, i.e. a directory that contains the following files and subdirectories:

- "Amplicons/ProjectDef/ampliconsProject.txt"
- "Amplicons/Results/Variants/currentVariantDefs.txt"
- "Amplicons/Results/Variants"
- "Amplicons/Results/Align"

There is an example project "AVASet" included in the `R453Plus1Toolbox` installation directory:

> projectDir = system.file("extdata", "AVASet", package = "R453Plus1Toolbox")
> avaSet = AVASet(dirname=projectDir)

2.1.2 Import from AVA with AVA-CLI

The function `AVASet` can directly access the AVA Command Line Interface (AVA-CLI) from within R. If the AVA software is installed on the same machine that runs R, the easiest way to import a project is to specify the project

> projectDir = system.file("extdata", "AVASet", package = "R453Plus1Toolbox")
> avaSet = AVASet(dirname=projectDir)
directory with `dirname` and the path to the binaries in the AVA software’s installation directory with `avaBin`. It is usually the directory "bin" containing the AVA-CLI command interpreter "doAmplicon". Let’s say the AVA software was installed to the directory "/home/User/AVA". Then, the function call looks like:

```r
> projectDir = "My/AVA/Project"
> avaSet = AVASet(dirname=projectDir, avaBin="/home/User/AVA/bin")
```

### 2.1.3 Import from AVA with projects exported via AVA-CLI

If the AVA software is not installed on the same machine that runs R, all data must be exported manually using AVA-CLI. It can be accessed via the command line interpreter "doAmplicon" from the AVA software’s installation directory. Within the AVA-CLI, load your project with the command "open". `AVASet` expects five files (variant information is optional):

<table>
<thead>
<tr>
<th>AVASet argument</th>
<th>AVA-CLI command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>file_sample</td>
<td>list sample -outputFile sample.csv</td>
<td>Table with sample names and annotations</td>
</tr>
<tr>
<td>file_amp</td>
<td>list amplicon -outputFile amp.csv</td>
<td>Table with primer sequences, positions and annotations</td>
</tr>
<tr>
<td>file_reference</td>
<td>list reference -outputFile reference.csv</td>
<td>Reference sequences</td>
</tr>
<tr>
<td>file_variant</td>
<td>list variant -outputFile variant.csv</td>
<td>Detected variants (if available)</td>
</tr>
<tr>
<td>file_variantHits</td>
<td>report variantHits -outputFile variantHits.csv</td>
<td>Variant hits for all samples (if available)</td>
</tr>
</tbody>
</table>

Table 1: `AVASet` function arguments for loading projects exported via AVA-CLI.

Note, that all exported tables are expected to be in csv-format.

There is an example project "AVASet_doAmplicon" included in the `R453Plus1Toolbox` installation directory:

```r
> projectDir = system.file("extdata", "AVASet_doAmplicon", package="R453Plus1Toolbox")
> avaSetExample = AVASet(dirname=projectDir, file_sample="sample.csv",
                        file_amp="amp.csv", file_reference="reference.csv", file_variant="variant.csv",
                        file_variantHits="variantHits.csv")
```

AVASet searches the specified `dirname` for the exported csv-files. `file_variant` and `file_variantHits` can be omitted if no variant information is available for the project.

### 2.2 The AVASet class

The `AVASet` class defines a container to store data imported from projects conducted with Roche’s AVA software. It extends the `Biobase eSet` to store all relevant information.
An object of class `AVASet` consists of three main components:

1. Variants:
   The variants part stores data about the found variants and is accessible by the functions `assayData`, `featureData` and `phenoData` known from Biobase eSet. The `assayData` slot contains four matrices with variants as rows and samples as columns:
      - `variantForwCount`: Matrix containing the number of reads with the respective variant in forward direction.
      - `variantRevCount`: Matrix containing the number of reads with the respective variant in reverse direction.
      - `totalForwCount`: Matrix containing the total coverage for every variant location in forward direction.
• totalRevCount: Matrix containing the total coverage for every variant location in reverse direction.

```r
> assayData(avaSet)$totalForwCount[1:3, ]
```

<table>
<thead>
<tr>
<th></th>
<th>Sample_1</th>
<th>Sample_2</th>
<th>Sample_3</th>
<th>Sample_4</th>
<th>Sample_5</th>
<th>Sample_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1438</td>
<td>119</td>
<td>1516</td>
<td>137</td>
<td>1729</td>
<td>1288</td>
<td>140</td>
</tr>
<tr>
<td>C369</td>
<td>267</td>
<td>1152</td>
<td>195</td>
<td>1518</td>
<td>1016</td>
<td>190</td>
</tr>
<tr>
<td>C595</td>
<td>258</td>
<td>1805</td>
<td>230</td>
<td>1885</td>
<td>1775</td>
<td>221</td>
</tr>
</tbody>
</table>

The `featureData` slot provides additional information on the variants. `fData` returns a data frame with variants as rows and the following columns:

- `name/canonicalPattern`: Short identifiers of a variant including the position and the bases changed.
- `referenceSeq`: Gives the identifier of the reference sequence (see below).
- `start/end`: The position of the variant relative to the reference sequence.
- `variantBase/referenceBases`: The bases changed in the variant.

```r
> fData(avaSet)[1:3, ]
```

<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>canonicalPattern</th>
<th>referenceSeqID</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1438</td>
<td>303:T/C</td>
<td>s(303,C)</td>
<td>I37</td>
<td>303</td>
<td>303</td>
</tr>
<tr>
<td>C369</td>
<td>309:T/C</td>
<td>s(309,C)</td>
<td>I36</td>
<td>309</td>
<td>309</td>
</tr>
<tr>
<td>C595</td>
<td>108:T/C</td>
<td>s(108,C)</td>
<td>I40</td>
<td>108</td>
<td>108</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>variantBase</th>
<th>referenceBases</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1438</td>
<td>C</td>
<td>T</td>
</tr>
<tr>
<td>C369</td>
<td>C</td>
<td>T</td>
</tr>
<tr>
<td>C595</td>
<td>C</td>
<td>T</td>
</tr>
</tbody>
</table>

The `phenoData` slot provides sample-IDs, multiplexer IDs (MID1, MID2), the pico titer plate (PTP) accession number, the lane, the read group and additional textual annotation for each sample. Most of these informations are imported directly from Roche’s software.

```r
> pData(avaSet)
```

<table>
<thead>
<tr>
<th></th>
<th>SampleID</th>
<th>MID1</th>
<th>MID2</th>
<th>PTP_AccNum</th>
<th>Lane</th>
<th>ReadGroup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample_1</td>
<td>I9646</td>
<td>Mid3</td>
<td>Mid3</td>
<td>GGSFDBH</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>Sample_2</td>
<td>I116</td>
<td>Mid1</td>
<td>Mid1</td>
<td>GA0582C</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>Sample_3</td>
<td>I9644</td>
<td>Mid1</td>
<td>Mid1</td>
<td>GGSFDBH</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>Sample_4</td>
<td>I118</td>
<td>Mid3</td>
<td>Mid3</td>
<td>GA0582C</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>Sample_5</td>
<td>I117</td>
<td>Mid2</td>
<td>Mid2</td>
<td>GA0582C</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>Sample_6</td>
<td>I9645</td>
<td>Mid2</td>
<td>Mid2</td>
<td>GGSFDBH</td>
<td>07</td>
<td></td>
</tr>
</tbody>
</table>

Annotation
2. Amplicons:
   This part stores information about the used amplicons and is accessible by the functions `assayDataAmp` and `fDataAmp`.
   The slot `assayDataAmp` contains two matrices with amplicons as rows and samples as columns:
   - `forwCount`: Matrix containing the number of reads for each amplicon and each sample in forward direction.
   - `revCount`: Matrix containing the number of reads for each amplicon and each sample in reverse direction.

   ```r
   > assayDataAmp(avaSet)$forwCount
   Sample_1  Sample_2  Sample_3  Sample_4  Sample_5
   TET2_E11.04  119   1516    137    1729    1288
   TET2_E06     248   400    224    478    339
   TET2_E11.03  267   1152   195    1518    1016
   TET2_E04     258   1805   230    1885    1775
   Sample_6
   TET2_E11.04  140
   TET2_E06     204
   TET2_E11.03  190
   TET2_E04     221
   ```

   The slot `featureDataAmp` contains an AnnotatedDataFrame with additional information on each amplicon:
   - `ampID`: The identifier of the current amplicon.
   - `primer1`, `primer2`: The primer sequences for each amplicon.
   - `referenceSeqID`: The identifier of the reference sequence (see below).
   - `targetStart/targetEnd`: The coordinates of the target region.

   ```r
   > fDataAmp(avaSet)
   ampID primer1
   TET2_E11.04 I90 CATTCACCTTCTACATAATCCA
   TET2_E06   I81 TGCAAGTGTGCCCTGTTTTG
   TET2_E11.03 I89 GCTCAGTCTACCACCCATCC
   TET2_E04   I79 GGGTTAAACCTTGTGGATG
   ```
primer2 referenceSeqID
targetStart targetEnd

<table>
<thead>
<tr>
<th>Amplicon</th>
<th>Primer</th>
<th>Reference Sequence</th>
<th>PhenoData</th>
</tr>
</thead>
<tbody>
<tr>
<td>TET2_E11.04</td>
<td>GAATTGACCCATGAGTTGGAG</td>
<td>I37</td>
<td></td>
</tr>
<tr>
<td>TET2_E06</td>
<td>AACCAAAGATGGGCTTTCC</td>
<td>I42</td>
<td></td>
</tr>
<tr>
<td>TET2_E11.03</td>
<td>AGATGCAGGGCATGAAGAGA</td>
<td>I36</td>
<td></td>
</tr>
<tr>
<td>TET2_E04</td>
<td>TTGTGACTCTCTGTTGAAATAGCA</td>
<td>I40</td>
<td></td>
</tr>
</tbody>
</table>

As both refer to the same samples, the variants phenoData slot is used for amplicons as well.

3. Reference sequences:
This part stores data about the reference sequences the amplicons were selected from. All information is stored into an object of class `AlignedRead`. The reads are accessible via `sread`. To retrieve additional information from Ensembl about the chromosome, the position and the strand of each reference sequence run function `alignShortReads` (see section 2.6 for details).

```r
> library(ShortRead)
> referenceSequences(avaSet)
```

class: AlignedRead
length: 4 reads; width: 339..346 cycles
chromosome: NA NA NA NA
position: 1 1 1 1
strand: NA NA NA NA
alignQuality: NumericQuality
alignData varLabels: name refSeqID gene

```r
> sread(referenceSequences(avaSet))
```

A DNAStringSet instance of length 4

<table>
<thead>
<tr>
<th>Read</th>
<th>Sequence</th>
<th>PhenoData</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>345 GGGTTAAGCTTT...CAGAGGTACCA</td>
<td>I40</td>
</tr>
<tr>
<td>[2]</td>
<td>346 CATTCACCTTCT...CATGGTCAATTC</td>
<td>I37</td>
</tr>
<tr>
<td>[3]</td>
<td>339 GTCAAGTCTACCA...ATGCCCTGCTCT</td>
<td>I36</td>
</tr>
<tr>
<td>[4]</td>
<td>341 TGCAAGTGGACCT...CCAATCTTTGGTT</td>
<td>I42</td>
</tr>
</tbody>
</table>

The following table sums up the available slots and accessor functions:
## Function/Slot | Description
---|---
**assayData** | Contains the number of reads and the total coverage for every variant and each sample in forward and reverse direction.

**fData/featureData** | Contains information about the type, position and reference of each variant.

**pData/phenoData** | Contains sample-IDs, multiplexer IDs (MID1, MID2), the pico titer plate (PTP) accession number, the lane, the read group and additional textual annotation for each sample.

**assayDataAmp** | Contains the number of reads for every amplicon and each sample in forward/reverse direction.

**fDataAmp/featureDataAmp** | Contains the primer sequences, reference sequence and the coordinates of the target region for each amplicon.

**referenceSequences** | Contains the reference sequences for the amplicons together with additional annotations.

### Table 2: AVASet contents and accessor functions.

### 2.3 Subsetting an AVASet

A subset of an AVASet object can be generated using the common "[]"-notation:

```r
> avaSubSet = avaSet[1:10, "Sample_1"]
```

The first dimension refers to the variants and the second dimension to the samples, so an AVASet with ten variants and one sample is returned. This is a short and to some extend equivalent version of the function `subset`, which expects a `subset` argument and the respective dimension (either "variants", "samples" or "amplicons"):

```r
> avaSubSet = subset(avaSet, subset=1:10, dimension="variants")
```

The following is equivalent to the "[]"-example above:

```r
> avaSubSet = subset(subset(avaSet, subset=1:10, dimension="variants"), subset="Sample_1", dimension="samples")
```

In contrast to the "[]"-Notation, the function `subset` allows further subsetting by amplicons:

```r
> avaSubSet = subset(avaSet, subset=c("TET2_E11.04", "TET2_E06"), dimension="amplicons")
```

When subsetting by amplicons all variants referring to amplicons that are not in the subset will be excluded.
2.4 Setting filters on an AVASet

Another way of generating a subset of an AVASet object is filtering only those variants, whose coverage (in percent) in forward and reverse direction respectively is higher than a given filter value in at least one sample. Here, the coverage is defined as the percentual amount of the reads with the given variant on the number of all reads covering the variant’s position. The function setVariantFilter returns an updated AVASet object that meets the given requirements:

> avaSetFiltered1 = setVariantFilter(avaSet, filter=0.05)

The above example returns an AVASet, which only contains variants whose coverage is greater than 5% in at least one sample. Passing a vector of two filter values applies filtering according to forward and reverse read direction separately:

> avaSetFiltered2 = setVariantFilter(avaSet, filter=c(0.1, 0.05))

In fact, when filtering an AVASet, the whole object is still available. The filter only affects the output given by accessor functions like fData, featureData and assayData.

The process can be reversed and the filter value(s) can be reset to zero by calling

> avaSet = setVariantFilter(avaSetFiltered1, filter=0)

or simply

> avaSet = setVariantFilter(avaSetFiltered2)

2.5 Variant coverage

The function getVariantPercentages displays the coverage of the variants for a given direction (either "forward", "reverse", or "both"):

> getVariantPercentages(avaSet, direction="both")[20:25, 1:4]

<table>
<thead>
<tr>
<th>Sample_1</th>
<th>Sample_2</th>
<th>Sample_3</th>
<th>Sample_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>C386</td>
<td>0.00000000</td>
<td>0.00000000</td>
<td>0.00000000</td>
</tr>
<tr>
<td>C1808</td>
<td>0.00000000</td>
<td>0.00000000</td>
<td>0.00000000</td>
</tr>
<tr>
<td>C1338</td>
<td>0.00000000</td>
<td>0.00000000</td>
<td>0.00000000</td>
</tr>
<tr>
<td>C1052</td>
<td>0.03202847</td>
<td>0.044400452</td>
<td>0.03076923</td>
</tr>
<tr>
<td>C818</td>
<td>0.00000000</td>
<td>0.003019628</td>
<td>0.00000000</td>
</tr>
<tr>
<td>C681</td>
<td>0.00000000</td>
<td>0.00000000</td>
<td>0.00000000</td>
</tr>
</tbody>
</table>

In the example above, getVariantPercentages is simply a short form of calculating

> (assayData(avaSet)[[1]] + assayData(avaSet)[[3]]) / (assayData(avaSet)[[2]] + assayData(avaSet)[[4]])
2.6 Annotations and Variant Reports

Before creating the variant and quality report, the reference sequences must be aligned against a reference genome and afterwards the variants have to be annotated.

The method `alignShortReads` aligns the reference sequences from an AVASet against a given reference genome. Only exact (no errors) and unique matches are returned. In the example below the hg19 assembly as provided by UCSC from package `BSgenome.Hsapiens.UCSC.hg19` is used as reference:

```r
> library(BSgenome.Hsapiens.UCSC.hg19)
> seqNames = names(Hsapiens)[1:24]
> avaSet = alignShortReads(avaSet, bsGenome=Hsapiens,
  seqNames=seqNames, ensemblNotation=TRUE)
```

The function `annotateVariants` annotates genomic variants (mutations) given in a data frame or more likely an AVASet. Annotation includes affected genes, exons and codons. Resulting amino acid changes are returned as well as dbSNP identifiers if the mutation is already known. All information is fetched from Ensembl via `biomaRt` and returned in an object of class `AnnotatedVariants`. It is advisable to filter the AVASet (see section 2.4) prior to that since the annotation process is very time consuming for a large number (>500) of variants.

```r
> avaSet = setVariantFilter(avaSet, filter=0.05)
> avaAnnot = annotateVariants(avaSet)
```

For an AVASet with corresponding annotated variants, the function `htmlReport` creates a html report containing variant and quality information. The report is structured into three pages:

1. Variant report by reference: This page sums up additional information for each variant including name, type, reference gene, position, changed nucleotides and affected samples. In addition, every variant is linked to a page with further details about the affected genes and transcripts (e.g. Ensembl gene-IDs, transcript-IDs, codon sequences, changes of amino acids (if coding)).

2. Variant report by sample: The upper fraction of this page presents an overview of all samples together with links to individual amplicon coverage plots for each sample. In the lower fraction the found variants are listed for each sample seperately in the same way as described in the variant report by reference above.

3. Quality report: The report shows the coverage of every amplicon in forward and/or reverse direction. Further plots display the coverage by MID and PTP (if this information is given in the pheno data of the object).

The following command creates a report containing only variants covered by at least 5% of the reads using the argument `minMut` (`minMut=3` is the default
value). The argument blocks can be used to structure the page by assigning each variant to a block. In this example the corresponding genes for each variant are used to create blocks, resulting in only one block in the example data set:

```r
> blocks = as.character(sapply(annotatedVariants(avaAnnot),
function(x) x$genes$external_gene_id))
> htmlReport(avaSet, annot=avaAnnot, blocks=blocks, dir="htmlReportExampleAVA",
title="htmlReport Example", minMut=3)
```

2.7 Plotting

2.7.1 Plot amplicon coverage

The function `plotAmpliconCoverage` creates a plot showing the coverage (number of reads) per amplicon, MID or PTP. This results in a barplot if the AVASet contains only one sample or in a boxplot for all other cases.

```r
> plotAmpliconCoverage(avaSet[, 2], type="amplicon")
```

![Figure 1: Barplot of the amplicon coverage for sample 2.](image)

```r
> plotAmpliconCoverage(avaSet, bothDirections=TRUE, type="amplicon")
```
2.7.2 Plot variation frequency

Given a Roche Amplicon Variant Analyzer Global Alignment export file, the function `plotVariationFrequency` creates a plot similar to the variation frequency plot in Roche’s GS Amplicon Variant Analyzer. The plot shows the reference sequence along the x-axis and indicates variants as bars at the appropriate positions. The height of the bars corresponds to the percentage of reads carrying the variant. A second y-axis indicates the absolute number of reads covering the variant. `plotRange` defines the start and end base of the reference sequence that should be plotted.

```r
> file = system.file("extdata", "AVAVarFreqExport", "AVAVarFreqExport.xls", package="R453Plus1Toolbox")
> plotVariationFrequency(file, plotRange=c(50, 150))
```

2.7.3 Plot variant locations

The function `plotVariants` illustrates the positions and types of mutations within a given gene and transcript (specified by an Ensembl gene/transcript id). The plot shows only coding regions (thus, units are amino acids / codons). The coding region is further divided into exons labeled with their rank in the transcript. An attribute `regions` allows to highlight special, predefined areas on the transcript like for example protein domains.

The function can be used in two ways:
It offers the most functionality when used as a "standalone" function by passing all mutations as a data frame. This mode allows an individual and detailed annotation of the mutations like labels, colors and user defined mutation types. It requires the columns "label", "pos" "mutation" and "color". It is recommended to add more detailed info for each mutation type by preparing a data frame for the parameter `mutationInfo` which requires the three columns "mutation", "legend" and "color".

The following example calls `plotVariants` for the gene TET2 having th Ensembl id "ENSG00000168769" and transcript "ENST00000513237" (see Figure 4 below):

```r
> data(plotVariantsExample)
> geneInfo = plotVariants(data=variants, gene="ENSG00000168769", transcript="ENST00000513237", regions=regions, mutationInfo=mutationInfo, horiz=TRUE, cex=0.8)
```

Especially for integration into the R453Plus1 Toolbox and for compatibility to older versions `plotVariants` also accepts annotated variants of class `annotatedVariants` (see section 2.6). The function then only distinguishes missense, nonsense and silent point mutations and deletions and does not include mutation labels.
2.8 VCF export

The variant call format (VCF) is a generic file format for storing DNA polymorphism data such as SNPs, insertions, deletions and structural variants, together with rich annotations ([Danecek et al., 2011](#)). The following command exports all variants stored in an AVASet object into a vcf file with the given name. Make sure to run `alignShortReads` first and optional add dbSNP identifiers with `annotateVariants` (see section 2.6 for details).

```r
> ava2vcf(avaSet, filename="variants.vcf", annot=avaAnnot)
```

3 Analysis of GS Mapper projects

Mapping projects allow the alignment of arbitrary reads from one or more sequencing runs to a given reference sequence.

3.1 Importing a GS Reference Mapper project

The function `MapperSet` imports data from Roche’s GS Reference Mapper. The GS Mapper software stores information for each sample in a separate directory, so `MapperSet` expects a character vector `dirs` containing the directories of all samples to read in, i.e. directories containing the files:

- "mapping/454HCDiffs.txt"
- "mapping/454NewblerMetrics.txt"

Furthermore the parameter `samplenames` allows the separate specification of sample names. If missing, the directory names are taken. The following example imports a project containing 3 samples (N01, N03, N04) with a total of 111 variants:

```r
> dir_sample01 = system.file("extdata", "MapperSet", "N01", package = "R453Plus1Toolbox")
> dir_sample03 = system.file("extdata", "MapperSet", "N03", package = "R453Plus1Toolbox")
> dir_sample04 = system.file("extdata", "MapperSet", "N04", package = "R453Plus1Toolbox")
> dirs = c(dir_sample01, dir_sample03, dir_sample04)
> mapperSet = MapperSet(dirs=dirs, samplenames=c("N01", "N03", "N04"))
```

3.2 The MapperSet class

An object of class `MapperSet` is a container to store data imported from a project of Roche’s GS Reference Mapper Software. It directly extends the `Biobase eSet` class and as such provides the following slots:
1. The **assayData** slot contains four matrices with variants as rows and samples as columns:
   - `variantForwCount/variantRevCount`: Matrices containing the number of reads with the respective variant in forward/reverse direction.
   - `totalForwCount/totalRevCount`: Matrices containing the total read coverage for every variant location in forward/reverse direction.

2. The **featureData** slot holds the variants as rows together with additional information on each variant within the following columns:
   - `chromosome/start/end/strand`: Give the location of each variant.
   - `referenceBases/variantBase`: Show the base(s) changed in each variant.
   - `regName`: The name of the region (gene) where the variant is located.
   - `knownSNP`: Contains dbSNP reference cluster ids for known SNPs as given by the GS Mapper software (if any).

3. The **phenoData** slot contains additional information about the samples represented as rows:
   - By default, the phenoData slot only contains an accession number indicating the PTP of every sample.

```r
> mapperSet
MapperSet (storageMode: list)
  assayData: 111 features, 3 samples
    element names: variantForwCount, totalForwCount, variantRevCount, totalRevCount
  protocolData: none
  phenoData
    sampleNames: N01 N03 N04
    varLabels: accessionNumber
    varMetadata: labelDescription
  featureData
    featureNames: 1 2 ... 111 (111 total)
    fvarLabels: chr strand ... knownSNP (8 total)
    fvarMetadata: labelDescription
  experimentData: use 'experimentData(object)'
Annotation:

As the **MapperSet** is derived from the **Biobase eSet** the methods used to access or to manipulate the elements of a **MapperSet** object remain the same:

```r
> assayData(mapperSet)$variantForwCount[1:4, ]
```
3.3 Setting filters and subsetting a MapperSet

The MapperSet uses the same methods for filtering and subsetting as the AVASet (see section 2.3 and 2.4 for details).

3.4 Annotations and Variant Reports

Before creating the variant and quality report, the variants have to be annotated using function annotateVariants. Annotation includes affected genes, exons and codons. Resulting amino acid changes are returned as well as dbSNP identifiers, if the mutation is already known. All information is fetched from Ensembl via biomaRt and returned in an object of class AnnotatedVariants. It is advisable to filter the MapperSet (see section 3.3) since the annotation process is very time consuming for a large number (>500) of variants.
mapperAnnot = annotateVariants(mapperSet)

For a MapperSet with corresponding annotated variants, the function htmlReport creates a html report containing detailed variant information. The report is structured into two pages:

1. Variant report by reference: This page sums up additional information for each variant including name, type, reference gene, position, changed nucleotides and affected samples. Furthermore every variant is linked to a page with further details about the affected genes and transcripts (e.g. Ensembl gene-IDs, transcript-IDs, codon sequences, changes of amino acids (if coding)).

2. Variant report by sample: The upper fraction of this page presents an overview of all samples. In the lower fraction the found variants are listed for each sample seperately in the same way as described in the variant report by reference above.

The following command creates a report containing only variants covered by at least 3% of the reads using the argument minMut (minMut=3 is also the default value):


4 Detection of structural variants

Structural variants like translocations or inversions can be detected using non-paired reads if at least one read spans the breakpoint of the variant. These reads originate from two different locations on the reference genome and are called 'chimeric reads'.

4.1 Data preparation

Before breakpoints can be detected, the generated raw sequences must be preprocessed and aligned. Of course, data preprocessing depends on the applied laboratory protocols. The exemplary data set used in this vignette is a subset of the data set presented by Kohlmann et al. ([Kohlmann et al., 2009]) and is described in detail therein.

In our example data set, each region of the pico titer plate contains reads from three different samples which were loaded into that region. To reallocate reads to samples, each sample has a unique multiplex sequence prefixing all reads from that sample. This allocation process is called demultiplexing. In the code section below, the multiplexed sequences are read in and demultiplexed according to the given multiplex sequences (MIDs) using the demultiplexReads method. The standard multiplex sequences used by the Genome Sequence MID
library kits can be retrieved by calling `genomeSequencerMIDs`. The last two commands show that all reads could be successfully demultiplexed.

```r
> fnaFile = system.file("extdata", "SVDetection", "R_2009_07_30", "D_2009_07_31", "1.TCA.454Reads.fna", package="R453Plus1Toolbox")
> seqs = readDNAStringSet(fnaFile, format="fasta")
> MIDSeqs = genomeSequencerMIDs(c("MID1", "MID2", "MID3"))
> dmReads = demultiplexReads(seqs, MIDSeqs, numMismatches=2, trim=TRUE)
> length(seqs)
[1] 523
> sum(sapply(dmReads, length))
[1] 523
```

A sequence capture array was used to ensure that the example data set predominantly contains reads from certain genomic regions of interest. The applied NimbleGen array captured short segments corresponding to all exon regions of 92 distinct target genes. In addition, contiguous genomic regions for three additional genes, i.e. `CBFB`, `MLL`, and `RUNX1`, were present on the array. During sample preparation, linkers were ligated to the polished fragments in the library to provide a priming site as recommended by the NimbleGen protocol. These linker sequences were sequenced and are located at the 5 prime end of the reads. In case of long reads, the reverse complement of the linker may be located at the 3 prime end. The function `removeLinker` can be used to remove these linkers. Additionally, very short reads are discarded in the following code snippet.

```r
> minReadLength = 15
> gSel3 = sequenceCaptureLinkers("gSel3")[[1]]
> trimReads = lapply(dmReads, function (reads) {
  reads = reads[width(reads) >= minReadLength]
  reads = removeLinker(reads, gSel3)
  reads = reads[width(reads) >= minReadLength]
  readsRev = reverseComplement(reads)
  readsRev = removeLinker(readsRev, gSel3)
  reads = reverseComplement(readsRev)
  reads = reads[width(reads) >= minReadLength]
  return(reads)
})
```

Finally, the preprocessed reads must be aligned against a reference genome. For this purpose, we write the reads to a .fasta file and use the BWA-SW
algorithm for generating local alignments. The BWA-SW algorithm can be substituted by other local alignment algorithms. However, BWA-SW has the useful feature to only report the best local alignments. Hence, two local alignments do not overlap on the query sequence (they may overlap on the reference). This is an assumption made by the pipeline implemented in this package.

```r
> write.XStringSet(trimReads[['MID1']], file="/tmp/N01.fasta", format="fasta")
```

### 4.2 Computing and assessing putative structural variants

As chimeric reads may also be caused by technical issues during sample preparation, the function `filterChimericReads` implements several filter steps to remove artificial chimeric reads.

The remaining reads are passed to the `detectBreakpoints` method to create clusters representing putative breakpoints. Each cluster contains all chimeric reads that span this breakpoint. Promising candidates are clusters with more than one read and ideally with reads from different strands. Some structural variations like translocations or inversion lead to two related breakpoints. In the context of fusion genes, these breakpoints are referred to as *pathogenic* and *reciprocal* breakpoint. By the use of read orientation and strand information during clustering, it is ensured that reads from the pathogenic breakpoint will not cluster together with reads from the reciprocal breakpoint, although their genomic coordinates may be close to each other or even equal. After clustering, consensus breakpoint coordinates are computed for each cluster.

In the last step, the function `mergeBreakpoints` searches breakpoints that originate from the same structural variation (i.e. the pathogenic and the related reciprocal breakpoint) and merges them. We observed, that the distance between two related breakpoints may be up to a few hundred basepairs, whereas the breakpoint coordinates of single reads spanning the same breakpoint vary only by a very few bases due to sequencing errors or ambiguities during alignment.

In the following example, we use the reads from sample N01 presented in the previous section. The reads have been aligned using BWA-SW:

```r
> library("Rsamtools")
> bamFile = system.file("extdata", "SVDetection", "bam", "N01.bam",
package="R453Plus1Toolbox")
> parameters = ScanBamParam(what=scanBamWhat())
> bam = scanBam(bamFile, param=parameters)
```

For the filtering step, we specify a target region, i.e. the used capture array in form of a `RangesList`. All chimeric reads not overlapping this region with at least one local alignment are discarded. The following example creates a target
region out of a given .bed file containing region information using functions from package rtracklayer.

```r
> library("rtracklayer")
> bedFile = system.file("extdata", "SVDetection", "chip", 
  "CaptureArray_hg19.bed", package="R453Plus1Toolbox")
> chip = import.ucsc(bedFile, subformat="bed")
> chip = split(ranges(chip[[1]]), seqnames(chip[[1]]))
> names(chip) = gsub("chr", ",", names(chip))
> linker = sequenceCaptureLinkers("gSel3")[[1]]
> filterReads = filterChimericReads(bam, targetRegion=chip, linkerSeq=linker)
> filterReads$log

<table>
<thead>
<tr>
<th>AlignedReads</th>
<th>ChimericReads</th>
<th>TwoLocalAlignments</th>
<th>TargetRegion</th>
<th>NoLinker</th>
<th>MinimumDistance</th>
<th>Unique5PrimeStart</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>213</td>
<td>24</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The linkerSeq argument allows to specify the linker sequence used during sample preparation. All chimeric reads that have this linker sequence between their local alignments are removed.

Finally, we call the detectBreakpoints and mergeBreakpoints functions:

```r
> bp = detectBreakpoints(filterReads, minClusterSize=1)
> bp

Size ChrA ChrB
BP1 8 16 16
BP2 4 16 16
BP3 1 21 1
BP4 1 1 7
BP5 1 1 1
BP6 1 1 16

> table(bp)
size
1 4 8
1 1 1

> mbp = mergeBreakpoints(bp)
> summary(mbp)

<table>
<thead>
<tr>
<th>ChrA ChrB</th>
<th>BpACase1</th>
<th>BpBCase1</th>
<th>BpACase2</th>
<th>BpBCase2</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP1_BP2</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>15815191</td>
</tr>
<tr>
<td>BP3</td>
<td>21</td>
<td>1</td>
<td>36496155</td>
<td>177984464</td>
</tr>
<tr>
<td>BP4</td>
<td>2</td>
<td>1</td>
<td>16382474</td>
<td>186276897</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>BP</th>
<th>1</th>
<th>7</th>
<th>186275614</th>
<th>102017970</th>
<th>NA</th>
<th>NA</th>
</tr>
</thead>
<tbody>
<tr>
<td>BP6</td>
<td>1</td>
<td>16</td>
<td>186271118</td>
<td>67130495</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BP7</td>
<td>1</td>
<td>11</td>
<td>174926056</td>
<td>118389220</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BP8</td>
<td>1</td>
<td>16</td>
<td>120222145</td>
<td>15853548</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BP9</td>
<td>6</td>
<td>1</td>
<td>168290170</td>
<td>178598476</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BP10</td>
<td>21</td>
<td>1</td>
<td>37093848</td>
<td>150600467</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BP11</td>
<td>15</td>
<td>1</td>
<td>63213753</td>
<td>164769097</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BP12</td>
<td>21</td>
<td>1</td>
<td>36450706</td>
<td>18673804</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>BP13</td>
<td>1</td>
<td>21</td>
<td>192053733</td>
<td>37167301</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

NoReadsCase1 NoReadsCase2 NoReadsTotal
BP1_BP2 4/4 3/1 12
BP3 1/0 0/0 1
BP4 1/0 0/0 1
BP5 1/0 0/0 1
BP6 1/0 0/0 1
BP7 1/0 0/0 1
BP8 1/0 0/0 1
BP9 1/0 0/0 1
BP10 1/0 0/0 1
BP11 1/0 0/0 1
BP12 1/0 0/0 1
BP13 1/0 0/0 1

One cluster of size 8 and another cluster of size 4 were detected. Both putative breakpoints span two regions on chromosome 16. Further, 11 clusters of size one were found. The `mergeBreakpoints` function merges the first two clusters. The summary reveals that the coordinates of the breakpoints only differ by two bases at each region on chromosome 16. Moreover, both strands from both breakpoints were sequenced. Obviously, we detected two related breakpoints caused by an inversion on chromosome 16.

4.3 Visualization of breakpoints

The function `plotChimericReads` takes the output of the function `mergeBreakpoints` and produces a plot of the breakpoint regions together with the aligned reads and marks deletions, insertions and mismatches. If a pathogenic and a reciprocal breakpoint exist, `plotChimericReads` creates two plots as shown in the example below.

The following example shows the breakpoints (pathogenic and reciprocal) of an inversion on chromosome 16 where 12 reads aligned:

```r
> plotChimericReads(mbp[1], legend=TRUE)
```

Optionally (if the argument `plotBasePairs` is `TRUE`), `plotChimericReads` displays all base pairs within a given region of size `maxBasePairs` around the breakpoint:

```r
```
5 Analysis and manipulation of SFF files

The Standard Flowgram Format (SFF) is a binary file format designed by Roche to store the homopolymer stretches typical for 454 sequencing. It consists of a common header section, containing general information (e.g. number of reads, nucleotides used for each flow) and for each read a read header (e.g. read length, read name) and read data section (e.g. called bases, flow values, quality scores).

5.1 Importing SFF files

SFF files be imported using the `readSFF` function.

```r
> file <- system.file("extdata", "SFF", "example.sff", package="R453Plus1Toolbox")
> sffContainer <- readSFF(file)
Reading file example.sff ... done!
```

5.2 The SFF container

The contents of the SFF file are stored in an object of class `SFFContainer` with different slots:

```r
> showClass("SFFContainer")
Class "SFFContainer" [package "R453Plus1Toolbox"]

Slots:

Name: name        flowgramFormat
Class: character   numeric

Name: flowChars    keySequence
Class: character   character

Name: clipQualityLeft    clipQualityRight
Class: numeric           numeric

Name: clipAdapterLeft    clipAdapterRight
Class: numeric           numeric

Name: flowgrams      flowIndexes
Class: list           list

Name: reads
Class: QualityScaledDNAStringSet
```
The most important slot is the reads slot containing the called bases and the corresponding quality measures:

```r
> reads(sffContainer)
```

A QualityScaledDNAStringSet instance containing:

A DNAStringSet instance of length 10

<table>
<thead>
<tr>
<th>width</th>
<th>seq names</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>TCAGACTACTATG...AGGCGATACGNN GWDFKFT02CLU66</td>
</tr>
<tr>
<td>[2]</td>
<td>TCAGTCTAGTGCAC...ACGNNNNNNNNN GWDFKFT02BRR5H</td>
</tr>
<tr>
<td>[3]</td>
<td>TCAGCGACGTGAC...AGGACCGATACG GWDFKFT02BRRRE3</td>
</tr>
<tr>
<td>[4]</td>
<td>TCAGACTACTATG...CAAGCGCATAG GWDFKFT02BRR6G</td>
</tr>
<tr>
<td>[5]</td>
<td>TCAGAGACCAGTCG...CAAGCGCATAG GWDFKFT02BRRRE3</td>
</tr>
<tr>
<td>[6]</td>
<td>TCAGTCTAGGCGAC...GCAAGCGCATAG GWDFKFT02CLVYX</td>
</tr>
<tr>
<td>[7]</td>
<td>TCAGGCAGGTGAC...AGGCGCATAGNN GWDFKFT02BRR9U</td>
</tr>
<tr>
<td>[8]</td>
<td>TCAGACTACTATG...AAGCGCATAGNN GWDFKFT02BR4IB</td>
</tr>
<tr>
<td>[9]</td>
<td>TCAGCGACGTGAC...GAGCGCATAGNN GWDFKFT02BSHNB</td>
</tr>
<tr>
<td>[10]</td>
<td>TCAGACTACTATG...AAGCGCATAGNN GWDFKFT02BSAFV</td>
</tr>
</tbody>
</table>

A PhredQuality instance of length 10

<table>
<thead>
<tr>
<th>width</th>
<th>seq names</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>IIIIIIIIIIIIF&gt;...2119:EEA?!! GWDFKFT02CLU66</td>
</tr>
<tr>
<td>[2]</td>
<td>IIIIIIIIIIIII...EEE!!!!!!! GWDFKFT02BRR5H</td>
</tr>
<tr>
<td>[3]</td>
<td>IIIIIIIIIIIII...9;;?==EEEII GWDFKFT02BRRRE3</td>
</tr>
<tr>
<td>[4]</td>
<td>IIIIIIIIIIIII...000566&lt;&lt;&lt;&lt; GWDFKFT02BRR6G</td>
</tr>
<tr>
<td>[5]</td>
<td>IIIIIIIIIIIII...E???&gt;CIIIII GWDFKFT02BRRRE3</td>
</tr>
<tr>
<td>[6]</td>
<td>IIIIIIIIIIIII...&gt;&gt;77998@AA@E GWDFKFT02CLVYX</td>
</tr>
<tr>
<td>[7]</td>
<td>IIIIIIIIIIIII...22/&lt;&lt;CEECE!! GWDFKFT02BRR9U</td>
</tr>
<tr>
<td>[8]</td>
<td>IIIIIIIIIIIII...1127&lt;EHHEG!! GWDFKFT02BR4IB</td>
</tr>
<tr>
<td>[9]</td>
<td>HIHHHHHHHHHHH...-.5:&lt;57!! GWDFKFT02BSHNB</td>
</tr>
<tr>
<td>[10]</td>
<td>IIIIIIIIIIIII...111179CHHCE! GWDFKFT02BSAFV</td>
</tr>
</tbody>
</table>

An **SFFContainer** object can be subsetted using the [ operator and some read names or numbers:

```r
> subSffContainer <- sffContainer[1:5]
```

### 5.3 Writing SFF files

An **SFFContainer** can be written back into a file using the `writeSFF` method:

```r
> writeSFF(subSffContainer, subSffFile.sff)
```

### 5.4 Quality control of SFF files

The `qualityReportSFF` function creates a PDF document from one or more SFF files containing information relevant for quality control (e.g. read length
distributions, quality histograms, GC content). Two example plots are shown below:

> positionQualityBoxplot(sffContainer)

> dinucleotideOddsRatio(sffContainer)

References


Figure 2: Boxplot of the coverage for four amplicons separated by read direction.
Figure 3: Plot of the variation frequency for a given reference sequence.
Figure 4: Plot of the variants for gene TET2 by passing mutations as a data frame. This version includes mutation labels and allows user defined mutation types.
Figure 5: Plot of the breakpoint region.

Figure 6: Plot of the breakpoint region including base pairs.
Boxplot of quality per position
example.sff
Read position in bp (Bin size: 10bp)
Quality score
20 40 60 80 100
0 10 20 30 40

Figure 7: Position quality boxplot - One of the plots contained in the PDF quality report.
Figure 8: Dinucleotide odds ratio showing the over-/under-representation of dinucleotides - One of the plots contained in the PDF quality report.