Package ‘affxparser’

May 3, 2024

Version 1.76.0

Depends R (>= 2.14.0)

Suggests R.oo (>= 1.22.0), R.utils (>= 2.7.0), AffymetrixDataTestFiles

Title Affymetrix File Parsing SDK

Author Henrik Bengtsson [aut], James Bullard [aut], Robert Gentleman [ctb], Kasper Daniel Hansen [aut, cre], Jim Hester [ctb], Martin Morgan [ctb]

Maintainer Kasper Daniel Hansen <kasperdanielhansen@gmail.com>

Description Package for parsing Affymetrix files (CDF, CEL, CHP, BPMAP, BAR). It provides methods for fast and memory efficient parsing of Affymetrix files using the Affymetrix' Fusion SDK. Both ASCII- and binary-based files are supported. Currently, there are methods for reading chip definition file (CDF) and a cell intensity file (CEL). These files can be read either in full or in part. For example, probe signals from a few probesets can be extracted very quickly from a set of CEL files into a convenient list structure.

Note Fusion SDK v1.1.2

License LGPL (>= 2)

LazyLoad yes

URL https://github.com/HenrikBengtsson/affxparser

BugReports https://github.com/HenrikBengtsson/affxparser/issues

biocViews Infrastructure, DataImport, Microarray, ProprietaryPlatforms, OneChannel

git_url https://git.bioconductor.org/packages/affxparser

git_branch RELEASE_3_19

git_last_commit 4f86ff8

git_last_commit_date 2024-04-30

Repository Bioconductor 3.19

Date/Publication 2024-05-03
## Contents

affxparser-package

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

1. Dictionary

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>

2. Cell coordinates and cell indices

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

9. Advanced - Cell-index maps for reading and writing

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

applyCdfGroupFields

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>

applyCdfGroups

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
</tr>
</tbody>
</table>

arrangeCelFilesByChipType

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
</tbody>
</table>

cdfAddBaseMmCounts

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
</tbody>
</table>

cdfAddPlaqTypes

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
</tbody>
</table>

cdfAddProbeOffsets

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
</tr>
</tbody>
</table>

cdfGetFields

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
</tr>
</tbody>
</table>

cdfGetGroups

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
</tr>
</tbody>
</table>

cdfGtypeCelToPQ

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
</tr>
</tbody>
</table>

cdfHeaderToCelHeader

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

cdfMergeAlleles

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
</tr>
</tbody>
</table>

cdfMergeStrands

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
</tr>
</tbody>
</table>

cdfMergeToQuartets

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
</tr>
</tbody>
</table>

cdfOrderBy

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
</tr>
</tbody>
</table>

cdfOrderColumnsBy

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
</tr>
</tbody>
</table>

cdfSetDimension

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

cmpareCdfs

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
</tr>
</tbody>
</table>

cmpareCels

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
</tr>
</tbody>
</table>

cmpvertCdf

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
</tr>
</tbody>
</table>

cmpvertCel

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
</tr>
</tbody>
</table>

copyCel

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
</tbody>
</table>

createCel

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
</tr>
</tbody>
</table>

cmpfindCdf

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
</tr>
</tbody>
</table>

cmpfindFiles

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
</tr>
</tbody>
</table>

invertMap

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>36</td>
</tr>
</tbody>
</table>

isCelFile

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>37</td>
</tr>
</tbody>
</table>

parseDatHeaderString

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>38</td>
</tr>
</tbody>
</table>

readBpmap

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>39</td>
</tr>
</tbody>
</table>

readCcg

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
</tr>
</tbody>
</table>

readCcgHeader

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
</tr>
</tbody>
</table>

readCdf

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>43</td>
</tr>
</tbody>
</table>

readCdfCellIndices

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
</tr>
</tbody>
</table>

readCdfDataFrame

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
</tr>
</tbody>
</table>

readCdfGroupName

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>47</td>
</tr>
</tbody>
</table>

readCdfHeader

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
</tr>
</tbody>
</table>

readCdfIsPm

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
</tr>
</tbody>
</table>

readCdfNbrOfCellsPerUnitGroup

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
</tr>
</tbody>
</table>

readCdfQc

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
</tr>
</tbody>
</table>

readCdfUnitNames

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
</tr>
</tbody>
</table>

readCdfUnits

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
</tr>
</tbody>
</table>

readCdfUnitsWriteMap

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>56</td>
</tr>
</tbody>
</table>

readCel
affxparser-package

readCelHeader .................................. 62
readCelIntensities ................................ 63
readCelRectangle .................................. 65
readCelUnits ...................................... 66
readChp ............................................ 68
readClf ............................................. 69
readClfEnv ......................................... 70
readClfHeader ...................................... 71
readPgf ............................................. 71
readPgfEnv ......................................... 72
readPgfHeader ...................................... 74
updateCel .......................................... 75
updateCelUnits ..................................... 78
writeCdf ............................................ 80
writeCdfHeader ..................................... 81
writeCdfQcUnits ................................... 82
writeCdfUnits ....................................... 82
writeCelHeader ..................................... 83
writeTpmap ......................................... 84

Index 86

affxparser-package Package affxparser

Description

The affxparser package provides methods for fast and memory efficient parsing of Affymetrix files [1] using the Affymetrix' Fusion SDK [2,3]. Both traditional ASCII- and binary (XDA)-based files are supported, as well as Affymetrix future binary format "Calvin". The efficiency of the parsing is dependent on whether a specific file is binary or ASCII.

Currently, there are methods for reading chip definition file (CDF) and a cell intensity file (CEL). These files can be read either in full or in part. For example, probe signals from a few probesets can be extracted very quickly from a set of CEL files into a convenient list structure.

To get started

To get started, see:

1. readCelUnits() - reads one or several Affymetrix CEL file probeset by probeset.
2. readCel() - reads an Affymetrix CEL file. by probe.
3. readCdf() - reads an Affymetrix CDF file. by probe.
4. readCdfUnits() - reads an Affymetrix CDF file unit by unit.
5. readCdfCellIndices() - Like readCdfUnits(), but returns cell indices only, which is often enough to read CEL files unit by unit.
6. applyCdfGroups() - Re-arranges a CDF structure.
7. findCdf() - Locates an Affymetrix CDF file by chip type. This page also describes how to setup default search path for CDF files.
Setting up the CDF search path

Some of the functions in this package search for CDF files automatically by scanning certain directories. To add directories to the default search path, see instructions in `findCdf()`.

Future Work

Other Affymetrix files can be parsed using the Fusion SDK. Given sufficient interest we will implement this, e.g. DAT files (image files).

Running examples

In order to run the examples, data files must exist in the current directory. Otherwise, the example scripts will do nothing. Most of the examples require a CDF file or a CEL file, or both. Make sure the CDF file is of the same chip type as the CEL file.

Affymetrix provides data sets of different types at http://www.affymetrix.com/support/datasets. affx that can be used. There are both small and very large data sets available.

Technical details

This package implements an interface to the Fusion SDK from Affymetrix.com. This SDK (software development kit) is an open source library used for parsing the various file formats used by the Affymetrix platform.

The intention is to provide interfaces to most if not all file formats which may be parsed using Fusion.

The SDK supports parsing of all the different versions of a specific file format. This means that ASCII, binary as well as the new binary format (codename Calvin) used by Affymetrix is supported through a single API. We also expect any future changes to the file formats to be reflected in the SDK, and subsequently in this package.

However, as the current Fusion SDK does not support compressed files, neither does affxparser. This is in contrast to some of the existing code in affy and relatives (see below for links).

In general we aim to provide functions returning all information in the respective files. Currently it seems that future Affymetrix chip designs may consist of so many features that returning all information will lead to an unnecessary overhead in the case a user only wants access to a subset.

We have tried to make this possible.

For older file, certain entries in the files have been removed from newer specifications, and the SDK does not provide utilities for reading these entries. This includes for instance the FEAT column of CDF files.

Currently the package as well as the Fusion SDK is in beta stage. Bugs may be related to either codebase. We are very interested in users being unable to compile/parse files using this library - this includes users with custom chip designs.

In addition, since we aim to return all information stored in the file (and accessible using the Fusion SDK) we would like reports from users being unable to do that.

The efficiency of the underlying code may vary with the version of the file being parsed. For example, we currently report the number of outliers present in a CEL file when reading the header of the file using `readCelHeader`. In order to obtain this information from text based CEL files
1. Dictionary

(version 2), the entire file needs to be read into memory. With version 3 of the file format, this information is stored in the header.

With the introduction of the Fusion SDK (and the next version of their file formats) Affymetrix has made it possible to use multibyte character sets. This implies that character information may be inaccessible if the compiler used to compile the C++ code does not support multibyte character sets (specifically we require that the R installation has defined the macro SUPPORT_MCBS in the Rconfig.h header file). For example GCC needs to be version 3.4 or greater on Solaris.

In the info subdirectory of the package installation, information regarding changes to the Fusion SDK is stored, e.g.

```r
pathname <- system.file("info", "changes2fusion.txt", package="affxparser")
file.show(pathname)
```

**Acknowledgments**

We would like to thanks Ken Simpson (WEHI, Melbourne) and Seth Falcon (FHCRC, Seattle) for feedback and code contributions.

**License**

The releases of this package is licensed under LGPL version 2.1 or newer. This applies also to the Fusion SDK.

**Author(s)**

Henrik Bengtsson [aut], James Bullard [aut], Robert Gentleman [ctb], Kasper Daniel Hansen [aut, cre], Martin Morgan [ctb]

**References**

2. Cell coordinates and cell indices

Description

This part describes non-obvious terms used in this package.

**affxparser**  The name of this package.

**API**  Application program interface, which describes the functional interface of underlying methods.

**block**  (aka group).

**BPMAP**  A file format containing information related to the design of the tiling arrays.

**Calvin**  A special binary file format.

**CDF**  A file format: chip definition file.

**CEL**  A file format: cell intensity file.

**cell**  (aka feature)  A probe.

**cell index**  An integer that identifies a probe uniquely.

**chip**  An array.

**chip type**  An identifier specifying a chip design uniquely, e.g. "Mapping50K_Xba240".

**DAT**  A file format: contains pixel intensity values collected from an Affymetrix GeneArray scanner.

**feature**  A probe.

**Fusion SDK**  Open-source software development kit (SDK) provided by Affymetrix to access their data files.

**group**  (aka block)  Defines a unique subset of the cells in a unit. Expression arrays typically only have one group per unit, whereas SNP arrays have either two or four groups per unit, one for each of the two allele times possibly repeated for both strands.

**MM**  Mismatch-match, e.g. MM probe.

**PGF**  A file format: probe group file.

**TPMAP**  A file format storing the relationship between (PM,MM) pairs (or PM probes) and positions on a set of sequences.

**QC**  Quality control, e.g. QC probes and QC probe sets.

**unit**  A probeset.

**XDA**  A file format, aka as the binary file format.

---

2. Cell coordinates and cell indices

2. Cell coordinates and cell indices

Description

This part describes how Affymetrix cells, also known as probes or features, are addressed.
2. Cell coordinates and cell indices

Cell coordinates

In Affymetrix data files, cells are uniquely identified by their cell coordinates, i.e. \((x, y)\). For an array with \(N \times K\) cells in \(N\) rows and \(K\) columns, the \(x\) coordinate is an integer in \([0, K - 1]\), and the \(y\) coordinate is an integer in \([0, N - 1]\). The cell in the upper-left corner has coordinate \((x, y) = (0, 0)\) and the one in the lower-right corner \((x, y) = (K - 1, N - 1)\).

Cell indices and cell-index offsets

To simplify addressing of cells, a coordinate-to-index function is used so that each cell can be addressed using a single integer instead (of two). Affymetrix defines the cell index, \(i\), of cell \((x, y)\) as

\[
i = K \cdot y + x + 1,
\]

where one is added to give indices in \([1, N \times K]\). Continuing, the above definition means that cells are ordered row by row, that is from left to right and from top to bottom, starting at the upper-left corner. For example, with a chip layout \((N, K) = (1600, 1600)\) the cell at \((x, y) = (0, 0)\) has index \(i=1\), and the cell at \((x, y) = (1599, 1599)\) has index \(i = 2560000\). A cell at \((x, y) = (1498, 3)\) has index \(i = 6299\).

Given the cell index \(i\), the coordinate \((x, y)\) can be calculated as

\[
y = \text{floor}((i - 1)/K)
\]

\[
x = (i - 1) - K \cdot y.
\]

Continuing the above example, the coordinate for cell \(i = 1\) is found to be \((x, y) = (0, 0)\), for cell \(i = 2560000\) it is \((x, y) = (1599, 1599)\), for cell \(i = 6299\) it is \((x, y) = (1498, 3)\).

Converting between cell indices and \((x,y)\) coordinates in R

Although not needed to use the methods in this package, to get the cell indices for the cell coordinates or vice versa, see \texttt{xy2indices()} and \texttt{indices2xy()} in the \texttt{affy} package.

Note on the zero-based "index" field of Affymetrix CDF files

An Affymetrix CDF file provides information on which cells should be grouped together. To identify these groups of cells, the cells are specified by their \((x,y)\) coordinates, which are stored as zero-based coordinates in the CDF file.

All methods of the \texttt{affxparsr} package make use of these \((x,y)\) coordinates, and some methods make it possible to read them as well. However, it is much more common that the methods return cell indices calculated from the \((x,y)\) coordinates as explained above.

In order to conveniently work with cell indices in \(R\), the convention in \texttt{affxparsr} is to use one-based indices. Hence the addition (and subtraction) of 1:s in the above equations. This is all taken care of by \texttt{affxparsr}.

Note that, in addition to \((x,y)\) coordinates, a CDF file also contains a one-based "index" for each cell. This "index" is redundant to the \((x,y)\) coordinate and can be calculated analogously to the above cell index while leaving out the addition (subtraction) of 1:s. Importantly, since this "index" is redundant (and exists only in CDF files), we have decided to treat this field as an internal field. Methods of \texttt{affxparsr} do neither provide access to nor make use of this internal field.
Description

This part defines read and write maps that can be used to remap cell indices before reading and writing data from and to file, respectively.

This package provides methods to create read and write (cell-index) maps from Affymetrix CDF files. These can be used to store the cell data in an optimal order so that when data is read it is read in contiguous blocks, which is faster.

In addition to this, read maps may also be used to read CEL files that have been "reshuffled" by other software. For instance, the dChip software (http://www.dchip.org/) rotates Affymetrix Exon, Tiling and Mapping 500K data. See example below how to read such data "unrotated".

For more details how cell indices are defined, see 2. Cell coordinates and cell indices.

Motivation

When reading data from file, it is faster to read the data in the order that it is stored compared with, say, in a random order. The main reason for this is that the read arm of the hard drive has to move more if data is not read consecutively. Same applies when writing data to file. The read and write cache of the file system may compensate a bit for this, but not completely.

In Affymetrix CEL files, cell data is stored in order of cell indices. Moreover, (except for a few early chip types) Affymetrix randomizes the locations of the cells such that cells in the same unit (probeset) are scattered across the array. Thus, when reading CEL data arranged by units using for instance readCelUnits(), the order of the cells requested is both random and scattered.

Since CEL data is often queried unit by unit (except for some probe-level normalization methods), one can improve the speed of reading data by saving data such that cells in the same unit are stored together. A write map is used to remap cell indices to file indices. When later reading that data back, a read map is used to remap file indices to cell indices. Read and write maps are described next.

Definition of read and write maps

Consider cell indices $i = 1, 2, ..., N \times K$ and file indices $j = 1, 2, ..., N \times K$. A read map is then a bijective (one-to-one) function $h()$ such that

$$ i = h(j), $$

and the corresponding write map is the inverse function $h^{-1}()$ such that

$$ j = h^{-1}(i). $$
Since the mapping is required to be bijective, it holds that \( i = h^{-1}(h(j)) \) and that \( j = h^{-1}(h(i)) \). For example, consider the "reversing" read map function \( h(j) = N \times K - j + 1 \). The write map function is \( h^{-1}(i) = N \times K - i + 1 \). To verify the bijective property of this map, we see that
\[
 h(h^{-1}(i)) = h(N \times K - i + 1) = N \times K - (N \times K - i + 1) + 1 = i \text{ as well as } h^{-1}(h(j)) = h^{-1}(N \times K - j + 1) = N \times K - (N \times K - j + 1) + 1 = j.
\]

### Read and write maps in R

In this package, read and write maps are represented as integer vectors of length \( N \times K \) with unique elements in \( \{1, 2, ..., N \times K\} \). Consider cell and file indices as in previous section.

For example, the "reversing" read map in previous section can be represented as

```r
readMap <- (N*K):1
```

Given a vector \( j \) of file indices, the cell indices are the obtained as \( i = \text{readMap}[j] \). The corresponding write map is

```r
writeMap <- (N*K):1
```

and given a vector \( i \) of cell indices, the file indices are the obtained as \( j = \text{writeMap}[i] \).

Note also that the bijective property holds for this mapping, that is \( i == \text{readMap[writeMap[i]]} \) and \( i == \text{writeMap[readMap[i]]} \) are both TRUE.

Because the mapping is bijective, the write map can be calculated from the read map by:

```r
writeMap <- \text{order(readMap)}
```

and vice versa:

```r
readMap <- \text{order(writeMap)}
```

Note, the \text{invertMap()} \ method is much faster than \text{order()}.

Since most algorithms for Affymetrix data are based on probeset (unit) models, it is natural to read data unit by unit. Thus, to optimize the speed, cells should be stored in contiguous blocks of units. The methods \text{readCdfUnitsWriteMap()} \ can be used to generate a write map from a CDF file such that if the units are read in order, \text{readCelUnits()} \ will read the cells data in order. Example:

Find any CDF file
```r
cdfFile <- findCdf()
```

# Get the order of cell indices
```r
indices <- readCdfCellIndices(cdfFile)
indices <- unlist(indices, use.names=FALSE)
```

# Get an optimal write map for the CDF file
applyCdfGroupFields

writeMap <- readCdfUnitsWriteMap(cdfFile)

# Get the read map
readMap <- invertMap(writeMap)

# Validate correctness
indices2 <- readMap[indices]  # == 1, 2, 3, ..., N*K

Warning, do not misunderstand this example. It can not be used improve the reading speed of default CEL files. For this, the data in the CEL files has to be rearranged (by the corresponding write map).

**Reading rotated CEL files**

It might be that a CEL file was rotated by another software, e.g. the dChip software rotates Affymetrix Exon, Tiling and Mapping 500K arrays 90 degrees clockwise, which remains rotated when exported as CEL files. To read such data in a non-rotated way, a read map can be used to "unrotate" the data. The 90-degree clockwise rotation that dChip effectively uses to store such data is explained by:

```r
h <- readCdfHeader(cdfFile)
# (x,y) chip layout rotated 90 degrees clockwise
nrow <- h$cols
ncol <- h$rows
y <- (nrow-1):0
x <- rep(1:ncol, each=nrow)
writeMap <- as.vector(y*ncol + x)
```

Thus, to read this data "unrotated", use the following read map:

```r
readMap <- invertMap(writeMap)
data <- readCel(celFile, indices=1:10, readMap=readMap)
```

**Author(s)**

Henrik Bengtsson

---

**applyCdfGroupFields**

 Applies a function to a list of fields of each group in a CDF structure

**Description**

Applies a function to a list of fields of each group in a CDF structure.
applyCdfGroups

Usage
applyCdfGroupFields(cdf, fcn, ...)

Arguments

cdf A CDF list structure.
fcn A function that takes a list structure of fields and returns an updated list of fields.
... Arguments passed to the fcn function.

Value
Returns an updated CDF list structure.

Author(s)
Henrik Bengtsson

See Also
applyCdfGroups().

applyCdfGroups Applies a function over the groups in a CDF structure

Description
Applies a function over the groups in a CDF structure.

Usage
applyCdfGroups(cdf, fcn, ...)

Arguments

cdf A CDF list structure.
fcn A function that takes a list structure of group elements and returns an updated list of groups.
... Arguments passed to the fcn function.

Value
Returns an updated CDF list structure.
Pre-defined restructuring functions

• Generic:
  - `cdfGetFields()` - Gets a subset of groups fields in a CDF structure.
  - `cdfGetGroups()` - Gets a subset of groups in a CDF structure.
  - `cdfOrderBy()` - Orders the fields according to the value of another field in the same CDF group.
  - `cdfOrderColumnsBy()` - Orders the columns of fields according to the values in a certain row of another field in the same CDF group.

• Designed for SNP arrays:
  - `cdfAddBaseMmCounts()` - Adds the number of allele A and allele B mismatching nucleotides of the probes in a CDF structure.
  - `cdfAddProbeOffsets()` - Adds probe offsets to the groups in a CDF structure.
  - `cdfGtypeCelToPQ()` - Function to imitate Affymetrix' gtype_cel_to_pq software.
  - `cdfMergeAlleles()` - Function to join CDF allele A and allele B groups strand by strand.
  - `cdfMergeStrands()` - Function to join CDF groups with the same names.

We appreciate contributions.

Author(s)
Henrik Bengtsson

Examples

```r
if (require("AffymetrixDataTestFiles")) {
  # START #

  cdfFile <- findCdf("Mapping10K_Xba131")

  # Identify the unit index from the unit name
  unitName <- "SNP_A-1509436"
  unit <- which(readCdfUnitNames(cdfFile) == unitName)

  # Read the CDF file
  cdf0 <- readCdfUnits(cdfFile, units=unit, stratifyBy="pmmm", readType=FALSE, readDirection=FALSE)
  cat("Default CDF structure:\n"
  print(cdf0)

  # Tabulate the information in each group
  cdf <- applyCdfGroups(cdf, lapply, as.data.frame)
  print(cdf)

  # Infer the (true or the relative) offset for probe quartets.
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```
applyCdfGroups

cdf <- applyCdfGroups(cdf0, cdfAddProbeOffsets)
cat("Probe offsets:\n")
print(cdf)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Identify the number of nucleotides that mismatch the
# allele A and the allele B sequences, respectively.
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
cdf <- applyCdfGroups(cdf, cdfAddBaseMmCounts)
cat("Allele A & B target sequence mismatch counts:\n")
print(cdf)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Combine the signals from the sense and the anti-sense
# strands in a SNP CEL files.
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# First, join the strands in the CDF structure.
cdf <- applyCdfGroups(cdf, cdfMergeStrands)
cat("Joined CDF structure:\n")
print(cdf)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Rearrange values of group fields into quartets. This
# requires that the values are already arranged as PMs and MMs.
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
cdf <- applyCdfGroups(cdf0, cdfMergeAlleles)
cat("Probe quartets:\n")
print(cdf)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Get the x and y cell locations (note, zero-based)
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
x <- unlist(applyCdfGroups(cdf, cdfGetFields, "x"), use.names=FALSE)
y <- unlist(applyCdfGroups(cdf, cdfGetFields, "y"), use.names=FALSE)

# Validate
ncol <- readCdfHeader(cdfFile)$cols
cells <- as.integer(y*ncol+x+1)
cells <- sort(cells)
cells0 <- readCdfCellIndices(cdfFile, units=unit)
cells0 <- unlist(cells0, use.names=FALSE)
cells0 <- sort(cells0)

stopifnot(identical(cells0, cells))

##############################################################
} # STOP #
##############################################################
**arrangeCelFilesByChipType**

*Moves CEL files to subdirectories with names corresponding to the chip types*

**Description**

Moves CEL files to subdirectories with names corresponding to the chip types according to the CEL file headers. For instance, a HG_U95Av2 CEL file with pathname "data/foo.CEL" will be moved to subdirectory `celFiles/HG_U95Av2/`.

**Usage**

```r
arrangeCelFilesByChipType(pathnames=list.files(pattern = "[.](cel|CEL)$"), path="celFiles/", aliases=NULL, ...)
```

**Arguments**

- `pathnames`: A character vector of CEL pathnames to be moved.
- `path`: A character string specifying the root output directory, which in turn will contain chip-type subdirectories. All directories will be created, if missing.
- `aliases`: A named character string with chip type aliases. For instance, `aliases=c("Focus"="HG-Focus")` will treat a CEL file with chip type label 'Focus' (early-access name) as if it was 'HG-Focus' (official name).
- `...`: Not used.

**Value**

Returns (invisibly) a named character vector of the new pathnames with the chip types as the names. Files that could not be moved or where not valid CEL files are set to missing values.

**Author(s)**

Henrik Bengtsson

**See Also**

The chip type is inferred from the CEL file header, cf. `readCelHeader()`.
cdfAddBaseMmCounts

Add the number of allele A and allele B mismatching nucleotides of the probes in a CDF structure.

Description

Adds the number of allele A and allele B mismatching nucleotides of the probes in a CDF structure. This function is designed to be used with `applyCdfGroups()` on an Affymetrix Mapping (SNP) CDF list structure. Identifies the number of nucleotides (bases) in probe sequences that mismatch the target sequence for allele A and the allele B, as used by [1].

Usage

cdfAddBaseMmCounts(groups, ...)

Arguments

groups: A list structure with groups. Each group must contain the fields `tbase`, `pbase`, and `offset` (from `cdfAddProbeOffsets()`).

...: Not used.

Details

Note that the above counts can be inferred from the CDF structure alone, i.e. no sequence information is required. Consider a probe group interrogating allele A. First, all PM probes match the allele A target sequence perfectly regardless of shift. Moreover, all these PM probes mismatch the allele B target sequence at exactly one position. Second, all MM probes mismatch the allele A sequence at exactly one position. This is also true for the allele B sequence, except for an MM probe with zero offset, which only mismatch at one (the middle) position. For a probe group interrogating allele B, the same rules apply with labels A and B swapped. In summary, the mismatch counts for PM probes can take values 0 and 1, and for MM probes they can take values 0, 1, and 2.

Value

Returns a list structure with the same number of groups as the `groups` argument. To each group, two fields are added:

- `mmACount`: The number of nucleotides in the probe sequence that mismatches the target sequence of allele A.
- `mmBCount`: The number of nucleotides in the probe sequence that mismatches the target sequence of allele B.

Author(s)

Henrik Bengtsson
cdfAddPlasqTypes

Add the PLASQ types for the probes in a CDF structure

Description

Adds the PLASQ types for the probes in a CDF structure.

This function is design to be used with applyCdfGroups() on an Affymetrix Mapping (SNP) CDF list structure.

Usage

cdfAddPlasqTypes(groups, ...)

Arguments

groups A list structure with groups. Each group must contain the fields tbase, pbase, and expos.

... Not used.

Details

This function identifies the number of nucleotides (bases) in probe sequences that mismatch the target sequence for allele A and the allele B, as used by PLASQ [1], and adds an integer [0,15] interpreted as one of 16 probe types. In PLASQ these probe types are referred to as: 0=MMoBR, 1=MMoBF, 2=MMcBR, 3=MMcBF, 4=MMoAR, 5=MMoAF, 6=MMcAR, 7=MMcAF, 8=PMoBR, 9=PMoBF, 10=PMcBR, 11=PMcBF, 12=PMoAR, 13=PMoAF, 14=PMcAR, 15=PMcAF.

Pseudo rule for finding out the probe-type value:

- PM/MM: For MMs add 0, for PMs add 8.
- A/B: For Bs add 0, for As add 4.
- o/c: For shifted (o) add 0, for centered (c) add 2.

References


cdfAddProbeOffsets

- R/F: For antisense (R) add 0, for sense (F) add 1.

Example: \((PM, A, c, R) = 8 + 4 + 2 + 0 = 14 (=PMcAR)\)

Value

Returns a list structure with the same number of groups as the groups argument. To each group, one fields is added:

pla$\text{sqType}$  A vector of integers in \([0,15]\).

Author(s)

Henrik Bengtsson

References


cdfAddProbeOffsets  Adds probe offsets to the groups in a CDF structure

Description

Adds probe offsets to the groups in a CDF structure.

This function is design to be used with applyCdfGroups() on an Affymetrix Mapping (SNP) CDF list structure.

Usage

cdfAddProbeOffsets(groups, ...)

Arguments

groups  A list structure with groups. Each group must contain the fields tbase, and expos.

...  Not used.

Value

Returns a list structure with half the number of groups as the groups argument (since allele A and allele B groups have been joined).

Author(s)

Henrik Bengtsson
cdfGetFields

Gets a subset of groups fields in a CDF structure

Description

Gets a subset of groups fields in a CDF structure.
This function is designed to be used with applyCdfGroups().

Usage

cdfGetFields(groups, fields, ...)

Arguments

groups A list of groups.
fields A character vector of names of fields to be returned.
... Not used.

Details

Note that an error is not generated for missing fields. Instead the field is returned with value NA. The reason for this is that it is much faster.

Value

Returns a list structure of groups.

Author(s)

Henrik Bengtsson

See Also

applyCdfGroups().

References

cdfGetGroups

*Gets a subset of groups in a CDF structure*

Description

Gets a subset of groups in a CDF structure.

This function is designed to be used with `applyCdfGroups()`.

Usage

```
cdfGetGroups(groups, which, ...)
```

Arguments

- `groups`: A list of groups.
- `which`: An integer or character vector of groups be returned.
- `...`: Not used.

Value

Returns a list structure of groups.

Author(s)

Henrik Bengtsson

See Also

- `applyCdfGroups()`

---

cdfGtypeCelToPQ

*Function to imitate Affymetrix’ gtype_cel_to_pq software*

Description

Function to imitate Affymetrix’ gtype_cel_to_pq software.

This function is design to be used with `applyCdfGroups()` on an Affymetrix Mapping (SNP) CDF list structure.

Usage

```
cdfGtypeCelToPQ(groups, ...)
```
Arguments

- groups: A list structure with groups.
- ...: Not used.

Value

Returns a list structure with a single group. The fields in this groups are in turn vectors (all of equal length) where the elements are stored as subsequent quartets (PMA, MMA, PMB, MMB) with all forward-strand quartets first followed by all reverse-strand quartets.

Author(s)

Henrik Bengtsson

References


See Also

applyCdfGroups().

cdfHeaderToCelHeader

*Creates a valid CEL header from a CDF header*

Description

Creates a valid CEL header from a CDF header.

Usage

cdfHeaderToCelHeader(cdfHeader, sampleName="noname", date=Sys.time(), ..., version="4")

Arguments

- cdfHeader: A CDF list structure.
- sampleName: The name of the sample to be added to the CEL header.
- date: The (scan) date to be added to the CEL header.
- ...: Not used.
- version: The file-format version of the generated CEL file. Currently only version 4 is supported.

Value

Returns a CDF list structure.
Function to join CDF allele A and allele B groups strand by strand

Description
Function to join CDF allele A and allele B groups strand by strand.
This function is design to be used with applyCdfGroups() on an Affymetrix Mapping (SNP) CDF list structure.

Usage
cdfMergeAlleles(groups, compReverseBases=FALSE, collapse="", ...)  

Arguments
- **groups**: A list structure with groups.
- **compReverseBases**: If TRUE, the group names, which typically are names for bases, are turned into their complementary bases for the reverse strand.
- **collapse**: The character string used to collapse the allele A and the allele B group names.
- **...**: Not used.

Details
Allele A and allele B are merged into a matrix where first row hold the elements for allele A and the second elements for allele B.

Value
Returns a list structure with the two groups forward and reverse, if the latter exists.

Author(s)
Henrik Bengtsson

References

See Also
applyCdfGroups().
cdfMergeStrands  

Function to join CDF groups with the same names

Description

Function to join CDF groups with the same names.

This function is designed to be used with applyCdfGroups() on an Affymetrix Mapping (SNP) CDF list structure.

This can be used to join the sense and anti-sense groups of the same allele in SNP arrays.

Usage

cdfMergeStrands(groups, ...)

Arguments

- groups: A list structure with groups.
- ...: Not used.

Details

If a unit has two strands, they are merged such that the elements for the second strand are concatenated to the end of the elements of the first strand (This is done separately for the two alleles).

Value

Returns a list structure with only two groups.

Author(s)

Henrik Bengtsson

References


See Also

applyCdfGroups().
cdfMergeToQuartets

Function to re-arrange CDF groups values in quartets

Description

Function to re-arrange CDF groups values in quartets.

This function is design to be used with applyCdfGroups() on an Affymetrix Mapping (SNP) CDF list structure.

Note, this requires that the group values have already been arranged in PMs and MMs.

Usage

cdfMergeToQuartets(groups, ...)

Arguments

  groups    A list structure with groups.
  ...       Not used.

Value

Returns a list structure with the two groups forward and reverse, if the latter exists.

Author(s)

Henrik Bengtsson

References


See Also

applyCdfGroups().
cdfOrderBy

Orders the fields according to the value of another field in the same CDF group

Description
Orders the fields according to the value of another field in the same CDF group. This function is design to be used with applyCdfGroups() on an Affymetrix Mapping (SNP) CDF list structure.

Usage
cdfOrderBy(groups, field, ...)

Arguments
- groups: A list of groups.
- field: The field whose values are used to order the other fields.
- ...: Optional arguments passed order().

Value
Returns a list structure of groups.

Author(s)
Henrik Bengtsson

See Also
cdfOrderColumnsBy(). applyCdfGroups().

cdfOrderColumnsBy

Orders the columns of fields according to the values in a certain row of another field in the same CDF group

Description
Orders the columns of fields according to the values in a certain row of another field in the same CDF group. Note that this method requires that the group fields are matrices. This function is design to be used with applyCdfGroups() on an Affymetrix Mapping (SNP) CDF list structure.
cdfSetDimension

Usage

cdfOrderColumnsBy(groups, field, row=1, ...)

Arguments

  groups     A list of groups.
  field      The field whose values in row row are used to order the other fields.
  row        The row of the above field to be used to find the order.
  ...        Optional arguments passed order().

Value

  Returns a list structure of groups.

Author(s)

  Henrik Bengtsson

See Also

  cdfOrderBy(). applyCdfGroups().

_____________________________________________________

cdfSetDimension    Sets the dimension of an object

_____________________________________________________

Description

  Sets the dimension of an object.
  This function is designed to be used with applyCdfGroupFields().

Usage

  cdfSetDimension(field, dim, ...)

Arguments

  field        An R object.
  dim          An integer vector.
  ...          Not used.

Value

  Returns a list structure of groups.

Author(s)

  Henrik Bengtsson
compareCdfs

Compares the contents of two CDF files

Description

Compares the contents of two CDF files.

Usage

compareCdfs(pathname, other, quick=FALSE, verbose=0, ...)

Arguments

pathname       The pathname of the first CDF file.
other          The pathname of the seconds CDF file.
quick          If TRUE, only a subset of the units are compared, otherwise all units are compared.
verbose        An integer. The larger the more details are printed.
...             Not used.

Details

The comparison is done with an upper-limit memory usage, regardless of the size of the CDFs.

Value

Returns TRUE if the two CDF are equal, otherwise FALSE. If FALSE, the attribute reason contains a string explaining what difference was detected, and the attributes value1 and value2 contain the two objects/values that differs.

Author(s)

Henrik Bengtsson

See Also

convertCdf().
**compareCels**

---

**compareCels**

*Compares the contents of two CEL files*

**Description**

Compares the contents of two CEL files.

**Usage**

```r
compareCels(pathname, other, readMap=NULL, otherReadMap=NULL, verbose=0, ...)
```

**Arguments**

- `pathname` The pathname of the first CEL file.
- `other` The pathname of the second CEL file.
- `readMap` An optional read map for the first CEL file.
- `otherReadMap` An optional read map for the second CEL file.
- `verbose` An `integer`. The larger the more details are printed.
- `...` Not used.

**Value**

Returns `TRUE` if the two CELs are equal, otherwise `FALSE`. If `FALSE`, the attribute `reason` contains a string explaining what difference was detected, and the attributes `value1` and `value2` contain the two objects/values that differs.

**Author(s)**

Henrik Bengtsson

**See Also**

`convertCel()`.
**convertCdf**

*Converts a CDF into the same CDF but with another format*

**Description**

Converts a CDF into the same CDF but with another format. Currently only CDF files in version 4 (binary/XDA) can be written. However, any input format is recognized.

**Usage**

```r
classNames = convertCdf(filename, outFilename, version="4", force=FALSE, ..., .validate=TRUE, verbose=FALSE)
```

**Arguments**

- `filename`: The pathname of the original CDF file.
- `outFilename`: The pathname of the destination CDF file. If the same as the source file, an exception is thrown.
- `version`: The version of the output file format.
- `force`: If `FALSE`, and the version of the original CDF is the same as the output version, the new CDF will not be generated, otherwise it will.
- `...`: Not used.
- `.validate`: If `TRUE`, a consistency test between the generated and the original CDF is performed. Note that the memory overhead for this can be quite large, because two complete CDF structures are kept in memory at the same time.
- `verbose`: If `TRUE`, extra details are written while processing.

**Value**

Returns (invisibly) `TRUE` if a new CDF was generated, otherwise `FALSE`.

**Benchmarking of ASCII and binary CDFs**

Binary CDFs are much faster to read than ASCII CDFs. Here are some example for reading complete CDFs (the difference is even larger when reading CDFs in subsets):

- HG-U133A (22283 units): ASCII 11.7s (9.3x), binary 1.20s (1x).
- Hu6800 (7129 units): ASCII 3.5s (6.1x), binary 0.57s (1x).

**Confirmed conversions to binary (XDA) CDFs**

The following chip types have been converted using `convertCdf()` and then verified for correctness using `compareCdfs()`: ASCII-to-binary: HG-U133A, Hu6800. Binary-to-binary: Test3.

**Author(s)**

Henrik Bengtsson
convertCel

See Also
See `compareCdfs()` to compare two CDF files. `writeCdf()`.

Examples

```r
if (require("AffymetrixDataTestFiles")) {
  chipType <- "Test3"
  cdfFiles <- findCdf(chipType, firstOnly=FALSE)
  cdfFiles <- list(
    ASCII=grep("ASCII", cdfFiles, value=TRUE),
    XDA=grep("XDA", cdfFiles, value=TRUE)
  )
  outFile <- file.path(tempdir(), sprintf("%s.cdf", chipType))
  convertCdf(cdfFiles$ASCII, outFile, verbose=TRUE)
}
```

convertCel

Converts a CEL into the same CEL but with another format

Description

Converts a CEL into the same CEL but with another format. Currently only CEL files in version 4 (binary/XDA) can be written. However, any input format is recognized.

Usage

```r
convertCel(filename, outFile, readMap=NULL, writeMap=NULL, version="4",
newChipType=NULL, ..., .validate=FALSE, verbose=FALSE)
```

Arguments

- **filename**: The pathname of the original CEL file.
- **outFile**: The pathname of the destination CEL file. If the same as the source file, an exception is thrown.
- **readMap**: An optional read map for the input CEL file.
- **writeMap**: An optional write map for the output CEL file.
- **version**: The version of the output file format.
newChipType (Only for advanced users who fully understands the Affymetrix CEL file format!) An optional string for overriding the chip type (label) in the CEL file header.

... Not used.

.validate If TRUE, a consistency test between the generated and the original CEL is performed.

verbose If TRUE, extra details are written while processing.

Value

Returns (invisibly) TRUE if a new CEL was generated, otherwise FALSE.

Benchmarking of ASCII and binary CELs

Binary CELs are much faster to read than ASCII CELs. Here are some example for reading complete CELs (the difference is even larger when reading CELs in subsets):

- To do

WARNING: Changing the chip type label

The newChipType argument changes the label in the part of DAT header that specifies the chip type of the CEL file. Note that it does not change anything else in the CEL file. This type of relabeling is valid for updating the chip type label of CEL files that where generated during, say, an "Early Access" period leading to a different chip type label than what more recent CEL files of the same physical chip type have.

Author(s)

Henrik Bengtsson

See Also

createCel().

Examples

```r
if (require("AffymetrixDataTestFiles")) {
  # START #
  # Search for some available Calvin CEL files
  path <- system.file("rawData", package="AffymetrixDataTestFiles")
  files <- findFiles(pattern="[.]\(cel\|CEL\)$", path=path, recursive=TRUE, firstOnly=FALSE)
  files <- grep("FusionSDK_Test3", files, value=TRUE)
  files <- grep("Calvin", files, value=TRUE)
  file <- files[1]

  outFile <- file.path(tempdir(), gsub("[.]CEL\$", ",XBA.CEL", basename(file)))
  ```
if (file.exists(outFile))
  file.remove(outFile)
convertCel(file, outFile, .validate=TRUE)

##############################################################
}  # STOP #
##############################################################

---

### copyCel

**Copies a CEL file**

**Description**

Copies a CEL file.

The file must be a valid CEL file, if not an exception is thrown.

**Usage**

```r
copyCel(from, to, overwrite=FALSE, ...)
```

**Arguments**

- **from**: The filename of the CEL file to be copied.
- **to**: The filename of destination file.
- **overwrite**: If `FALSE` and the destination file already exists, an exception is thrown, otherwise not.
- `...`: Not used.

**Value**

Return `TRUE` if file was successfully copied, otherwise `FALSE`.

**Author(s)**

Henrik Bengtsson

**See Also**

`isCelFile()`
createCel

Creates an empty CEL file.

Usage

createCel(filename, header, nsubgrids=0, overwrite=FALSE, ..., cdf=NULL, verbose=FALSE)

Arguments

filename: The filename of the CEL file to be created.
header: A list structure describing the CEL header, similar to the structure returned by readCellHeader(). This header can be of any CEL header version.
overwrite: If FALSE and the file already exists, an exception is thrown, otherwise the file is created.
nsubgrids: The number of subgrids.
...: Not used.
cdf: (optional) The pathname of a CDF file for the CEL file to be created. If given, the CEL header (argument header) is validated against the CDF header, otherwise not. If TRUE, a CDF file is located automatically based using findCdf(header$chiptype).
verbose: An integer specifying how much verbose details are outputted.

Details

Currently only binary (v4) CEL files are supported. The current version of the method does not make use of the Fusion SDK, but its own code to create the CEL file.

Value

Returns (invisibly) the pathname of the file created.

Redundant fields in the CEL header

There are a few redundant fields in the CEL header. To make sure the CEL header is consistent, redundant fields are cleared and regenerated. For instance, the field for the total number of cells is calculated from the number of cell rows and columns.

Author(s)

Henrik Bengtsson
findCdf

Examples

if (require("AffymetrixDataTestFiles")) {
  # START #
  #########

  # Search for first available ASCII CEL file
  path <- system.file("rawData", package="AffymetrixDataTestFiles")
  files <- findFiles(pattern="^[.]\(cel\|CEL\)$", path=path, recursive=TRUE, firstOnly=FALSE)
  files <- grep("ASCII", files, value=TRUE)
  file <- files[1]

  # Read the CEL header
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  hdr <- readCelHeader(file)

  # Assert that we found an ASCII CEL file, but any will do
  stopifnot(hdr$version == 3)

  # Create a CEL v4 file of the same chip type
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  outFile <- file.path(tempdir(), "zzz.CEL")
  if (file.exists(outFile))
    file.remove(outFile)
  createCel(outFile, hdr, overwrite=TRUE)
  str(readCelHeader(outFile))

  # Verify correctness by update and re-read a few cells
  intensities <- as.double(1:100)
  indices <- seq(along=intensities)
  updateCel(outFile, indices=indices, intensities=intensities)
  value <- readCel(outFile, indices=indices)$intensities
  stopifnot(identical(intensities, value))

  #########
  # STOP #
  # - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -#
}

# END #

findCdf

Search for CDF files in multiple directories

Description

Search for CDF files in multiple directories.
findCdf

Usage

findCdf(chipType=NULL, paths=NULL, recursive=TRUE, pattern="\.[cC][dD][fF]", ...)  

Arguments

chipType A character string of the chip type to search for.
paths A character vector of paths to be searched. The current directory is always searched at the beginning. If NULL, default paths are searched. For more details, see below.
recursive If TRUE, directories are searched recursively.
pattern A regular expression file name pattern to match.
... Additional arguments passed to findFiles().

Details

Note, the current directory is always searched first, but never recursively (unless it is added to the search path explicitly). This provides an easy way to override other files in the search path.

If paths is NULL, then a set of default paths are searched. The default search path constitutes:

1. getOption("AFFX_CDF_PATH")
2. Sys.getenv("AFFX_CDF_PATH")

One of the easiest ways to set system variables for R is to set them in an .Renviron file, e.g.

# affxparser: Set default CDF path
AFFX_CDF_PATH=${AFFX_CDF_PATH};M:/Affymetrix_2004-100k_trios/cdf
AFFX_CDF_PATH=${AFFX_CDF_PATH};M:/Affymetrix_2005-500k_data/cdf

See Startup for more details.

Value

Returns a vector of the full pathnames of the files found.

Author(s)

Henrik Bengtsson

See Also

This method is used internally by readCelUnits() if the CDF file is not specified.
**findFiles**

Finds one or several files in multiple directories

**Description**

Finds one or several files in multiple directories.

**Usage**

```r
findFiles(pattern=NULL, paths=NULL, recursive=FALSE, firstOnly=TRUE, allFiles=TRUE, ...)```

**Arguments**

- `pattern` A regular expression file name pattern to match.
- `paths` A character vector of paths to be searched.
- `recursive` If TRUE, the directory structure is searched breath-first, in lexicographic order.
- `firstOnly` If TRUE, the method returns as soon as a matching file is found, otherwise not.
- `allFiles` If FALSE, files and directories starting with a period will be skipped, otherwise not.
- `...` Arguments passed to `list.files()`.

**Examples**

```r
if (require("AffyMatrixDataTestFiles")) {
  # START #
  # Find a specific CDF file
  cdfFile <- findCdf("Mapping10K_Xba131")
  print(cdfFile)

  # Find the first CDF file (no matter what it is)
  cdfFile <- findCdf()
  print(cdfFile)

  # Find all CDF files in search path and display their headers
  cdfFiles <- findCdf(firstOnly=FALSE)
  for (cdfFile in cdfFiles) {
    cat("=======================================\n    hdr <- readCdfHeader(cdfFile)
    str(hdr)
  }
  # STOP #
  # STOP #
} # STOP #
```
invertMap

Description

Inverts a read or a write map.

Usage

invertMap(map, ...)

Arguments

map An integer vector.

... Not used.

Details

An map is defined to be a vector of \( n \) with unique finite values in \([1, n]\). Finding the inverse of a map is the same as finding the rank of each element, cf. order(). However, this method is much faster, because it utilizes the fact that all values are unique and in \([1, n]\). Moreover, for any map it holds that taking the inverse twice will result in the same map.

Value

Returns an integer vector.

Author(s)

Henrik Bengtsson
See Also

To generate an optimized write map for a CDF file, see `readCdfUnitsWriteMap()`.

Examples

```r
set.seed(1)

# Simulate a read map for a chip with 1.2 million cells
nbrOfCells <- 1200000
readMap <- sample(nbrOfCells)

# Get the corresponding write map
writeMap <- invertMap(readMap)

# A map inverted twice should be equal itself
stopifnot(identical(invertMap(writeMap), readMap))

# Another example illustrating that the write map is the
# inverse of the read map
idx <- sample(nbrOfCells, size=1000)
stopifnot(identical(writeMap[readMap[idx]], idx))

# invertMap() is much faster than order()
t1 <- system.time(invertMap(readMap))[3]
cat(sprintf("invertMap() : %5.2fs [ 1.00x]\n", t1))

t2 <- system.time(writeMap2 <- sort.list(readMap, na.last=NA, method="quick"))[3]
cat(sprintf("'quick sort' : %5.2fs [%5.2fx]\n", t2, t2/t1))

# Clean up
rm(nbrOfCells, idx, readMap, writeMap, writeMap2)
```

---

### isCelFile

**Checks if a file is a CEL file or not**

**Description**

Checks if a file is a CEL file or not.

**Usage**

```r
isCelFile(filename, ...)
```
parseDatHeaderString

Arguments

filename A filename.
...

Value

Returns TRUE if a CEL file, otherwise FALSE. ASCII (v3), binary (v4;XDA), and binary (CCG v1;Calvin) CEL files are recognized. If file does not exist, an exception is thrown.

Author(s)

Henrik Bengtsson

See Also

readCel(), readCelHeader(), readCelUnits().

parseDatHeaderString  Parses a DAT header string

Description

Parses a DAT header string.

Usage

parseDatHeaderString(header, timeFormat="%m/%d/%y %H:%M:%S", ...)

Arguments

header A character string.
timeFormat The format string used to parse the timestamp. For more details, see strftime(). If NULL, no parsing is done.
...

Value

Returns named list structure.

Author(s)

Henrik Bengtsson

See Also

readCelHeader().
readBpmap

Parses a Bpmap file

Description

Parses (parts of) a Bpmap (binary probe mapping) file from Affymetrix.

Usage

readBpmap(filename, seqIndices = NULL, readProbeSeq = TRUE, readSeqInfo = TRUE, readPMXY = TRUE, readMMXY = TRUE, readStartPos = TRUE, readCenterPos = FALSE, readStrand = TRUE, readMatchScore = FALSE, readProbeLength = FALSE, verbose = 0)

readBpmapHeader(filename)

readBpmapSeqinfo(filename, seqIndices = NULL, verbose = 0)

Arguments

filename The filename as a character.
seqIndices A vector of integers, detailing the indices of the sequences being read. If NULL, the entire file is being read.
readProbeSeq Do we read the probe sequences.
readSeqInfo Do we read the sequence information (a list containing information such as sequence name, number of hits etc.)
readPMXY Do we read the (x,y) coordinates of the PM-probes.
readMMXY Do we read the (x,y) coordinates of the MM-probes (only relevant if the file has MM information)
readStartPos Do we read the start position of the probes.
readCenterPos Do we return the start position of the probes.
readStrand Do we return the strand of the hits.
readMatchScore Do we return the matchscore.
readProbeLength Do we return the probelength.
verbose How verbose do we want to be.

Details

readBpmap reads a BPMAP file, which is a binary file containing information about a given probe’s location in a sequence. Here sequence means some kind of reference sequence, typically a chromosome or a scaffold. readBpmapHeader reads the header of the BPMAP file, and readBpmapSeqinfo reads the sequence info of the sequences (so this function is merely a convenience function).
Value

For readBpmap: A list of lists, one list for every sequence read. The components of the sequence lists, depends on the argument of the function call. For readBpmapheader a list with two components version and numSequences. For readBpmapSeqinfo a list of lists containing the sequence info.

Author(s)

Kasper Daniel Hansen

See Also

tpm2bmap for information on how to write Bpmap files.

readCcg

Reads an Affymetrix Command Console Generic (CCG) Data file

Description

Reads an Affymetrix Command Console Generic (CCG) Data file. The CCG data file format is also known as the Calvin file format.

Usage

readCcg(pathname, verbose=0, .filter=NULL, ...)

Arguments

pathname
  The pathname of the CCG file.
verbose
  An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.
.filter
  A list.
...
  Not used.

Details

Note, the current implementation of this methods does not utilize the Affymetrix Fusion SDK library. Instead, it is implemented in R from the file format definition [1].

Value

A named list structure consisting of ...
About the CCG file format

A CCG file, consists of a "file header", a "generic data header", and "data" section, as outlined here:

- File Header
- Generic Data Header (for the file)
  1. Generic Data Header (for the files 1st parent)
     (a) Generic Data Header (for the files 1st parents 1st parent)
     (b) Generic Data Header (for the files 1st parents 2nd parent)
     (c) ...
     (d) Generic Data Header (for the files 1st parents Mth parent)
  2. Generic Data Header (for the files 2nd parent)
  3. ...
  4. Generic Data Header (for the files Nth parent)
- Data
  1. Data Group #1
     (a) Data Set #1
        – Parameters
        – Column definitions
        – Matrix of data
     (b) Data Set #2
     (c) ...
     (d) Data Set #L
  2. Data Group #2
  3. ...
  4. Data Group #K

Author(s)
Henrik Bengtsson

References

See Also
readCcgHeader(), readCdfUnits().
readCcgHeader

Reads an the header of an Affymetrix Command Console Generic (CCG) file.

Description

Reads an the header of an Affymetrix Command Console Generic (CCG) file.

Usage

```r
readCcgHeader(pathname, verbose=0, .filter=list(fileHeader = TRUE, dataHeader = TRUE), ...)
```

Arguments

- `pathname` The pathname of the CCG file.
- `verbose` An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.
- `.filter` A list.
- `...` Not used.

Details

Note, the current implementation of this methods does not utilize the Affymetrix Fusion SDK library. Instead, it is implemented in R from the file format definition [1].

Value

A named list structure consisting of ...

Author(s)

Henrik Bengtsson

References


See Also

`readCcg()`
readCdf

Parsing a CDF file using Affymetrix Fusion SDK

Description

Parsing a CDF file using Affymetrix Fusion SDK. This function parses a CDF file using the
Affymetrix Fusion SDK. This function will most likely be replaced by the more general readCdfUnits() function.

Usage

readCdf(filename, units=NULL,
readXY=TRUE, readBases=TRUE,
readIndexpos=TRUE, readAtoms=TRUE,
readUnitType=TRUE, readUnitDirection=TRUE,
readUnitNumber=TRUE, readUnitAtomNumbers=TRUE,
readGroupAtomNumbers=TRUE, readGroupDirection=TRUE,
readIndices=FALSE, readIsPm=FALSE,
stratifyBy=c("nothing", "pmmm", "pm", "mm"),
verbose=0)

Arguments

filename The filename of the CDF file.
units An integer vector of unit indices specifying which units to be read. If NULL, all units are read.
readXY If TRUE, cell row and column (x,y) coordinates are retrieved, otherwise not.
readBases If TRUE, cell P and T bases are retrieved, otherwise not.
readIndexpos If TRUE, cell indexpos are retrieved, otherwise not.
readUnitType If TRUE, unit types are retrieved, otherwise not.
readUnitDirection If TRUE, unit directions are retrieved, otherwise not.
readUnitNumber If TRUE, unit numbers are retrieved, otherwise not.
readUnitAtomNumbers If TRUE, unit atom numbers are retrieved, otherwise not.
readGroupAtomNumbers If TRUE, group atom numbers are retrieved, otherwise not.
readGroupDirection If TRUE, group directions are retrieved, otherwise not.
readIndices If TRUE, cell indices calculated from the row and column (x,y) coordinates are retrieved, otherwise not. Note that these indices are one-based.
readIsPm If TRUE, cell flags indicating whether the cell is a perfect-match (PM) probe or not are retrieved, otherwise not.
stratifyBy A character string specifying which and how elements in group fields are returned. If "nothing", elements are returned as is, i.e. as vectors. If "pm"/"mm", only elements corresponding to perfect-match (PM) / mismatch (MM) probes are returned (as vectors). If "pmmm", elements are returned as a matrix where the first row holds elements corresponding to PM probes and the second corresponding to MM probes. Note that in this case, it is assumed that there are equal number of PMs and MMs; if not, an error is generated. Moreover, the PMs and MMs may not even be paired, i.e. there is no guarantee that the two elements in a column corresponds to a PM-MM pair.

verbose An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.

Value
A list with one component for each unit. Every component is again a list with three components

groups This is again a list with one component for each group (also called block). The information on each group is a list with 5 components, x, y, pbase, tbase, expos.
type type of the unit.
direction direction of the unit.

Cell indices are one-based
Note that in affxparser all cell indices are by convention one-based, which is more convenient to work with in R. For more details on one-based indices, see 2. Cell coordinates and cell indices.

Note
This version of the function does not return information on the QC probes. This will be added in a (near) future release. In addition we expect the header to be part of the returned object.
So expect changes to the structure of the value of the function in next release. Please contact the developers for details.

Author(s)
James Bullard and Kasper Daniel Hansen.

References

See Also
It is recommended to use readCdfUnits() instead of this method. readCdfHeader() for getting the header of a CDF file.
readCdfCellIndices  

*Reads (one-based) cell indices of units (probesets) in an Affymetrix CDF file*

**Description**

Reads (one-based) cell indices of units (probesets) in an Affymetrix CDF file.

**Usage**

```r
readCdfCellIndices(filename, units=NULL, stratifyBy=c("nothing", "pmmm", "pm", "mm"), verbose=0)
```

**Arguments**

- `filename`  
  The filename of the CDF file.

- `units`   
  An integer vector of unit indices specifying which units to be read. If `NULL`, all units are read.

- `stratifyBy`   
  A character string specifying which and how elements in group fields are returned. If "nothing", elements are returned as is, i.e. as vectors. If "pm"/"mm", only elements corresponding to perfect-match (PM) / mismatch (MM) probes are returned (as vectors). If "pmmm", elements are returned as a matrix where the first row holds elements corresponding to PM probes and the second corresponding to MM probes. Note that in this case, it is assumed that there are equal number of PMs and MMs; if not, an error is generated. Moreover, the PMs and MMs may not even be paired, i.e. there is no guarantee that the two elements in a column corresponds to a PM-MM pair.

- `verbose`  
  An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.

**Value**

A named list where the names corresponds to the names of the units read. Each unit element of the list is in turn a list structure with one element groups which in turn is a list. Each group element in groups is a list with a single field named indices. Thus, the structure is

```r
cdf
  +- unit #1
    |  +- "groups"
    |    +- group #1
    |    |    +- "indices"
    |    +- group #2
    |    |    +- "indices"
    |    .
    |  +- group #K
    |     +- "indices"
```

This is structure is compatible with what `readCdfUnits()` returns.
Note that these indices are one-based.

**Cell indices are one-based**

Note that in `affxparser` all cell indices are by convention one-based, which is more convenient to work with in R. For more details on one-based indices, see 2. Cell coordinates and cell indices.

**Author(s)**

Henrik Bengtsson

**See Also**

`readCdfUnits()`.

---

**readCdfDataFrame**

Reads units (probesets) from an Affymetrix CDF file

**Description**

Reads units (probesets) from an Affymetrix CDF file. Gets all or a subset of units (probesets).

**Usage**

`readCdfDataFrame(filename, units=NULL, groups=NULL, cells=NULL, fields=NULL, drop=TRUE, verbose=0)`

**Arguments**

- `filename` The filename of the CDF file.
- `units` An integer vector of unit indices specifying which units to be read. If NULL, all are read.
- `groups` An integer vector of group indices specifying which groups to be read. If NULL, all are read.
- `cells` An integer vector of cell indices specifying which cells to be read. If NULL, all are read.
- `fields` A character vector specifying what fields to read. If NULL, all unit, group and cell fields are returned.
- `drop` If TRUE and only one field is read, then a vector (rather than a single-column data.frame) is returned.
- `verbose` An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.
Value

An \texttt{NxK data.frame} or a \texttt{vector} of length N.

Author(s)

Henrik Bengtsson

References


See Also

For retrieving the CDF as a \texttt{list} structure, see \texttt{readCdfUnits}.

Examples

```r
# Find any CDF file
cdfFile <- findCdf()

units <- 101:120
fields <- c("unit", "unitName", "group", "groupName", "cell")
df <- readCdfDataFrame(cdfFile, units=units, fields=fields)
stopifnot(identical(sort(unique(df$unit)), units))

fields <- c("unit", "unitName", "unitType")
fields <- c(fields, "group", "groupName")
fields <- c(fields, "x", "y", "cell", "pbase", "tbase")
df <- readCdfDataFrame(cdfFile, units=units, fields=fields)
stopifnot(identical(sort(unique(df$unit)), units))
```
**Description**

Reads group names for a set of units (probesets) in an Affymetrix CDF file.
This is for instance useful for SNP arrays where the nucleotides used for the A and B alleles are the same as the group names.

**Usage**

```r
readCdfGroupNames(filename, units=NULL, truncateGroupNames=TRUE, verbose=0)
```

**Arguments**

- `filename` The filename of the CDF file.
- `units` An integer vector of unit indices specifying which units to be read. If NULL, all units are read.
- `truncateGroupNames` A logical variable indicating whether unit names should be stripped from the beginning of group names.
- `verbose` An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.

**Value**

A named list structure where the names of the elements are the names of the units read. Each element is a character vector with group names for the corresponding unit.

**Author(s)**

Henrik Bengtsson

**See Also**

- `readCdfUnits()`

---

**readCdfHeader**

*Reads the header associated with an Affymetrix CDF file*

**Description**

Reads the header of an Affymetrix CDF file using the Fusion SDK.

**Usage**

```r
readCdfHeader(filename)
```

**Arguments**

- `filename` name of the CDF file.
**Value**

A named list with the following components:

- **rows**: the number of rows on the chip.
- **cols**: the number of columns on the chip.
- **probesets**: the number of probesets on the chip.
- **qcprobesets**: the number of QC probesets on the chip.
- **reference**: the reference sequence (this component only exists for resequencing chips).
- **chiptype**: the type of the chip.
- **filename**: the name of the cdf file.

**Author(s)**

James Bullard and Kasper Daniel Hansen

**See Also**

- `readCdfUnits()`.

**Examples**

```r
for (zzz in 0) {

  # Find any CDF file
  cdfFile <- findCdf()
  if (is.null(cdfFile))
    break

  header <- readCdfHeader(cdfFile)
  print(header)
}
```

---

**readCdfIsPm**  
*Checks if cells in a CDF file are perfect-match probes or not*

**Description**

Checks if cells in a CDF file are perfect-match probes or not.

**Usage**

`readCdfIsPm(filename, units=NULL, verbose=0)`
readCdfNbrOfCellsPerUnitGroup

Description

Gets the number of cells (probes) that each group of each unit in a CDF file.

Usage

readCdfNbrOfCellsPerUnitGroup(filename, units=NULL, verbose=0)

Arguments

filename  The filename of the CDF file.
units  An integer vector of unit indices specifying which units to be read. If NULL, all units are read.
verbose  An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.

Value

A named list of named integer vectors. The name of the list elements are unit names and the names of the integer vector are group names.

Author(s)

Henrik Bengtsson
readCdfNbrOfCellsPerUnitGroup

Examples

```r
if (require("AffymetrixDataTestFiles")) {
  # START #
  cdfFile <- findCdf("Mapping10K_Xba131")
  groups <- readCdfNbrOfCellsPerUnitGroup(cdfFile)

  # Number of units read
  print(length(groups))
  ## 11564

  # Details on two units
  print(groups[56:57])
  ##:
  ## SNP_A-1516438C SNP_A-1516438T SNP_A-1516438C SNP_A-1516438T
  ## 10 10 10 10
  ##:
  ## SNP_A-1508602A SNP_A-1508602G SNP_A-1508602A SNP_A-1508602G
  ## 10 10 10 10

  # Number of groups with different number of cells
  print(table(unlist(groups)))
  ## 10 60
  ## 46240 4

  # Number of cells per unit
  nbrOfCellsPerUnit <- unlist(lapply(groups, FUN=sum))
  print(table(nbrOfCellsPerUnit))
  nbrOfCellsPerUnit
  ## 40 60
  ## 11560 4

  # Number of groups per unit
  nbrOfGroupsPerUnit <- unlist(lapply(groups, FUN=length))

  # Details on a few units
  print(nbrOfGroupsPerUnit[20:30])
  ## SNP_A-1512666 SNP_A-1512740 SNP_A-1512132 SNP_A-1516082 SNP_A-1511962
  ## 4 4 4 4 4
  ## SNP_A-1515637 SNP_A-1515878 SNP_A-1518789 SNP_A-1518296 SNP_A-1519701
  ## 4 4 4 4 4
  ## SNP_A-1511743
  ## 4

  # Number of units for each unique number of groups
  print(table(nbrOfGroupsPerUnit))
```

```
## nbrOfGroupsPerUnit

<table>
<thead>
<tr>
<th>Size</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>11560</td>
</tr>
</tbody>
</table>

```r
x <- list()
for (size in unique(nbrOfGroupsPerUnit)) {
  subset <- groups[nbrOfGroupsPerUnit==size]
  t <- matrix(unlist(subset), nrow=size)
  colnames(t) <- names(subset)
  x[[as.character(size)]] <- t
  rm(subset, t)
}

# Check if there are any quartet units where the number
# of cells in Group 1 & 2 or Group 3 & 4 does not have
# the same number of cells.
# Group 1 & 2
print(sum(x[['4']][1,]-x[['4']][2,] != 0))
# 0

# Group 3 & 4
print(sum(x[['4']][3,]-x[['4']][4,] != 0))
# 0

##############################################################
# STOP #
##############################################################
```

---

readCdfQc

**Reads the QC units of CDF file**

### Description

Reads the QC units of CDF file.

### Usage

```r
readCdfQc(filename, units = NULL, verbose = 0)
```

### Arguments

- **filename**: name of the CDF file.
- **units**: The QC unit indices as a vector of integers. NULL indicates that all units should be read.
- **verbose**: how verbose should the output be. 0 means no output, with higher numbers being more verbose.

### Value

A list with one component for each QC unit.
readCdfUnitNames

Author(s)
Kasper Daniel Hansen

See Also
readCdf().

readCdfUnitNames  
Reads (probeset) names from an Affymetrix CDF file

Description
Gets the names of all or a subset of units (probesets) in an Affymetrix CDF file. This can be used to get a map between unit names and the internal unit indices used by the CDF file.

Usage
readCdfUnitNames(filename, units=NULL, verbose=0)

Arguments
filename  
The filename of the CDF file.

units  
An integer vector of unit indices specifying which units to be read. If NULL, all units are read.

verbose  
An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.

Value
A character vector of unit names.

Author(s)
Henrik Bengtsson (http://www.braju.com/R/)

See Also
readCdfUnits().

Examples
## Not run: See help(readCdfUnits) for an example
Description

Reads units (probesets) from an Affymetrix CDF file. Gets all or a subset of units (probesets).

Usage

readCdfUnits(filename, units=NULL, readXY=TRUE, readBases=TRUE, readExpos=TRUE, readType=TRUE, readDirection=TRUE, stratifyBy=c("nothing", "pmmm", "pm", "mm"), readIndices=FALSE, verbose=0)

Arguments

filename  The filename of the CDF file.
units     An integer vector of unit indices specifying which units to be read. If NULL, all units are read.
readXY    If TRUE, cell row and column (x,y) coordinates are retrieved, otherwise not.
readBases If TRUE, cell P and T bases are retrieved, otherwise not.
readExpos If TRUE, cell "expos" values are retrieved, otherwise not.
readType  If TRUE, unit types are retrieved, otherwise not.
readDirection If TRUE, unit and group directions are retrieved, otherwise not.
stratifyBy A character string specifying which and how elements in group fields are returned. If "nothing", elements are returned as is, i.e. as vectors. If "pm"/"mm", only elements corresponding to perfect-match (PM) / mismatch (MM) probes are returned (as vectors). If "pmmm", elements are returned as a matrix where the first row holds elements corresponding to PM probes and the second corresponding to MM probes. Note that in this case, it is assumed that there are equal number of PMs and MMs; if not, an error is generated. Moreover, the PMs and MMs may not even be paired, i.e. there is no guarantee that the two elements in a column corresponds to a PM-MM pair.
readIndices If TRUE, cell indices calculated from the row and column (x,y) coordinates are retrieved, otherwise not. Note that these indices are one-based.
verbose   An integer specifying the verbose level. If 0, the file is parsed quietly. The higher numbers, the more details.

Value

A named list where the names corresponds to the names of the units read. Each element of the list is in turn a list structure with three components:
groups A list with one component for each group (also called block). The information on each group is a list of up to seven components: x, y, pbase, tbase, expos, indices, and direction. All fields but the latter have the same number of values as there are cells in the group. The latter field has only one value indicating the direction for the whole group.

type An integer specifying the type of the unit, where 1 is "expression", 2 is "genotyping", 3 is "CustomSeq", and 4 "tag".

direction An integer specifying the direction of the unit, which defines if the probes are interrogating the sense or the anti-sense target, where 0 is "no direction", 1 is "sense", and 2 is "anti-sense".

Cell indices are one-based

Note that in affxparser all cell indices are by convention one-based, which is more convenient to work with in R. For more details on one-based indices, see 2. Cell coordinates and cell indices.

Author(s)

James Bullard and Kasper Daniel Hansen. Modified by Henrik Bengtsson to read any subset of units and/or subset of parameters, to stratify by PM/MM, and to return cell indices.

References


See Also

readCdfCellIndices().

Examples

# Find any CDF file
cdfFile <- findCdf()

# Read all units in a CDF file [-20s => 0.34ms/unit]
cdf0 <- readCdfUnits(cdfFile, readXY=FALSE, readExpos=FALSE)

# Read a subset of units in a CDF file [-6ms => 0.06ms/unit]
units1 <- c(5, 100:109, 34)
cdf1 <- readCdfUnits(cdfFile, units=units1, readXY=FALSE, readExpos=FALSE)
stopifnot(identical(cdf1, cdf0[units1]))
rm(cdf0)

# Create a unit name to index map
names <- readCdfUnitNames(cdfFile)
units2 <- match(names(cdf1), names)
stopifnot(all.equal(units1, units2))
cdf2 <- readCdfUnits(cdfFile, units=units2, readXY=FALSE, readExpos=FALSE)

stopifnot(identical(cdf1, cdf2))

##############################################################
} # STOP #
##############################################################

---

**readCdfUnitsWriteMap**

*Generates an Affymetrix cell-index write map from a CDF file*

**Description**

Generates an Affymetrix cell-index write map from a CDF file.

The purpose of this method is to provide a re-ordering of cell elements such that cells in units (probesets) can be stored in contiguous blocks. When reading cell elements unit by unit, minimal file re-position is required resulting in a faster reading.

Note: At the moment does this package not provide methods to write/reorder CEL files. In the meanwhile, you have to write and re-read using your own file format. That's not too hard using `writeBin()` and `readBin()`.

**Usage**

`readCdfUnitsWriteMap(filename, units=NULL, ..., verbose=FALSE)`

**Arguments**

- `filename` The pathname of the CDF file.
- `units` An integer vector of unit indices specifying which units to listed first. All other units are added in order at the end. If `NULL`, units are in order.
- `...` Additional arguments passed to `readCdfUnits()`.
- `verbose` Either a logical, a numeric, or a Verbose object specifying how much verbose/debug information is written to standard output. If a Verbose object, how detailed the information is is specified by the threshold level of the object. If a numeric, the value is used to set the threshold of a new Verbose object. If `TRUE`, the threshold is set to -1 (minimal). If `FALSE`, no output is written (and neither is the `R.utils` package required).

**Value**

A integer vector which is a write map.
Author(s)
Henrik Bengtsson

See Also
To invert maps, see invertMap(). readCel() and readCelUnits().

Examples

```r
if (require("AffymetrixDataTestFiles")) {
  # START #
  #############################################################################

  # Find any CDF file
cdfFile <- findCdf()

  # Create a cell-index map (for writing)
  writeMap <- readCdfUnitsWriteMap(cdfFile)

  # Inverse map to be used to read cell elements such that, when read
  # read unit by unit, they are read much faster.
  readMap <- invertMap(writeMap)

  # Validate the two maps
  stopifnot(identical(readMap[writeMap], 1:length(readMap)))

  cat("Summary of the \"randomness\" of the cell indices:\n")
  moves <- diff(readMap) - 1
  cat(sprintf("Number of unnecessary file re-positioning: %d (%.1f%)\n",
                sum(moves != 0), 100*sum(moves != 0)/length(moves)))
  cat(sprintf("Extra positioning: %.1fGb\n", sum(abs(moves))/1024^3))

  smallMoves <- moves[abs(moves) <= 25];
  largeMoves <- moves[abs(moves) > 25];
  layout(matrix(1:2))
  main <- "Non-signed file moves required in unordered file"
  hist(smallMoves, nclass=51, main=main, xlab="moves <=25 bytes")
  hist(largeMoves, nclass=101, main="", xlab="moves >25 bytes")

  # Clean up
  layout(1)
  rm(cdfFile, readMap, writeMap, moves, smallMoves, largeMoves, main)

  #############################################################################

  }  # STOP #
  #############################################################################
```
```
# Function to read Affymetrix probeset annotations
readAffymetrixProbesetAnnotation <- function(pathname, ...) {
  # Get headers
  header <- scan(pathname, what="character", sep="comma", quote=""",
                 quiet=TRUE, nlines=1);

  # Read only a subset of columns (unique to this example)
  cols <- c("Probe Set ID"="probeSet",
             "Chromosome"="chromosome",
             "Physical Position"="physicalPosition",
             "dbSNP RS ID"="dbSnpId");

  colClasses <- rep("NULL", length(header));
  colClasses[header %in% names(cols)] <- "character";

  # Read the data (this is what takes time)
  df <- read.table(pathname, colClasses=colClasses, header=TRUE, sep="comma",
                   quote=""", na.strings="---", strip.white=TRUE, check.names=FALSE,
                   blank.lines.skip=FALSE, fill=FALSE, comment.char="", ...);

  # Re-order columns
  df <- df[,match(names(cols), colnames(df))];
  colnames(df) <- cols;

  # Use "Probe Set ID" as rownames. Note that if we use 'row.names=1'
  # or similar something goes wrong. /HB 2006-03-06
  rownames(df) <- df[[1]];
  df <- df[-1];

  # Change types of columns
  df[[1]] <- factor(df[[1]], levels=c(1:22,"X","Y",NA), ordered=TRUE);
  df[[2]] <- as.integer(df[[2]]);

  df;
} # readAffymetrixProbesetAnnotation()

# Main
for (zz in 1) {
  # Chip to be remapped
  chipType <- "Mapping50K_Xba240"
  annoFile <- paste(chipType, ".annot.csv", sep=""
  cdfFile <- findCdf(chipType)
  if (is.null(cdfFile) || !file.exists(annoFile))
    break;
# Read SNP location details
snpInfo <- readAffymetrixProbesetAnnotation(annoFile)

# Order by chromosome and then physical position
o <- order(snpInfo[[1]], snpInfo[[2]])
snpInfo <- snpInfo[o,]
rm(o)

# Read unit names in CDF file
unitNames <- readCdfUnitNames(cdfFile)

# The CDF unit indices sorted by chromosomal position
units <- match(rownames(snpInfo), unitNames)

# ...and cell indices in the same order
writeMap <- readCdfUnitsWriteMap(cdfFile, units=units)

# Inverse map to be used to write cell elements such that, if they
# later are read unit by unit, they are read in contiguous blocks.
readMap <- invertMap(writeMap)

# Clean up
rm(chipType, annoFile, cdfFile, snpInfo, unitNames, units, readMap, writeMap)

} # for (zz in 1)

Reads an Affymetrix CEL file

**Description**

This function reads all or a subset of the data in an Affymetrix CEL file.

**Usage**

readCel(filename, 
    indices = NULL, 
    readHeader = TRUE, 
    readXY = FALSE, readIntensities = TRUE, 
    readStdvs = FALSE, readPixels = FALSE, 
    readOutliers = TRUE, readMasked = TRUE, 
    readMap = NULL, 
    verbose = 0, 
    .checkArgs = TRUE)
Arguments

filename the name of the CEL file.
indices a vector of indices indicating which features to read. If the argument is NULL all features will be returned.
readXY a logical: will the (x,y) coordinates be returned.
readIntensities a logical: will the intensities be returned.
readStdvs a logical: will the standard deviations be returned.
readPixels a logical: will the number of pixels be returned.
readOutliers a logical: will the outliers be return.
readMasked a logical: will the masked features be returned.
readHeader a logical: will the header of the file be returned.
readMap A vector remapping cell indices to file indices. If NULL, no mapping is used.
verbose how verbose do we want to be. 0 is no verbosity, higher numbers mean more verbose output. At the moment the values 0, 1 and 2 are supported.
.checkArgs If TRUE, the arguments will be validated, otherwise not. Warning: This should only be used if the arguments have been validated elsewhere!

Value

A CEL files consists of a header, a set of cell values, and information about outliers and masked cells.

The cell values, which are values extract for each cell (aka feature or probe), are the (x,y) coordinate, intensity and standard deviation estimates, and the number of pixels in the cell. If readIndices=NULL, cell values for all cells are returned. Only cell values specified by argument readIndices are returned.

This value returns a named list with components described below:

header The header of the CEL file. Equivalent to the output from readCelHeader, see the documentation for that function.
x, y (cell values) Two integer vectors containing the x and y coordinates associated with each feature.
intensities (cell value) A numeric vector containing the intensity associated with each feature.
stdvs (cell value) A numeric vector containing the standard deviation associated with each feature.
pixels (cell value) An integer vector containing the number of pixels associated with each feature.
outliers An integer vector of indices specifying which of the queried cells that are flagged as outliers. Note that there is a difference between outliers=NULL and outliers=integer(0); the last case happens when readOutliers=TRUE but there are no outliers.
masked An integer vector of indices specifying which of the queried cells that are flagged as masked. Note that there is a difference between `masked=NULL` and `masked=integer(0)`, the last case happens when `readMasked=TRUE` but there are no masked features.

The elements of the cell values are ordered according to argument `indices`. The lengths of the cell-value elements equals the number of cells read.

Which of the above elements that are returned are controlled by the `readNnn` arguments. If `FALSE`, the corresponding element above is `NULL`, e.g. if `readStdvs=FALSE` then `stdvs` is `NULL`.

**Outliers and masked cells**

The Affymetrix image analysis software flags cells as outliers and masked. This method does not return these flags, but instead vectors of cell indices listing which cells of the queried cells are outliers and masked, respectively. The current community view seems to be that this should be done based on statistical modeling of the actual probe intensities and should be based on the choice of preprocessing algorithm. Most algorithms are only using the intensities from the CEL file.

**Memory usage**

The Fusion SDK allocates memory for the entire CEL file, when the file is accessed (but does not actually read the file into memory). Using the `indices` argument will therefore only affect the memory use of the final object (as well as speed), not the memory allocated in the C function used to parse the file. This should be a minor problem however.

**Troubleshooting**

It is considered a bug if the file contains information not accessible by this function, please report it.

**Author(s)**

James Bullard and Kasper Daniel Hansen

**See Also**

`readCelHeader()` for a description of the header output. Often a user only wants to read the intensities, look at `readCelIntensities()` for a function specialized for that use.

**Examples**

```r
for (zzz in 0) { # Only so that 'break' can be used
  # Scan current directory for CEL files
  celFiles <- list.files(pattern="[.]c(e|E)(l|L)$")
  if (length(celFiles) == 0)
    break;
  celfile <- celFiles[1]
  # Read a subset of cells
```
idxs <- c(1:5, 1250:1500, 450:440)
cel <- readCel(celFile, indices=idxs, readOutliers=TRUE)
str(cel)

# Clean up
rm(celFiles, celFile, cel)

Parsing the header of an Affymetrix CEL file

Description
Reads in the header of an Affymetrix CEL file using the Fusion SDK.

Usage
readCelHeader(filename)

Arguments
filename the name of the CEL file.

Details
This function returns the header of a CEL file. Affymetrix operates with different versions of this file format. Depending on what version is being read, different information is accessible.

Value
A named list with components described below. The entries are obtained from the Fusion SDK interface functions. We try to obtain all relevant information from the file.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>the name of the cel file.</td>
</tr>
<tr>
<td>version</td>
<td>the version of the cel file.</td>
</tr>
<tr>
<td>cols</td>
<td>the number of columns on the chip.</td>
</tr>
<tr>
<td>rows</td>
<td>the number of rows on the chip.</td>
</tr>
<tr>
<td>total</td>
<td>the total number of features on the chip. Usually equal to rows times cols, but since it is a separate attribute in the SDK we decided to include it anyway.</td>
</tr>
<tr>
<td>algorithm</td>
<td>the algorithm used to create the CEL file.</td>
</tr>
<tr>
<td>parameters</td>
<td>the parameters used in the algorithm. Seems to be semi-colon separated.</td>
</tr>
<tr>
<td>chiptype</td>
<td>the type of the chip.</td>
</tr>
<tr>
<td>header</td>
<td>the entire header of the CEL file. Only available for non-calvin format files.</td>
</tr>
<tr>
<td>datheader</td>
<td>the entire dat header of the CEL file. This contains for example a date.</td>
</tr>
</tbody>
</table>
readCelIntensities

library package the library package name of the file. Empty for older versions.
cellmargin a parameter used to generate the CEL file. According to Affymetrix, it designates the number of pixels to ignore around the feature border when calculating the intensity value (the number of pixels ignored are cellmargin divided by 2).
noutliers the number of features reported as outliers.
rmasked the number of features reported as masked.

Note

Memory usage: the Fusion SDK allocates memory for the entire CEL file, when the file is accessed. The memory footprint of this function will therefore seem to be (rather) large.
Speed: CEL files of version 2 (standard text files) needs to be completely read in order to report the number of outliers and masked features.

Author(s)

James Bullard and Kasper Daniel Hansen

See Also

readCel() for reading in the entire CEL file. That function also returns the header. See affxparserInfo for general comments on the package and the Fusion SDK.

Examples

# Scan current directory for CEL files
files <- list.files(pattern = "[.](c|C)(e|E)(l|L)$")
if (length(files) > 0) {
  header <- readCelHeader(files[1])
  print(header)
  rm(header)
}

# Clean up
rm(files)

---

readCelIntensities Reads the intensities contained in several Affymetrix CEL files

Description

Reads the intensities of several Affymetrix CEL files (as opposed to readCel() which only reads a single file).

Usage

readCelIntensities(filenames, indices = NULL, ..., verbose = 0)
readCellIntensities

Arguments

filenames the names of the CEL files as a character vector.
indices a vector of which indices should be read. If the argument is NULL all features will be returned.
... Additional arguments passed to readCel().
verbose an integer: how verbose do we want to be, higher means more verbose.

Details

The function will initially allocate a matrix with the same memory footprint as the final object.

Value

A matrix with a number of rows equal to the length of the indices argument (or the number of features on the entire chip), and a number of columns equal to the number of files. The columns are ordered according to the filenames argument.

Note

Currently this function builds on readCel(), and simply calls this function multiple times. If testing yields sufficient reasons for doing so, it may be re-implemented in C++.

Author(s)

James Bullard and Kasper Daniel Hansen

See Also

readCel() for a discussion of a more versatile function, particular with details of the indices argument.

Examples

# Scan current directory for CEL files
files <- list.files(pattern="[.](c|C)(e|E)(l|L)$")
if (length(files) >= 2) {
  cel <- readCellIntensities(files[1:2])
  str(cel)
  rm(cel)
}

# Clean up
rm(files)
readCelRectangle

**Description**

Reads a spatial subset of probe-level data from Affymetrix CEL files.

**Usage**

```r
readCelRectangle(filename, xrange=c(0, Inf), yrange=c(0, Inf), ..., asMatrix=TRUE)
```

**Arguments**

- **filename**: The pathname of the CEL file.
- **xrange**: A numeric vector of length two giving the left and right coordinates of the cells to be returned.
- **yrange**: A numeric vector of length two giving the top and bottom coordinates of the cells to be returned.
- **...**: Additional arguments passed to `readCel()`.
- **asMatrix**: If TRUE, the CEL data fields are returned as matrices with element (1,1) corresponding to cell (xrange[1],yrange[1]).

**Value**

A named list CEL structure similar to what `readCel()` returns. In addition, if `asMatrix` is TRUE, the CEL data fields are returned as matrices, otherwise not.

**Author(s)**

Henrik Bengtsson

**See Also**

The `readCel()` method is used internally.

**Examples**

```r
if (require("AffymetrixDataTestFiles")) {
  # START #
  rotate270 <- function(x, ...) {
    x <- t(x)
    nc <- ncol(x)
    if (nc < 2) return(x)
    x[,nc:1,drop=FALSE]
  }
```

# Search for some available CEL files
path <- system.file("rawData", package="AffymetrixDataTestFiles")
file <- findFiles(pattern=".*\(cel|CEL\)$", path=path, recursive=TRUE)

# Read CEL intensities in the upper left corner
cel <- readCelRectangle(file, xrange=c(0,250), yrange=c(0,250))
z <- rotate270(cel$intensities)
sub <- paste("Chip type:", cel$header$chiptype)
image(z, col=gray.colors(256), axes=FALSE, main=basename(file), sub=sub)
text(x=0, y=1, labels="(0,0)", adj=c(0,-0.7), cex=0.8, xpd=TRUE)
text(x=1, y=0, labels="(250,250)", adj=c(1,1.2), cex=0.8, xpd=TRUE)

# Clean up
rm(rotate270, files, file, cel, z, sub)

readCelUnits

---

**readCelUnits**

Reads probe-level data ordered as units (probesets) from one or several Affymetrix CEL files

### Description

Reads probe-level data ordered as units (probesets) from one or several Affymetrix CEL files by using the unit and group definitions in the corresponding Affymetrix CDF file.

### Usage

```r
readCelUnits(filenames, units=NULL, stratifyBy=c("nothing", "pmmm", "pm", "mm"),
cdf=NULL, ..., addDimnames=FALSE, dropArrayDim=TRUE, transforms=NULL, readMap=NULL,
verbose=FALSE)
```

### Arguments

- **filenames**
  - The filenames of the CEL files.
- **units**
  - An integer vector of unit indices specifying which units to be read. If NULL, all units are read.
- **stratifyBy**
  - Argument passed to low-level method readCdfCellIndices.
- **cdf**
  - A character filename of a CDF file, or a CDF list structure. If NULL, the CDF file is searched for by findCdf() first starting from the current directory and then from the directory where the first CEL file is.
Arguments passed to low-level method `readCel`, e.g. `readXY` and `readStdvs`.

- **addDimnames**: If `TRUE`, dimension names are added to arrays, otherwise not. The size of the returned CEL structure in bytes increases by 30-40% with dimension names.

- **dropArrayDim**: If `TRUE` and only one array is read, the elements of the group field do **not** have an array dimension.

- **transforms**: A list of exactly length(filenames) functions. If `NULL`, no transformation is performed. Intensities read are passed through the corresponding transform function before being returned.

- **readMap**: A vector remapping cell indices to file indices. If `NULL`, no mapping is used.

- **verbose**: Either a logical, a numeric, or a `Verbose` object specifying how much verbose/debug information is written to standard output. If a `Verbose` object, how detailed the information is is specified by the threshold level of the object. If a numeric, the value is used to set the threshold of a new `Verbose` object. If `TRUE`, the threshold is set to -1 (minimal). If `FALSE`, no output is written (and neither is the `R.utils` package required).

### Value

A named list with one element for each unit read. The names corresponds to the names of the units read. Each unit element is in turn a list structure with groups (aka blocks). Each group contains requested fields, e.g. intensities, stdvs, and pixels. If more than one CEL file is read, an extra dimension is added to each of the fields corresponding, which can be used to subset by CEL file.

Note that neither CEL headers nor information about outliers and masked cells are returned. To access these, use `readCelHeader()` and `readCel()`.

### Author(s)

Henrik Bengtsson

### References


### See Also

Internally, `readCelHeader()`, `readCdfUnits()` and `readCel()` are used.

### Examples

```r
# Search for some available CEL files
path <- system.file("rawData", package="AffymetrixDataTestFiles")
files <- findFiles(pattern="[.]cel[CE]L$", path=path, recursive=TRUE, firstOnly=FALSE)
```
files <- grep("FusionSDK_Test3", files, value=TRUE)
files <- grep("Calvin", files, value=TRUE)

# Fake more CEL files if not enough
files <- rep(files, length.out=5)
print(files);
rm(files);

#############################################################
} # STOP #
#############################################################

readChp

A function to read Affymetrix CHP files

Description
This function will parse any type of CHP file and return the results in a list. The contents of
the list will depend on the type of CHP file that is parsed and readers are referred to Affymetrix
documentation of what should be there, and how to interpret it.

Usage
readChp(filename, withQuant = TRUE)

Arguments
filename The name of the CHP file to read.
withQuant A boolean value, currently largely unused.

Details
This is an interface to the Affymetrix Fusion SDK. The Affymetrix documentation should be con-
sulted for explicit details.

Value
A list is returned. The contents of the list depend on the type of CHP file that was read. Users may
want to translate the different outputs into specific containers.

Troubleshooting
It is considered a bug if the file contains information not accessible by this function, please report
it.

Author(s)
R. Gentleman
**readClf**

*Parsing a CLF file using Affymetrix Fusion SDK*

**Description**

This function parses a CLF file using the Affymetrix Fusion SDK. CLF (chip layout) files contain information associating probe ids with chip x- and y-coordinates.

**Usage**

```r
readClf(file)
```

**Arguments**

- `file` character(1) providing a path to the CLF file to be input.

**Value**

An list. The header element is always present.

- `header` A list with information about the CLF file. The list contains elements described in the CLF file format document referenced below.
- `dims` A length-two integer vector of chip x- and y-coordinates.
- `id` An integer vector of length `prod(dims)` containing probe identifiers.
- `x` An integer vector of length `prod(dims)` containing x-coordinates corresponding to the entries in `id`.
- `y` An integer vector of length `prod(dims)` containing y-coordinates corresponding to the entries in `id`.

**Author(s)**

Martin Morgan
See Also


---

**readClfEnv**

**Parsing a CLF file using Affymetrix Fusion SDK**

**Description**

This function parses a CLF file using the Affymetrix Fusion SDK. CLF (chip layout) files contain information associating probe ids with chip x- and y-coordinates.

**Usage**

```r
readClfEnv(file, readBody = TRUE)
```

**Arguments**

- `file` character(1) providing a path to the CLF file to be input.
- `readBody` logical(1) indicating whether the entire file should be parsed (TRUE) or only the file header information describing the chips to which the file is relevant.

**Value**

An environment. The header element is always present; the remainder are present when readBody=TRUE.

- `header` A list with information about the CLF file. The list contains elements described in the CLF file format document referenced below.
- `dims` A length-two integer vector of chip x- and y-coordinates.
- `id` An integer vector of length prod(dims) containing probe identifiers.
- `x` An integer vector of length prod(dims) containing x-coordinates corresponding to the entries in `id`.
- `y` An integer vector of length prod(dims) containing y-coordinates corresponding to the entries in `id`.

**Author(s)**

Martin Morgan

**See Also**

readClfHeader  

**Read the header of a CLF file.**

**Description**

Reads the header of a CLF file. The exact information stored in this file can be viewed in the `readClfEnv()` documentation which reads the header in addition to the body.

**Usage**

`readClfHeader(file)`

**Arguments**

- **file** file a CLF file

**Value**

A list of header elements.

readPgf  

**Parsing a PGF file using Affymetrix Fusion SDK**

**Description**

This function parses a PGF file using the Affymetrix Fusion SDK. PGF (probe group) files describe probes present within probe sets, including the type (e.g., pm, mm) of the probe and probeset.

**Usage**

`readPgf(file, indices = NULL)`

**Arguments**

- **file** character(1) providing a path to the PGF file to be input.
- **indices** integer(n) a vector of indices of the probesets to be read.

**Value**

An list. The header element is always present; the remainder are present when `readBody=TRUE`. The elements present when `readBody=TRUE` describe probe sets, atoms, and probes. Elements within probe sets, for instance, are coordinated such that the ith index of one vector (e.g., `probesetId`) corresponds to the ith index of a second vector (e.g., `probesetType`). The atoms contained within probeset i are in positions `probesetStartAtom[i]:(probesetStartAtom[i+1]-1)` of the atom vectors. A similar map applies to probes within atoms, using `atomStartProbe` as the index.

The PGF file format includes optional elements; these elements are always present in the list, but with appropriate default values.
header  
A list with information about the PGF file. The list contains elements described in the PGF file format document referenced below.

probesetId  
integer vector of probeset identifiers.

probesetType  
character vector of probeset types. Types are described in the PGF file format document.

probesetName  
character vector of probeset names.

probesetStartAtom  
integer vector of the start index (e.g., in the element atomId of atoms belonging to this probeset).

atomId  
integer vector of atom identifiers.

atomExonPosition  
integer vector of probe interrogation position relative to the target sequence.

atomStartProbe  
integer vector of the start index (e.g., in the element probeId of probes belonging to this atom).

probeId  
integer vector of probe identifiers.

probeType  
character vector of probe types. Types are described in the PGF file format document.

probeGcCount  
integer vector of probe GC content.

probeLength  
integer vector of probe lengths.

probeInterrogationPosition  
integer vector of the position, within the probe, at which interrogation occurs.

probeSequence  
character vector of the probe sequence.

Author(s)

Martin Morgan

See Also


The internal function .pgfProbeIndexFromProbesetIndex provides a map between the indices of probe set entries and the indices of the probes contained in the probe set.

---

readPgfEnv  
Parsing a PGF file using Affymetrix Fusion SDK

Description

This function parses a PGF file using the Affymetrix Fusion SDK. PGF (probe group) files describe probes present within probe sets, including the type (e.g., pm, mm) of the probe and probeset.
**readPgfEnv**

**Usage**

`readPgfEnv(file, readBody = TRUE, indices = NULL)`

**Arguments**

- **file**: character(1) providing a path to the PGF file to be input.
- **readBody**: logical(1) indicating whether the entire file should be parsed (TRUE) or only the file header information describing the chips to which the file is relevant.
- **indices**: integer(n) vector of positive integers indicating which probesets to read. These integers must be sorted (increasing) and unique.

**Value**

An environment. The header element is always present; the remainder are present when `readBody=TRUE`. The elements present when `readBody=TRUE` describe probe sets, atoms, and probes. Elements within probe sets, for instance, are coordinated such that the ith index of one vector (e.g., `probesetId`) corresponds to the ith index of a second vector (e.g., `probesetType`). The atoms contained within probeset i are in positions `probesetStartAtom[i]:(probesetStartAtom[i+1]-1)` of the atom vectors. A similar map applies to probes within atoms, using `atomStartProbe` as the index.

The PGF file format includes optional elements; these elements are always present in the environment, but with appropriate default values.

- **header**: A list with information about the PGF file. The list contains elements described in the PGF file format document referenced below.
- **probesetId**: integer vector of probeset identifiers.
- **probesetType**: character vector of probeset types. Types are described in the PGF file format document.
- **probesetName**: character vector of probeset names.
- **probesetStartAtom**: integer vector of the start index (e.g., in the element `atomId` of atoms belonging to this probeset).
- **atomId**: integer vector of atom identifiers.
- **atomExonPosition**: integer vector of probe interrogation position relative to the target sequence.
- **atomStartProbe**: integer vector of the start index (e.g., in the element `probeId` of probes belonging to this atom).
- **probeId**: integer vector of probe identifiers.
- **probeType**: character vector of probe types. Types are described in the PGF file format document.
- **probeGcCount**: integer vector of probe GC content.
- **probeLength**: integer vector of probe lengths.
- **probeInterrogationPosition**: integer vector of the position, within the probe, at which interrogation occurs.
- **probeSequence**: character vector of the probe sequence.
Author(s)

Martin Morgan

See Also


The internal function .pgfProbeIndexFromProbesetIndex provides a map between the indices of probe set entries and the indices of the probes contained in the probe set.

```
readPgfHeader

readPgfHeader(file)  # Read the header of a PGF file into a list.
```

Description

This function reads the header of a PGF file into a list more details on what the exact fields are can be found in the details section.

Usage

```
readPgfHeader(file)
```

Arguments

```
file  # file:A file in PGF format
```

Details


Value

A list corresponding to the elements in the header.
updateCel

Description

Updates a CEL file.

Usage

updateCel(filename, indices=NULL, intensities=NULL, stdvs=NULL, pixels=NULL, writeMap=NULL, ..., verbose=0)

Arguments

filename  The filename of the CEL file.
indices   A numeric vector of cell (probe) indices specifying which cells to updated. If NULL, all indices are considered.
intensities   A numeric vector of intensity values to be stored. Alternatively, it can also be a named data.frame or matrix (or list) where the named columns (elements) are the fields to be updated.
stdvs   A optional numeric vector.
pixels   A optional numeric vector.
writeMap   An optional write map.
...   Not used.
verbose   An integer specifying how much verbose details are outputted.

Details

Currently only binary (v4) CEL files are supported. The current version of the method does not make use of the Fusion SDK, but its own code to navigate and update the CEL file.

Value

Returns (invisibly) the pathname of the file updated.

Author(s)

Henrik Bengtsson
Examples

if (require("AffymetrixDataTestFiles")) {
  # Search for some available Calvin CEL files
  path <- system.file("rawData", package="AffymetrixDataTestFiles")
  files <- findFiles(pattern="\.[.]\{cel|CEL\}$", path=path, recursive=TRUE, firstOnly=FALSE)
  files <- grep("FusionSDK_HG-U133A", files, value=TRUE)
  files <- grep("Calvin", files, value=TRUE)
  file <- files[1]

  # Convert to an XDA CEL file
  filename <- file.path(tempdir(), basename(file))
  if (file.exists(filename))
    file.remove(filename)
  convertCel(file, filename)

  fields <- c("intensities", "stdvs", "pixels")

  # Cells to be updated
  idxs <- 1:2

  # Get CEL header
  hdr <- readCelHeader(filename)

  # Get the original data
  cel <- readCel(filename, indices=idxs, readStdvs=TRUE, readPixels=TRUE)
  print(cel[fields])
  cel0 <- cel

  # Square-root the intensities
  updateCel(filename, indices=idxs, intensities=sqrt(cel$intensities))
  cel <- readCel(filename, indices=idxs, readStdvs=TRUE, readPixels=TRUE)
  print(cel[fields])

  data <- data.frame(
    intensities=cel0$intensities,
    stdvs=c(201.1, 3086.1)+0.5,
    pixels=c(9,9+1)
  )
  updateCel(filename, indices=idxs, data)

  # Assert correctness of update
  cel <- readCel(filename, indices=idxs, readStdvs=TRUE, readPixels=TRUE)
print(cel[fields])
for (ff in fields) {
  stopifnot(all.equal(cel[[ff]], data[[ff]], .Machine$double.eps^0.25))
}

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Update a region of the CEL file
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# Load pre-defined data
side <- 306
pathname <- system.file("extras/easternEgg.gz", package="affxparser")
con <- gzfile(pathname, open="rb")
z <- readBin(con=con, what="integer", size=1, signed=FALSE, n=side^2)
close(con)
z <- matrix(z, nrow=side, ncol=side)
side <- min(hdr$cols - 2*22, side)
z <- as.double(z[1:side,1:side])
x <- matrix(22+0:(side-1), nrow=side, ncol=side, byrow=TRUE)
idxs <- as.vector((1 + x) + hdr$cols*t(x))
# Load current data in the same region
z0 <- readCel(filename, indices=idxs)$intensities
# Mix the two data sets
z <- (0.3*z^2 + 0.7*z0)
# Update the CEL file
updateCel(filename, indices=idxs, intensities=z)

# Make some spatial changes
rotate270 <- function(x, ...) {
  x <- t(x)
  nc <- ncol(x)
  if (nc < 2) return(x)
x[,nc:1,drop=FALSE]
}
# Display a spatial image of the updated CEL file
cel <- readCelRectangle(filename, xrange=c(0,350), yrange=c(0,350))
z <- rotate270(cel$intensities)
sub <- paste("Chip type:", cel$header$chiptype)
image(z, col=gray.colors(256), axes=FALSE, main=basename(filename), sub=sub)
  text(x=0, y=1, labels="(0,0)", adj=c(0,-0.7), cex=0.8, xpd=TRUE)
  text(x=1, y=0, labels="(350,350)", adj=c(1,1.2), cex=0.8, xpd=TRUE)

# Clean up
file.remove(filename)
rm(files, cel, cel0, idxs, data, ff, fields, rotate270)

# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# STOP #
# - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
updateCelUnits  

**Description**

Updates a CEL file unit by unit.

*Please note that, contrary to readCelUnits(), this method can only update a single CEL file at the time.*

**Usage**

updateCelUnits(filename, cdf=NULL, data, ..., verbose=0)

**Arguments**

- **filename**
  The filename of the CEL file.

- **cdf**
  A (optional) CDF list structure either with field indices or fields x and y. If NULL, the unit names (and from there the cell indices) are inferred from the names of the elements in data.

- **data**
  A list structure in a format similar to what is returned by readCelUnits() for a single CEL file only.

- **...**
  Optional arguments passed to readCdfCellIndices(), which is called if cdf is not given.

- **verbose**
  An integer specifying how much verbose details are outputted.

**Value**

Returns what updateCel() returns.

**Working with re-arranged CDF structures**

Note that if the cdf structure is specified the CDF file is not queried, but all information about cell x and y locations, that is, cell indices is expected to be in this structure. This can be very useful when one work with a cdf structure that originates from the underlying CDF file, but has been restructured for instance through the applyCdfGroups() method, and data correspondingly. This update method knows how to update such structures too.

**Author(s)**

Henrik Bengtsson

**See Also**

Internally, updateCel() is used.
Examples

```
if (require("AffymetrixDataTestFiles")) {
  # Search for some available Calvin CEL files
  path <- system.file("rawData", package="AffymetrixDataTestFiles")
  files <- findFiles(pattern="[.]\{cell\}CEL\$", path=path, recursive=TRUE, firstOnly=FALSE)
  files <- grep("FusionSDK_Test3", files, value=TRUE)
  files <- grep("Calvin", files, value=TRUE)
  file <- files[1]

  # Convert to an XDA CEL file
  pathname <- file.path(tempdir(), basename(file))
  if (file.exists(pathname))
    file.remove(pathname)
  convertCel(file, pathname)

  # Check for the CDF file
  hdr <- readCelHeader(pathname)
  cdfFile <- findCdf(hdr$chiptype)
  hdr <- readCdfHeader(cdfFile)
  nbrOfUnits <- hdr$nunits
  print(nbrOfUnits);

  # Example: Read and re-write the same data
  units <- c(101, 51)
  data1 <- readCelUnits(pathname, units=units, readStdvs=TRUE)
  cat("Original data:
")
  str(data1)
  updateCelUnits(pathname, data=data1)
  data2 <- readCelUnits(pathname, units=units, readStdvs=TRUE)
  cat("Updated data:
")
  str(data2)
  stopifnot(identical(data1, data2))

  # Example: Random read and re-write "stress test"
  for (kk in 1:10) {
    nunits <- sample(min(1000,nbrOfUnits), size=1)
    units <- sample(nbrOfUnits, size=nunits)
    cat(sprintf("%02d. Selected %d random units: reading", kk, nunits));
    t <- system.time{
      data1 <- readCelUnits(pathname, units=units, readStdvs=TRUE)
    }
```
writeCdf

Creates a binary CDF file

Description

This function creates a binary CDF file given a valid CDF structure containing all necessary elements.

Warning: The API for this function is likely to be changed in future versions.

Usage

writeCdf(fname, cdfheader, cdf, cdfqc, overwrite=FALSE, verbose=0)

Arguments

fname name of the CDF file.
cdfheader A list with a structure equal to the output of readCdfHeader.
cdf A list with a structure equal to the output of readCdf.
cdfqc A list with a structure equal to the output of readCdfQc.
overwrite Overwrite existing file?
verbose how verbose should the output be. 0 means no output, with higher numbers being more verbose.

Details

This function has been validated mainly by reading in various ASCII or binary CDF files which are written back as new CDF files, and compared element by element with the original files.

Value

This function is used for its byproduct: creating a CDF file.
writeCdfHeader

Author(s)
Kasper Daniel Hansen

See Also
To read the CDF “regular” and QC units with all necessary fields and values for writing a CDF file, see readCdf, readCdfQc() and readCdfHeader. To compare two CDF files, see compareCdfs.

writeCdfHeader  Writes a CDF header

Description
 Writes a CDF header. This method is not intended to be used explicitly. To write a CDF, use writeCdf() instead.

Usage
writeCdfHeader(con, cdfHeader, unitNames, qcUnitLengths, unitLengths, verbose=0)

Arguments

- **con**: An open connection to which nothing has been written.
- **cdfHeader**: A CDF header list structure.
- **unitNames**: A character vector of all unit names.
- **qcUnitLengths**: An integer vector of all the number of bytes in each of the QC units.
- **unitLengths**: An integer vector of all the number of bytes in each of the (ordinary) units.
- **verbose**: An integer specifying how much verbose details are outputted.

Value
 Returns nothing.

Author(s)
Henrik Bengtsson

See Also
This method is called by writeCdf(). See also writeCdfQcUnits() and writeCdfUnits().
writeCdfQcUnits  

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes CDF QC units. <em>This method is not intended to be used explicitly. To write a CDF, use <code>writeCdf()</code> instead.</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>writeCdfQcUnits(con, cdfQcUnits, verbose=0)</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>con</code></td>
</tr>
<tr>
<td><code>cdfQcUnits</code></td>
</tr>
<tr>
<td><code>verbose</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns nothing.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Henrik Bengtsson</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>See Also</th>
</tr>
</thead>
<tbody>
<tr>
<td>This method is called by <code>writeCdf()</code>. See also <code>writeCdfHeader()</code> and <code>writeCdfUnits()</code>.</td>
</tr>
</tbody>
</table>

writeCdfUnits  

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writes CDF units. <em>This method is not intended to be used explicitly. To write a CDF, use <code>writeCdf()</code> instead.</em></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>writeCdfUnits(con, cdfUnits, verbose=0)</code></td>
</tr>
</tbody>
</table>
writeCelHeader

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>con</td>
<td>An open connection to which a CDF header and QC units already have been written by <code>writeCdfHeader()</code> and <code>writeCdfQcUnits()</code>, respectively.</td>
</tr>
<tr>
<td>cdfUnits</td>
<td>A list structure of CDF units as returned by <code>readCdf()</code> (<em>not</em> <code>readCdfUnits()</code>).</td>
</tr>
<tr>
<td>verbose</td>
<td>An integer specifying how much verbose details are outputted.</td>
</tr>
</tbody>
</table>

Value

Returns nothing.

Author(s)

Henrik Bengtsson

See Also

This method is called by `writeCdf()`. See also `writeCdfHeader()` and `writeCdfQcUnits()`.

---

writeCelHeader | Writes a CEL header to a connection

Description

Writes a CEL header to a connection.

Usage

`writeCelHeader(con, header, outputVersion=c("4"), ...)`

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>con</td>
<td>A connection.</td>
</tr>
<tr>
<td>header</td>
<td>A list structure describing the CEL header, similar to the structure returned by <code>readCelHeader()</code>.</td>
</tr>
<tr>
<td>outputVersion</td>
<td>A character string specifying the output format. Currently only CEL version 4 (binary:XDA) are supported.</td>
</tr>
<tr>
<td>...</td>
<td>Not used.</td>
</tr>
</tbody>
</table>

Details

Currently only CEL version 4 (binary:XDA) headers can be written.

Value

Returns (invisibly) the pathname of the file created.
Redundant fields

The CEL v4 header contains redundant information. To avoid inconsistency this method generates such redundant values from the original values. This is consistent to how the CEL reader in Fusion SDK does it, cf. `readCelHeader()`. The redundant information is in the (CEL v3) header field, which contains the CEL header information as it would appear in the CEL v3 format. This in turn contains a DAT header field reproducing the DAT header from the image analysis. It is from this DAT header that the chip type is extracted.

Author(s)

Henrik Bengtsson

---

**writeTpmap**

*Writes BMAP and TPMAP files.*

**Description**

Writes BMAP and TPMAP files.

**Usage**

```
writeTpmap(filename, bpmaplist, verbose = 0)

tmap2bpmap(tpmapname, bpmapname, verbose = 0)
```

**Arguments**

- **filename** The filename.
- **bpmaplist** A list structure similar to the result of `readBpmap`.
- **tpmapname** Filename of the TPMAP file.
- **bpmapname** Filename of the BMAP file.
- **verbose** How verbose do we want to be.

**Details**

`writeTpmap` writes a text probe map file, while `tmap2bpmap` converts such a file to a binary probe mapping file. Somehow Affymetrix has different names for the same structure, depending on whether the file is binary or text. I have seen many TPMAP files referred to as BMAP files.

**Value**

These functions are called for their side effects (creating files).

**Author(s)**

Kasper Daniel Hansen
writeTpmap

See Also

readBpmap
Index

* IO
  compareCdfs, 26
  compareCels, 27
  convertCdf, 28
  convertCel, 29
  copyCel, 31
  createCel, 32
  findCdf, 33
  findFiles, 35
  invertMap, 36
  isCelFile, 37
  parseDatHeaderString, 38
  readBpmap, 39
  readCcg, 40
  readCcgHeader, 42
  readCdf, 43
  readCdfCellIndices, 45
  readCdfDataFrame, 46
  readCdfGroupNames, 47
  readCdfHeader, 48
  readCdfIsPm, 49
  readCdfNbrOfCellsPerUnitGroup, 50
  readCdfQc, 52
  readCdfUnits, 54
  readCdfUnitsWriteMap, 56
  readCel, 59
  readCelHeader, 62
  readCelIntensities, 63
  readCelRectangle, 65
  readCelUnits, 66
  readChp, 68
  readClf, 69
  readClfEnv, 70
  readClfHeader, 71
  readPgf, 71
  readPgfEnv, 72
  readPgfHeader, 74
  updateCel, 75

  updateCelUnits, 78
  writeCdf, 80
  writeCdfHeader, 81
  writeCdfQcUnits, 82
  writeCdfUnits, 82
  writeCelHeader, 83
  writeTpm, 84

* documentation
  1. Dictionary, 5
  2. Cell coordinates and cell indices, 6
  9. Advanced - Cell-index maps for reading and writing, 8

* file
  compareCdfs, 26
  compareCels, 27
  convertCdf, 28
  convertCel, 29
  copyCel, 31
  createCel, 32
  findCdf, 33
  findFiles, 35
  invertMap, 36
  isCelFile, 37
  parseDatHeaderString, 38
  readBpmap, 39
  readCcg, 40
  readCcgHeader, 42
  readCdf, 43
  readCdfCellIndices, 45
  readCdfDataFrame, 46
  readCdfGroupNames, 47
  readCdfHeader, 48
  readCdfIsPm, 49
  readCdfNbrOfCellsPerUnitGroup, 50
  readCdfQc, 52
  readCdfUnits, 53
  readCdfUnitsWriteMap, 56
readCel, 59
readCelHeader, 62
readCelIntensities, 63
readCelRectangle, 65
readCelUnits, 66
readChp, 68
readClf, 69
readClfEnv, 70
readPgf, 71
readPgfEnv, 72
readPgfHeader, 74
updateCel, 75
updateCelUnits, 78
writeCdf, 80
writeCdfHeader, 81
writeCdfQcUnits, 82
writeCelUnits, 82
writeCelHeader, 83

* package
  affxparser-package, 3
  applyCdfGroupFields, 10
  applyCdfGroups, 11
  arrangeCelFilesByChipType, 14
cdfAddBaseMmCounts, 15
cdfAddPlasqTypes, 16
cdfAddProbeOffsets, 17
cdfGetFields, 18
cdfGetGroups, 19
cdfGtypeCelToPQ, 19
cdfHeaderToCelHeader, 20
cdfMergeAlleles, 21
cdfMergeStrands, 22
cdfMergeToQuartets, 23
cdfOrderBy, 24
cdfOrderColumnsBy, 24
cdfSetDimension, 25
copyCel, 31
parseDatHeaderString, 38

1. Dictionary, 5
2. Cell coordinates and cell indices, 6
9. Advanced - Cell-index maps for reading and writing, 8

affxparser (affxparser-package), 3
affxparser-package, 3
applyCdfBlocks (applyCdfGroups), 11
applyCdfGroupFields, 10, 12, 15, 16, 17
applyCdfGroups, 3, 11, 12, 15–25
arrangeCelFilesByChipType, 14
cdfAddBaseMmCounts, 12, 15
cdfAddPlasqTypes, 16
cdfAddProbeOffsets, 12, 15, 16, 17
cdfGetFields, 12, 18
cdfGetGroups, 12, 19
cdfGtypeCelToPQ, 12, 19
cdfHeaderToCelHeader, 20
cdfMergeAlleles, 12, 21
cdfMergeStrands, 12, 22
cdfMergeToQuartets, 23
cdfOrderBy, 12, 24, 25
cdfOrderColumnsBy, 12, 24, 24
cdfSetDimension, 25
character, 14, 18, 19, 21, 34, 35, 38, 44–46, 48, 53, 54, 66, 81, 83
compareCdfs, 26, 29, 81
compareCels, 27
connection, 81–83
convertCdf, 26, 28
convertCel, 27, 29
copyCel, 31
createCel, 30, 32
data.frame, 46, 47, 75
FALSE, 26–28, 30–32, 35, 38, 56, 67
filePath, 36
findCdf, 3, 4, 33, 66
findFiles, 34, 35
function, 11, 15–19, 21–25, 67
invertMap, 9, 36, 57
isCelFile, 31, 37
list, 11, 15–25, 32, 38, 40, 42, 45, 47, 48, 50, 54, 55, 65–67, 75, 78, 81–83
list.files, 35
logical, 48, 50, 56, 67
matrix, 21, 75
NA, 18
NULL, 34, 38, 43, 45, 46, 48, 50, 53, 54, 56, 60, 66, 67, 75, 78
numeric, 56, 63, 67, 75
order, 24, 25, 36
parseDatHeaderString, 38
readBin, 56
readBpmap, 39, 85
readBpmapHeader (readBpmap), 39
readBpmapSeqinfo (readBpmap), 39
readCcg, 40, 42
readCcgHeader, 41, 42
readCdf, 3, 43, 53, 81–83
readCdfCellIndices, 3, 45, 55, 66, 78
readCdfDataFrame, 46
readCdfGroupNames, 47
readCdfHeader, 44, 48, 81
readCdfIsFm, 49
readCdfNbrOfCellsPerUnitGroup, 50
readCdfQc, 52, 81
readCdfUnitNames, 53
readCdfUnits, 3, 41, 43, 44, 46–49, 53, 54, 56, 67, 82, 83
readCdfUnitsWriteMap, 9, 37, 56
readCel, 3, 38, 57, 59, 63–65, 67, 69
readCelHeader, 14, 32, 38, 61, 62, 67, 83, 84
readCelIntensities, 61, 63
readCelRectangle, 65
readCelUnits, 3, 8, 9, 34, 38, 57, 66, 78
readChp, 68
readClf, 69
readClfEnv, 70, 71
readClfHeader, 71
readPgf, 71
readPgfEnv, 72
readPgfHeader, 74
Startup, 34
strptime, 38
tpmap2bpmap, 40
tpmap2bpmap (writeTpmap), 84
TRUE, 9, 21, 26–28, 30–32, 34, 35, 38, 43, 46, 54, 56, 65, 67
updateCel, 75, 78
updateCelUnits, 78
vector, 9, 14, 17–19, 34–36, 43–48, 50, 53, 54, 56, 60, 66, 67, 75, 78
Verbose, 56, 67
writeCdf, 29, 80, 81–83
writeCdfHeader, 81, 82, 83
writeCdfQcUnits, 81, 82, 83
writeCdfUnits, 81, 82, 82
writeCelHeader, 83
writeTpmap, 84
xy2indices, 7