1 Introduction

The torrentR package provides some basic capabilities for working with data from Ion Torrent Systems.

The package and many of the underlying data types that it accesses are in a state of fairly rapid flux, so this document should be considered as a snapshot in time. Some of the methods and underlying data types are highly likely to change as the system evolves.

Instructions for building and installing the torrentR package can be found in the file torrentR/INSTALL.

2 Loading Data

The table below lists the data types that can be read with the torrentR package, along with the relevant functions for handling. The remainder of this document delves further into each data type and presents usage examples.

File Type	torrentR function(s)	Comments
*.dat	readDat, readDatCollection	Dat files contain the raw signal that comes directly from the PGM.
bfmask.bin	readBeadFindMask, readBeadFindMaskHeader	The bfmask.bin file contains information about the estimated classification of each well - whether or not it contains a bead, what kind of a bead it contains (test fragment or library), etc.
1.wells, 1.cafie-residuals	readWells	The 1.wells file is derived from the dat files and contains the estimate of the incorporoation signal for each flow in each well. The 1.cafie-residuals is an optional file that contains information about the CAFIE model fit.
rawlib*.sff	readSFF	The SFF file contains base calls and flow values and is the primary result delivered by the Analysis pipeline
Default.sam.parsed	readSamParsed	The Default.sam.parsed file contains alignment information for any library reads that were mapped to the genome
wellStats.txt, regionCafieDebug.txt	readTSV	The wellStats.txt and regionCafieDebug.txt files are tab-delimited text files containing information about the CAFIE model in the CAFIE-estimation and CAFIE-calling phases.
DefaultTFs.conf	readTfConf, readTfInfo	The DeftaultTFs.conf file contains information about the names and sequences of test fragments that may be present in the run.
TFTracking.txt	readTfStats, readTfInfo	The TFTracking.txt provides the identity of the test fragment for each bead identified as being some form of test fragment.

In the following sections we go through each data type to explore how it can be accessed and used. This vignette shows examples of reading data using a small dataset consisting of a small region cropped out of the middle of a 314 chip.

The remaining sections are loosely ordered in the chronology of their occurence in the data analysis pipeline.

3 DAT files

The DAT files contain the raw data that is ftp'ed directly from the Personal Genome Machine to the Torrent Server. They are the most basic data type handled by the torrentR package. A single DAT can be read with the readDat() function:

```
> library(torrentR)

Loading torrentR version 0.5.0

> dataDir <- system.file("extdata", package = "torrentR")
> dat1 <- readDat(sprintf("%s/acq_0000.dat", dataDir))
> str(dat1)

List of 10
 $ datFile   : chr "/tmp/Rinst1113396051/torrentR/extdata/acq_0000.dat"
 $ nCol      : int 50
 $ nRow      : int 50
 $ nFrame    : int 169
 $ nFlow     : int 1
 $ col       : int [1:2500] 0 1 2 3 4 5 6 7 8 9 ...
 $ row       : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ frameStart: num [1, 1:169] 0 0.068 0.136 0.204 0.272 0.34 0.408 0.476 0.544 0.612 ...
 $ frameEnd  : num [1, 1:169] 0.068 0.136 0.204 0.272 0.34 0.408 0.476 0.544 0.612 0.68 ..
 $ signal    : int [1:2500, 1:169] 0 0 0 1 1 1 2 0 0 1 ...
```

The example above reads all available frames and wells which make up the complete dataset for acq_0000.dat, the first nucleotide flow.

The readDatCollection() function can be used to read the data for multiple flows - for example, here is how one could read all available data for the first 8 flows:

```
> dat2 <- readDatCollection(datDir = dataDir, minFlow = 1, maxFlow = 8)
> str(dat2)

List of 11
 $ datFile   : chr [1:8] "/tmp/Rinst1113396051/torrentR/extdata/acq_0000.dat" "/tmp/Rinst1
 $ nCol      : int 50
 $ nRow      : int 50
 $ nFrame    : int 169
 $ nFlow     : int 8
 $ col       : int [1:2500] 0 1 2 3 4 5 6 7 8 9 ...
 $ row       : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
```

```
 $ frameStart: num [1:8, 1:169] 0 0 0 0 0 0 0 0 0.068 0.068 ...
 $ frameEnd  : num [1:8, 1:169] 0.068 0.068 0.068 0.068 0.068 0.068 0.068 0.068 0.136 0.13
 $ signal    : int [1:2500, 1:1352] 0 0 0 1 1 1 2 0 0 1 ...
 $ flow      : int [1:8] 1 2 3 4 5 6 7 8
```

Note how the returned list is essentially the same thing that is returned by readDat() but with an addition list element flow recording which flows were returned and with extra columns concatenated to the signal matrix for the additional flows.

For the toy dataset used here it is actually feasible to load all of the data for a subset of the flows because it is based on a small cropped region, however in most situations an attempt to read all data for one or a number of flows would be at risk of requiring more memory than is available. In some of the following examples we will explore some ways to sensibly limit or sub-sample the data to load. As with many of the functions, the man pages for readDat() and readDatCollection() provide more detail on some of the available options to help keep jobs manageable.

4 The bfmask.bin file

One of the first things that happens in the analysis pipeline is an attempt to classify wells into those that are loaded, or "bead wells", and those that are not - the "empty wells". Furthermore, each bead well is tested against the expected library and Test Fragment (TF) key sequences to partition the set of all bead wells into mutually-exlusive categories: duds (beads without sufficient incorporation signal and hence presumably without sufficient attached template), live library beads and live TF beads.

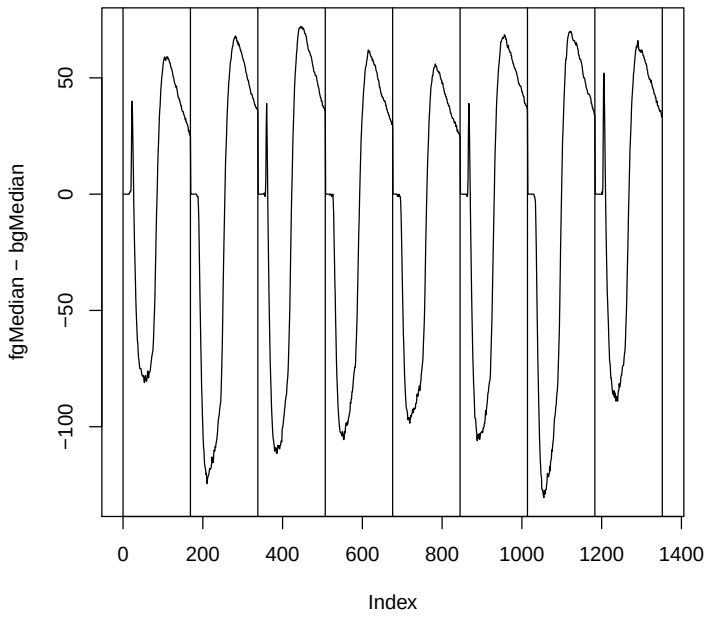
This information on the classification of wells, along with some well classification information derived further downstream, is written into a file named bfmask.bin in the analysis directory. The functions readBeadFindMask() and readBeadFindMaskHeader() can be used to parse the information in the bfmask.bin. It is a relatively compact file and it can typcially be loaded in its entirity without fear of running into memory issues.

```
> bfHeader <- readBeadFindMaskHeader(sprintf("%s/bfmask.bin", dataDir))
> str(bfHeader)

List of 2
 $ nRow: int 50
 $ nCol: int 50

> bf1 <- readBeadFindMask(sprintf("%s/bfmask.bin", dataDir))
> str(bf1)

List of 20
 $ beadFindMaskFile       : chr "/tmp/Rinst1113396051/torrentR/extdata/bfmask.bin"
 $ nCol                   : int 50
 $ nRow                   : int 50
 $ col                    : int [1:2500] 0 1 2 3 4 5 6 7 8 9 ...
 $ row                    : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
```

```
 $ maskEmpty              : int [1:2500] 0 0 1 1 0 0 0 0 0 0 ...
 $ maskBead               : int [1:2500] 1 1 0 0 1 1 1 1 1 1 ...
 $ maskLive               : int [1:2500] 1 1 0 0 0 1 1 1 1 1 ...
 $ maskDud                : int [1:2500] 0 0 0 0 1 0 0 0 0 0 ...
 $ maskAmbiguous          : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ maskTF                 : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ maskLib                : int [1:2500] 1 1 0 0 0 1 1 1 1 1 ...
 $ maskPinned             : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ maskIgnore             : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ maskWashout            : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ maskKeypass            : int [1:2500] 0 1 0 0 0 1 1 1 1 1 ...
 $ maskFilteredBadKey     : int [1:2500] 0 0 0 0 1 0 0 0 0 0 ...
 $ maskFilteredShort      : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ maskFilteredBadPPF     : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ maskFilteredBadResidual: int [1:2500] 1 0 0 0 0 0 0 0 0 0 ...
```

The information in the bfmask.bin file provides a very convenient way to intelligently load manageable subsets of information. For example:

```
> set.seed(0)
> libSample <- sample(which(bf1$maskLib == 1), 100)
> emptySample <- sample(which(bf1$maskEmpty == 1), 100)
> dat.lib <- readDatCollection(datDir = dataDir, minFlow = 1, maxFlow = 8,
+     col = bf1$col[libSample], row = bf1$row[libSample])
> dat.empty <- readDatCollection(datDir = dataDir, minFlow = 1,
+     maxFlow = 8, col = bf1$col[emptySample], row = bf1$row[emptySample])
> fgMedian <- apply(dat.lib$signal, 2, median)
> bgMedian <- apply(dat.empty$signal, 2, median)
> plot(fgMedian - bgMedian, type = "l")
> flowBoundaries <- (0:10) * dat.lib$nFrame
> abline(v = flowBoundaries)
```



5 Wells files

The 1.wells file stores the estimated incorporation signal that is derived from the raw data in the DAT files. It can be read with the readWells() function though as with reading DAT files one often needs to use some of the function options to load a subset of the available data.

A call to str() gives a quick peek of what is available in the returned list.

```
> wellFile <- system.file("extdata/1.wells", package = "torrentR")
> wells1 <- readWells(wellFile)
> str(wells1)
```

```
List of 14
 $ beadFindMaskFile: chr "/tmp/Rinst1113396051/torrentR/extdata/bfmask.bin"
 $ mask            :List of 10
  ..$ empty    : int [1:2500] 0 0 1 1 0 0 0 0 0 0 ...
  ..$ bead     : int [1:2500] 1 1 0 0 1 1 1 1 1 1 ...
  ..$ live     : int [1:2500] 1 1 0 0 0 1 1 1 1 1 ...
  ..$ dud      : int [1:2500] 0 0 0 0 1 0 0 0 0 0 ...
  ..$ ambiguous: int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ tf       : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ lib      : int [1:2500] 1 1 0 0 0 1 1 1 1 1 ...
  ..$ pinned   : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ ignore   : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
  ..$ washout  : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ col             : int [1:2500] 0 1 2 3 4 5 6 7 8 9 ...
```

```
 $ row             : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ flow            : int [1:260] 1 2 3 4 5 6 7 8 9 10 ...
 $ flowBase        : chr [1:260] "T" "A" "C" "G" ...
 $ wellFile        : chr "/tmp/Rinst1113396051/torrentR/extdata/1.wells"
 $ nCol            : int 50
 $ nRow            : int 50
 $ nLoaded         : int 2500
 $ nFlow           : int 260
 $ flowOrder       : chr "TACGTACGTCTGAGCATCGATCGATGTACAGCTACGTACGTCTGAGCATCGATCGATGTACAGC
 $ rank            : int [1:2500] 0 0 0 0 0 0 0 0 0 0 ...
 $ signal          : num [1:2500, 1:260] 2.85 3.18 0 0 0 ...
```

This next example reads in a rectangular slice, by specifing the columns and row bounds. Note that column and row are 0-based indices (as is also true of the corresponding values returned by the DAT and bfmask.bin readers).

```
> wells2 <- readWells(wellFile, colMin = 0, colMax = 49, rowMin = 10,
+     rowMax = 20)
```

This example reads in 4 wells at coordinates (1,6), (2,7), (3,8) and (4,9).

```
> wells3 <- readWells(wellFile, col = c(1, 2, 3, 4), row = c(6,
+     7, 8, 9))
```

Particular elements in the returned list can be accessed directly with the usual list $ operator:

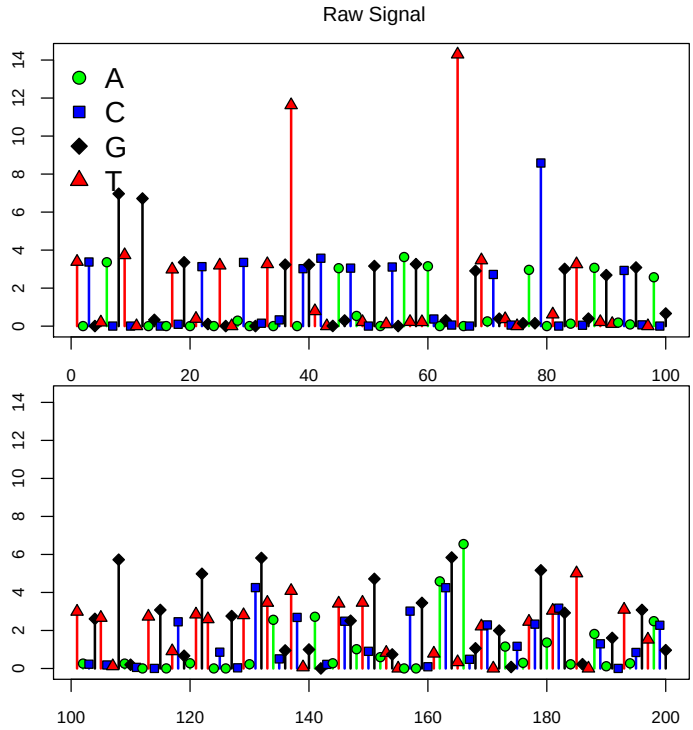
```
> wells3$col

[1] 1 2 3 4

> wells3$mask$tf

[1] 0 0 0 0

> wells3$signal[, 1:6]

         [,1]        [,2]     [,3]        [,4]       [,5]     [,6]
[1,] 0.000000 0.000000000 0.000000 0.000000000 0.00000000 0.000000
[2,] 1.805529 0.001894657 1.607895 0.139767021 0.03309803 1.875204
[3,] 0.000000 0.000000000 0.000000 0.000000000 0.00000000 0.000000
[4,] 2.048241 0.002532081 2.066236 0.002506921 0.01547170 2.208533
```

The next few sections describe some of the functions available for working with data from the 1.wells file:

5.1 plotIonogram()

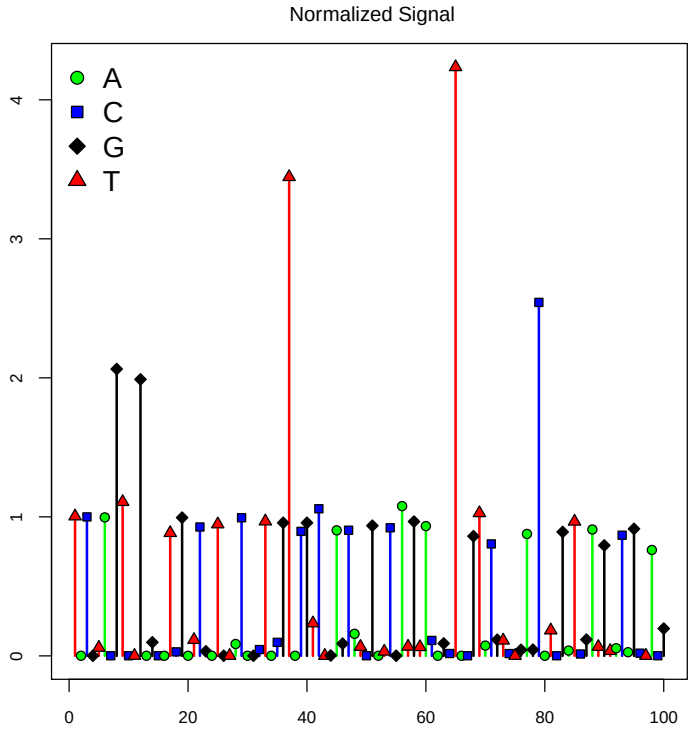
The plotIonogram() function provides functionality for basic plotting of data for a well:

```
> wells4 <- readWells(wellFile, col = bf1$col[libSample], row = bf1$row[libSample])
> plotIonogram(wells4$signal[1, ], wells4$flowOrder, wells4$flow,
+     flowRange = 1:200, flowsPerWindow = 100)
```



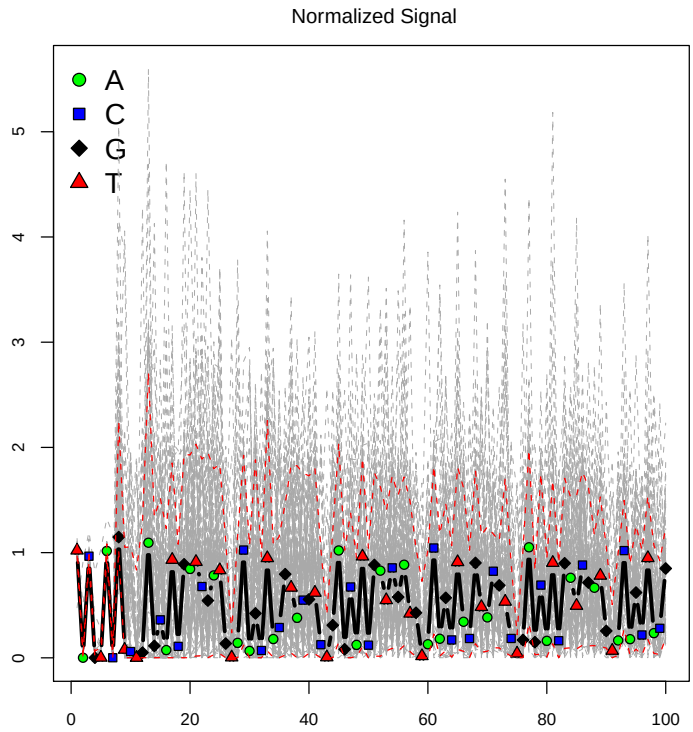
By default plotIonogram() plots the raw data, if key-normalized data is prefered it can be produced by setting plotType to ”norm” and supplying the key sequence:

```
> plotIonogram(wells4$signal[1, ], wells4$flowOrder, wells4$flow,
+     flowRange = 1:100, flowsPerWindow = 100, plotType = "norm",
+     keySeq = "TCAG")
```



plotIonogram() can also show data for multiple wells on the same plot, though this often only makes sense when done for wells that are sequencing the same template (such as wells sequencing the same TF)

```
> plotIonogram(wells4$signal, wells4$flowOrder, wells4$flow, flowRange = 1:100,
+     flowsPerWindow = 100, plotType = "norm", keySeq = "TCAG")
```



5.2 normalizeIonogram()

normalizeIonogram() can be used to perform normalization based on a provided key sequence. It can be applied to one well at a time or to multiple wells in a batch. To perform the normalization it needs to be told the key sequence and the flow order. The flow order is available in the list returned by readWells() but the key sequence must be specified by the user.

```
> seqKey <- "TCAG"
> norm1 <- normalizeIonogram(wells4$signal[1, ], seqKey, wells4$flowOrder)
> dim(norm1$normalized)

[1]   1 260

> norm2 <- normalizeIonogram(wells4$signal[, 1:50], seqKey, wells4$flowOrder)
> dim(norm2$normalized)

[1] 100  50
```

5.3 CAFIE residuals

Lastly, data other than estimated incorporation can be written to wells-formatted files. One example is the residuals after CAFIE-calling, which can be useful to inspect to examine possible errors. There is an option –cafie-residuals that can be supplied to the Analysis executable that causes a 1.cafie-residuals file in wells format to be written out after CAFIE calling is complete.

6 wellStats.txt and regionCafieDebug.txt

The wellStats.txt and regionCafieDebug.txt files are optional files that are sometimes available for an analysis run (created by the –well-stat-file and –region-cafie-debug-file options to Analysis). They contain per-read information generated at the time of CAFIE parameter estimation (regionCafieDebug.txt) and at the time of final CAFIE basecalling (wellStats.txt). This information can be helpful in run diagnostics.

Each file is written in tab-delimited text format, all columns are numeric and the first line is a header line. These and other tab-delimited text files can be read with readTSV():

6.1 regionCafieDebug.txt

```
> rCafie <- readTSV(sprintf("%s/regionCafieDebug.txt", dataDir))
> str(rCafie)

List of 13
 $ col        : num [1:1794] 0 9 5 6 7 11 8 32 16 24 ...
 $ row        : num [1:1794] 0 0 0 0 0 0 0 0 0 0 ...
 $ region_id  : num [1:1794] 0 0 0 0 0 0 0 0 0 0 ...
 $ res_old    : num [1:1794] 0.0467 0.0084 0.003 0.0031 0.0019 0.003 0.0015 0.0011 0.0025
 $ res_new    : num [1:1794] 0.2344 0.0756 0.0343 0.0451 0.0235 ...
 $ ppf        : num [1:1794] 0.562 0.469 0.406 0.438 0.406 ...
 $ ppf2       : num [1:1794] 0.917 0.45 0.433 0.483 0.433 ...
 $ ssq        : num [1:1794] 6.479 1.112 0.863 1.125 0.937 ...
 $ cf         : num [1:1794] 0.03 0 0.0202 0.004 0.007 0.0011 0.004 0 0.0046 0.0257 ...
 $ ie         : num [1:1794] 0.03 0.0101 0.008 0.0076 0.0032 0.0115 0.008 0.01 0.0052 0.03
 $ dr         : num [1:1794] 2e-04 2e-04 2e-04 2e-04 2e-04 2e-04 2e-04 2e-04 2e-04 2e-04 .
 $ used_in_est: num [1:1794] 0 0 1 1 1 1 1 0 1 0 ...
 $ clonal     : num [1:1794] 1 1 1 1 1 1 1 1 1 1 ...
```

There is one entry for each read that was considered in regional CAFIE parameter estimation. Some of the field names returned are quite obvious, those that might not be are as follows:

region_id: The region assignment for each read.

res_new: The median absolute CAFIE residual in the first 40 flows

ppf: The percentage of the first 40 flows that are positive (i.e. have one or more estimated incorporation).

cf,ie,dr: the per-read estimates of CF, IE and DR.

6.2 wellStats.txt

The wellStats.txt file contains per-read information available at the time of final CAFIE calling. There is one line for each read that goes through the CAFIE calling process.

```
> wStats <- readTSV(sprintf("%s/wellStats.txt", dataDir))
> str(wStats)
```

```
List of 18
 $ col       : num [1:1837] 4 5 7 1 6 8 9 22 11 21 ...
 $ row       : num [1:1837] 0 0 0 0 0 0 0 0 0 0 ...
 $ isTF      : num [1:1837] 0 0 0 0 0 0 0 0 0 0 ...
 $ isLib     : num [1:1837] 0 1 1 1 1 1 1 1 1 1 ...
 $ isDud     : num [1:1837] 1 0 0 0 0 0 0 0 0 0 ...
 $ isAmbg    : num [1:1837] 0 0 0 0 0 0 0 0 0 0 ...
 $ nCall     : num [1:1837] 26 109 114 120 113 114 118 112 109 102 ...
 $ cf        : num [1:1837] 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 0.012 ..
 $ ie        : num [1:1837] 0.0088 0.0088 0.0088 0.0088 0.0088 0.0088 0.0088 0.0088 0.0088
 $ dr        : num [1:1837] 0.00023 0.00023 0.00023 0.00023 0.00023 0.00023 0.00023 0.00023 0.0002
 $ keySNR    : num [1:1837] 0 18.24 15.54 12.26 9.58 ...
 $ keySD     : num [1:1837] 0.014 0.05 0.061 0.08 0.101 0.055 0.024 0.031 0.068 0.02 ...
 $ keySig    : num [1:1837] 0 0.905 0.947 0.985 0.966 ...
 $ oneSig    : num [1:1837] 0 0.913 0.951 0.986 0.967 ...
 $ zeroSig   : num [1:1837] 0 0.008 0.004 0.001 0.001 0.009 0.001 0.005 0.001 0.014 ...
 $ ppf       : num [1:1837] 0.233 0.433 0.417 0.467 0.467 0.4 0.45 0.433 0.333 0.367 ...
 $ medAbsRes : num [1:1837] 0.032 0.038 0.041 0.04 0.027 0.026 0.059 0.024 0.023 0.028 ...
 $ multiplier: num [1:1837] 1 0.288 0.336 0.325 0.338 0.296 0.355 0.332 0.357 0.3 ...
```

7 SFF files

The Standard Flowgram File (SFF) is the primary result from the Analysis pipeline, it contains the basecalls as well as CAFIE-corrected flow values and the mapping from bases to flows. The readSFF() function can be used to load SFF data directly into R.

The following example shows a basic call to readSFF() requesting that it read all the information in the SFF file.

```
> sffFile <- sprintf("%s/rawlib.sff", dataDir)
> sff1 <- readSFF(sffFile)
> str(sff1)

List of 13
 $ nFlow           : int 260
 $ col             : int [1:1419] 5 7 1 6 8 9 22 11 21 16 ...
 $ row             : int [1:1419] 0 0 0 0 0 0 0 0 0 0 ...
 $ length          : int [1:1419] 148 146 146 140 150 150 132 148 127 150 ...
 $ fullLength      : int [1:1419] 148 146 146 140 153 157 132 148 127 157 ...
 $ clipQualLeft    : int [1:1419] 5 5 5 5 5 5 5 5 5 5 ...
 $ clipQualRight   : int [1:1419] 0 0 0 0 0 0 0 0 0 0 ...
 $ clipAdapterLeft : int [1:1419] 0 0 0 0 0 0 0 0 0 0 ...
 $ clipAdapterRight: int [1:1419] 0 0 0 0 0 0 0 0 0 0 ...
 $ flow            : num [1:1419, 1:260] 0.92 1.02 1.04 1.14 1.01 0.94 1.02 1.03 1.05 0.94
 $ base            : chr [1:1419] "TCAGGTCAATAGTAACAACGGCAGCAATCCATACATGACACCAACATAGGCGACC
 $ qual            : int [1:1419, 1:150] 33 33 33 33 32 33 33 33 33 33 ...
 $ flowIndex       : int [1:1419, 1:150] 1 1 1 1 1 1 1 1 1 1 ...
```

The next example restricts to reading the SFF entries corresponding to the random sample of library wells. Note that not all wells classified as library in

the bfmask.bin make it through to the SFF file - additional more stringent filters are applied to determine what makes it into the SFF and some reads initially classified as library drop out along the way.

```
> sff2 <- readSFF(sffFile, col = bf1$col[libSample], row = bf1$row[libSample])
```

The documentation for readSFF contains details about the returned data. Some of the key returned values include flow (the CAFIE-corrected flow values), base (the base calls), qual (Phred-style quality values for each base) and flowIndex (the mapping from bases to flows).

8 The Default.sam.parsed file

Information about library read alignments to the genome is stored in SAM and BAM files. These formats are described at samtools.sourceforge.net/SAM1.pdf and ideally torrentR would be able to directly parse them, but at least until now it has been expedient to post-process the SAM format into a tab-delimited text file called Default.sam.parsed. However this file is neither a standard nor supported file so its use is being phased out and the functionality related to it will be replaced by something else. So what is described in this section may be redundant/unavailable by the time you read this.

The function readSamParsed provides a means to read this file:

```
> samFile <- sprintf("%s/Default.sam.parsed", dataDir)
> sam1 <- readSamParsed(samFile)
```

By default it returns only the well coordinates and the Q10 length of the read (which is defined as the maximal position in the read at which the total read error rate is equal to Q10 or 10%. The fields option allows for selection of other values in the returned list:

```
> sam2 <- readSamParsed(samFile, fields = c("name", "q17Len", "qDNA.a",
+     "match.a", "tDNA.a"))
```

The qDNA.a and tDNA.a contain the query and target sequences respectively. The following example finds the first well with a Q17 length larger than 100 and uses seqToFlow() to get the predicted ideal flow values for each in the first 30 flows:

```
> goodWell <- which(sam2$q17Len > 99)[1]
> rbind(ref = seqToFlow(sam2$tDNA.a[goodWell], wells4$flowOrder,
+     30), read = seqToFlow(sam2$qDNA.a[goodWell], wells4$flowOrder,
+     30))
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14]
ref     0    0    0    1    1    0    1    0    0     0     0     0     2     0
read    0    0    0    1    1    0    1    0    0     0     0     0     2     0
     [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24] [,25] [,26]
ref      0     0     1     0     0     1     0     0     1     0     1     0
read     0     0     1     0     0     1     0     0     1     0     1     0
     [,27] [,28] [,29] [,30]
ref      0     2     1     2
read     0     2     1     2
```