pqsfinder: User Guide

Jiří Hon, Matej Lexa, Tomáš Martínek

29 October 2019

Abstract

Instructions on how to use *pqsfinder* package for detecting DNA sequence patterns that are likely to fold into an intramolecular G-quadruplex.

Package

pqsfinder 2.2.0

Contents

1 Introduction

The main functionality of the *pqsfinder* package is to detect DNA and RNA sequence patterns that are likely to fold into an intramolecular G-quadruplex (G4). G4 is a nucleic acid structure that can form as an alternative to the canonical B-DNA. G4s are believed to be involved in regulation of diverse biological processes, such as telomere maintenance, DNA replication, chromatin formation, transcription, recombination or mutation (Maizels and Gray 2013; Kejnovsky, Tokan, and Lexa 2015). The main idea of our algorithmic approach is based on the fact that G4 structures arise from compact sequence motifs composed of four consecutive and possibly imperfect guanine runs (G-run) interrupted by loops of semi-arbitrary lengths. The algorithm first identifies four consecutive G-run sequences. Subsequently, it examines the potential of such G-runs to form a stable G4 and assigns a corresponding quantitative score to each. Non-overlapping potential quadruplex-forming sequences (PQS) with positive score are then reported.

It is important to note that unlike many other approaches, our algorithm is able to detect sequences responsible for G4s folded from imperfect G-runs containing bulges or mismatches and as such is more sensitive than competing algorithms.[1] We also believe the presented solution is the most scalable, since it can be easily and quickly customized (see chapter Customizing detection algorithm for details). The program can be made to detect novel or experimental G4 types that might be discovered or studied in future.

[1] We have tested *pqsfinder* on experimentally verified G4 sequences. The results of that work are reflected in default settings of searches. Details of these tests will be presented elsewhere.

For those interested in non-B DNA, we have previously authored a similar package that can be used to search for triplexes, another type of non-B DNA structure. For details, please see *triplex* package landing page.

2 G-quadruplex detection

As usual, before first package use, it is necessary to load the *pqsfinder* package using the following command:

```
library(pqsfinder)
```

Identification of potential quadruplex-forming sequences (PQS) in DNA is performed using the `pqsfinder` function. This function has one required parameter representing the studied DNA sequence in the form of a `DNAString` object and several modifying options with predefined values. For complete description, please see `pqsfinder` function man page.

2.1 Basic quadruplex detection

As a simple example, let's find all PQS in a short DNA sequence.

```
seq <- DNAString("TTTTGGGCGGGAGGAGTGGAGTTTTTAACCCCAAAAATTTGGGAGGGTGGGTGGGAGAA")
pqs <- pqsfinder(seq, min_score = 20)
pqs
```

```
##   PQS views on a 59-letter DNAString subject
## subject: TTTTGGGCGGGAGGAGTGGAGTTTTTAACCCCAAAAATTTGGGAGGGTGGGTGGGAGAA
## quadruplexes:
##     start width score strand nt nb nm
## [1]     5    17    33      +  3  2  0 [GGGCGGGAGGAGTGGAG]
## [2]    41    15    73      +  3  0  0 [GGGAGGGTGGGTGGG]
```

Detected PQS are returned in the form of a `PQSViews` class, which represents the basic container for storing a set of views on the same input sequence based on `XStringViews` object from *Biostrings* package. Each PQS in the view is defined by (i) start location, (ii) width, (iii) score, (iv) strand, (v) number of G-tetrads `nt`, (vi) number of bulges `nb` and (vii) number of mismatches `nm`. The first four values can be accessed by standard functions `start(x)`, `width(x)` and `score(x)` and `strand(x)`. To get other PQS features, please use `elementMetadata(x)` function. It additionaly provides run and loop lengths of the detected PQS (`rl1`, `rl2`, `rl3`, `ll1`, `ll2`, `ll3`).

```
elementMetadata(pqs)
```

```
## DataFrame with 2 rows and 11 columns
##        strand     score        nt        nb        nm       rl1       rl2
##   <character> <integer> <integer> <integer> <integer> <integer> <integer>
## 1           +        33         3         2         0         3         3
## 2           +        73         3         0         0         3         3
##         rl3       ll1       ll2       ll3
##   <integer> <integer> <integer> <integer>
## 1         4         1         1         1
## 2         3         1         1         1
```

By default, `pqsfinder` function reports only the locally best non-overlapping PQS, ignoring any other that would overlap it. However, it's possible to change the default behavior by setting the `overlapping` option to `TRUE`.

```
pqsfinder(seq, overlapping = TRUE, min_score = 30)
```

```
##   PQS views on a 59-letter DNAString subject
## subject: TTTTGGGCGGGAGGAGTGGAGTTTTTAACCCCAAAAATTTGGGAGGGTGGGTGGGAGAA
## quadruplexes:
##     start width score strand nt nb nm
## [1]     5    17    33      +  3  2  0 [GGGCGGGAGGAGTGGAG]
## [2]     5    43    37      +  3  0  0 [GGGCGGGAGGAGTGGAG...CCAAAAATTTGGGAGG
G]
## [3]     5    47    33      +  3  0  0 [GGGCGGGAGGAGTGGAG...AAATTTGGGAGGGTGG
G]
## [4]     9    43    37      +  3  0  0 [GGGAGGAGTGGAGTTTT...AAATTTGGGAGGGTGG
G]
## [5]     9    47    33      +  3  0  0 [GGGAGGAGTGGAGTTTT...TTGGGAGGGTGGGTGG
G]
## [6]    41    15    73      +  3  0  0 [GGGAGGGTGGGTGGG]
## [7]    41    17    53      +  3  1  0 [GGGAGGGTGGGTGGGAG]
## [8]    42    13    30      +  2  0  0 [GGAGGGTGGGTGG]
```

Alternatively, it's possible to get numbers of all overlapping PQS at each position of the input sequence. To achieve that, set `deep` option to `TRUE` and then call `density(x)` function on the `PQSViews` object:[2]

[2] Clusters of overlapping PQS usually have steep edges when the number of neighboring G-runs is low, but could be more spread out in other situations.

```
pqs <- pqsfinder(seq, deep = TRUE, min_score = 20)
density(pqs)
```

```
##  [1]   0   0   0   0  38  42  43  43  71  81  86  86  99 103 103 107 107 111
## [19] 112 108 108 104 104 104 104 104 104 104 104 104 104 104 104 104 104 104
## [37] 104 104 104 104 124 132 132 125 125 125 125 101 101 101 101  63  63  63
## [55]  55  11  11   0   0
```

The following example shows, how such density vector could be simply visualized along the input sequence using *Gviz* from Bioconductor.

```
library(Gviz)
ss <- DNAStringSet(seq)
names(ss) <- "chr1"
dtrack <- DataTrack(
  start = 1:length(density(pqs)), width = 1, data = density(pqs),
  chromosome = "chr1", genome = "", name = "density")
strack <- SequenceTrack(ss, chromosome = "chr1", name = "sequence")
suppressWarnings(plotTracks(c(dtrack, strack), type = "h"))
```

2.2 Modifying basic algorithm options

Depending on the particular type of PQS you want to detect, the algorithm options can be tuned to find the PQS effectively and exclusively. The table bellow gives an overview of all basic algorithm options and their descriptions.

Option name	Description
`strand`	Strand specification (`+`, `-` or `*`).
`overlapping`	If true, than overlapping PQS will be reported.
`max_len`	Maximal total length of PQS.
`min_score`	Minimal score of PQS to be reported. The default value 52 shows the best balanced accuracy on human G4 sequencing data (Chambers et al. 2015).
`run_min_len`	Minimal length of each PQS run (G-run).
`run_max_len`	Maximal length of each PQS run.
`loop_min_len`	Minimal length of each PQS inner loop.
`loop_max_len`	Maximal length of each PQS inner loop.
`max_bulges`	Maximal number of runs containing a bulge.
`max_mismatches`	Maximal number of runs containing a mismatch.
`max_defects`	Maximum number of defects in total (#bulges + #mismatches).

The more you narrow these options in terms of shorter PQS length, narrower run or loop length ranges and lower number of defects, the faster the detection process will be, with a possible loss of sensitivity.

Important note: In each G-run, the algorithm allows at most one type of defect and at least one G-run must be perfect, that means without any defect. Therefore the values of `max_bulges`, `max_mismatches` and `max_defects` must fall into the range from 0 to 3.

Example 1: If you are insterested solely in G-quadruplexes with perfect G-runs, just restrict `max_defects` to zero:

```
pqsfinder(seq, max_defects = 0, min_score = 20)
```

```
##   PQS views on a 59-letter DNAString subject
## subject: TTTTGGGCGGGAGGAGTGGAGTTTTTAACCCCAAAAATTTGGGAGGGTGGGTGGGAGAA
## quadruplexes:
##     start width score strand nt nb nm
## [1]     6    14    29      +  2  0  0 [GGCGGGAGGAGTGG]
## [2]    41    15    73      +  3  0  0 [GGGAGGGTGGGTGGG]
```

Example 2: In case you don't mind defects in G-runs, but you want to report only high-quality PQS, increase `min_score` value:

```
pqsfinder(seq, min_score = 70)
```

```
##   PQS views on a 59-letter DNAString subject
## subject: TTTTGGGCGGGAGGAGTGGAGTTTTTAACCCCAAAAATTTGGGAGGGTGGGTGGGAGAA
## quadruplexes:
##     start width score strand nt nb nm
## [1]    41    15    73      +  3  0  0 [GGGAGGGTGGGTGGG]
```

3 Exporting results

As mentioned above, the results of detection are stored in the `PQSViews` object. Because the `PQSViews` class is only an extension of the `XStringViews` class, all operations applied to the `XStringViews` object can also be applied to the `PQSViews` object as well.

Additionaly, `PQSViews` class supports a conversion mechanism to create `GRanges` objects. Thus, all detected PQS can be easily transformed into elements of a `GRanges` object and saved as a GFF3 file, for example.

3.1 GRanges conversion and export to GFF3

In this example, the output of the `pqsfinder` function will be stored in a `GRanges` object and subsequently exported as a GFF3 file. At first, let's do the conversion using the following command:

```
gr <- as(pqs, "GRanges")
gr
```

```
## GRanges object with 2 ranges and 12 metadata columns:
##       seqnames    ranges strand |     score        nt        nb        nm
##          <Rle> <IRanges>  <Rle> | <integer> <integer> <integer> <integer>
##   [1]     chr1      5-21      + |        33         3         2         0
##   [2]     chr1     41-55      + |        73         3         0         0
##             rl1       rl2       rl3       ll1       ll2       ll3
##       <integer> <integer> <integer> <integer> <integer> <integer>
##   [1]         3         3         4         1         1         1
##   [2]         3         3         3         1         1         1
##            source        type
##       <character> <character>
##   [1]   pqsfinder   G_quartet
##   [2]   pqsfinder   G_quartet
##   -------
##   seqinfo: 1 sequence from an unspecified genome
```

Please note that the chromosome name is arbitrarily set to `chr1`, but it can be freely changed to any other value afterwards. In the next step the resulting `GRanges` object is exported as a GFF3 file.

```
library(rtracklayer)
export(gr, "test.gff", version = "3")
```

Please note, that it is necessary to load the `rtracklayer` library before running the `export` command. The contents of the resulting GFF3 file are:

```
##gff-version 3
##source-version rtracklayer 1.46.0
##date 2019-10-29
chr1 pqsfinder G_quartet 5 21 33 + .
   nt=3;nb=2;nm=0;rl1=3;rl2=3;rl3=4;ll1=1;ll2=1;ll3=1
chr1 pqsfinder G_quartet 41 55 73 + .
   nt=3;nb=0;nm=0;rl1=3;rl2=3;rl3=3;ll1=1;ll2=1;ll3=1
```

Another possibility of utilizing the results of detection is to transform the `PQSViews` object into a `DNAStringSet` object, another commonly used class of the `Biostrings` package. PQS stored inside `DNAStringSet` can be exported into a FASTA file, for example.

3.2 DNAStringSet conversion and export to FASTA

In this example, the output of the `pqsfinder` function will be stored in a `DNAStringSet` object and subsequently exported as a FASTA file. At first, let's do the conversion using the following command:

```
dss <- as(pqs, "DNAStringSet")
dss
```

```
##   A DNAStringSet instance of length 2
##     width seq                                               names
## [1]    17 GGGCGGGAGGAGTGGAG                                 pqsfinder;G_quar
t...
## [2]    15 GGGAGGGTGGGTGGG                                   pqsfinder;G_quar
t...
```

In the next step, the `DNAStringSet` object is exported as a FASTA file.

```
writeXStringSet(dss, file = "test.fa", format = "fasta")
```

The contents of the resulting FASTA file are:

```
>pqsfinder;G_quartet;start=5;end=21;strand=+;score=33;nt=3;nb=2;nm=0;rl1=3;rl2=
3;rl3=4;ll1=1;ll2=1;ll3=1;
GGGCGGGAGGAGTGGAG
>pqsfinder;G_quartet;start=41;end=55;strand=+;score=73;nt=3;nb=0;nm=0;rl1=3;rl2
=3;rl3=3;ll1=1;ll2=1;ll3=1;
GGGAGGGTGGGTGGG
```

Please, note that all attributes of detection such as start position, end position and score value are stored as a `name` parameter (inside the `DNAStringSet`), and thus, they are also shown in the header line of the FASTA format (the line with the initial `>` symbol).

4 A real world example

In the following example, we load the human genome from the *BSgenome* package and identify all potential G4 (PQS) in the region of AHNAK gene on chromose 11. We then export the identified positions into a genome annotation track (via a GFF3 file) and an additional FASTA file. Finally, we plot some graphs showing the PQS score distribution and the distribution of PQS along the studied genomic sequence.

1. Load necessary libraries and genomes.

```
library(pqsfinder)
library(BSgenome.Hsapiens.UCSC.hg38)
library(rtracklayer)
library(ggplot2)
library(Gviz)
```

2. Retrive AHNAK gene annotation.

```
gnm <- "hg38"
gene <- "AHNAK"
# Load cached AHNAK region track:
load(system.file("extdata", "gtrack_ahnak.RData", package="pqsfinder"))
# Alternatively, query biomaRt API with the following commands:
# library(biomaRt)
# gtrack <- BiomartGeneRegionTrack(genome = gnm, symbol = gene, name = gen
e)
```

3. Get AHNAK sequence from *BSgenome* package extended by 1000 nucleotides on both sides.

```
extend <- 1000
seq_start <- min(start(gtrack)) - extend
seq_end <- max(end(gtrack)) + extend
chr <- chromosome(gtrack)
seq <- Hsapiens[[chr]][seq_start:seq_end]
```

4. Search for PQS on both strands.

```
pqs <- pqsfinder(seq, deep = TRUE)
```

5. Display the results.

```
pqs
```

```
##   PQS views on a 124694-letter DNAString subject
## subject: GCGGGTGTCTGTAATCCCAGCTACTTGGGAGGC...ATGCACCAGCTGCACCTAGCATTTTC
AGATCC
## quadruplexes:
##         start width score strand nt nb nm
##   [1]     778    29    85      +  4  1  0 [GGGGAGGGGGAGCAAGGGGTGTAAGAGG
G]
##   [2]    1071    39    71      -  5  2  1 [CCCCCTCTAGTCCCAA...CCACACTCTG
TCCCC]
##   [3]    1846    29    54      -  3  0  0 [CCCTTCACCTTCCCTCCCTGTCGTCTCC
C]
##   [4]    1912    32    60      -  4  2  0 [CCTCCTCCCCGAGTCACACCCAACTCATC
CCC]
##   [5]    2958    34    50      -  4  3  0 [CCCCCGGGGTTCCCGCCATTCTCCTGCCT
CAGCC]
##   [6]    4531    42    58      -  4  2  0 [CCCCAAAACGTTCCCT...CCAATCCATA
TCCCC]
##   [7]    5111    21    64      -  3  0  0 [CCCAGCCCAAATCCCTTACCC]
##   [8]    7613    33    52      +  4  3  0 [GGGCCCTGAGGGAAAAGTGAGGGGGTGGC
CGGG]
##   [9]    7877    20    52      -  3  1  0 [CCCTCCCAGCCACGAGCCCC]
##   ...     ...   ...   ...    ... ... ... ...
## [184] 123115    33    51      -  4  1  1 [CCCCTCCCTTCAACATTCTAGGCTTCCCC
CACC]
## [185] 123174    27    56      +  3  0  0 [GGGCATGTGGGCAGCTGGTGGGATGGG]
## [186] 123462    26    58      -  4  1  1 [CAGCCCTGAGCCCCGACCCCTTCTCC]
## [187] 123604    46    64      -  4  1  0 [CCCCTGATCCATTCAA...TCTGCTCCTC
ACCCC]
## [188] 123830    26    47      +  3  1  0 [GGAGAGCCCAGCGGGGATGGGAAGGG]
## [189] 123891    18    52      +  3  1  0 [GGGCAGGGCGGGGGACAG]
## [190] 123977    44    74      -  5  3  0 [CTCCCCTACCCACCAC...TTGCATCCCA
CCCCC]
## [191] 124288    30    72      -  5  1  2 [CCCATGGCCCACCCAGAACCCCCGACCCA
C]
## [192] 124618    17    53      -  3  1  0 [CAGCCATCCCCCCACCC]
```

6. Sort the results by score to see the best one.

```
pqs_s <- pqs[order(score(pqs), decreasing = TRUE)]
pqs_s
```

```
##   PQS views on a 124694-letter DNAString subject
## subject: GCGGGTGTCTGTAATCCCAGCTACTTGGGAGGC...ATGCACCAGCTGCACCTAGCATTTTC
AGATCC
## quadruplexes:
##         start width score strand nt nb nm
##   [1] 114398    42   125      +  6  3  0 [GGGACCCGGGAGTGGG...GGGGGCCGCT
GGGGG]
##   [2] 103330    36   118      -  6  2  1 [CCCTGCCCTTCCCTCC...CCCCACCGAC
CCCCC]
##   [3]  73196    37   115      -  5  1  0 [CCCCCGACACACCTCC...TCTCCACCCG
CCCCC]
##   [4] 113317    47   113      +  6  3  0 [GGGAGTTGGGCGGGGG...GAGGGGAAGG
GGCGG]
##   [5] 109892    24   106      -  4  0  0 [CCCCTCCCCATCACCCCCTTCCCC]
##   [6] 114459    36   104      -  5  2  0 [CCCCCTCCCCGCATCC...CCCCTGTCCT
GTCCC]
##   [7]  72215    27   101      -  4  0  0 [CCCCTGCCCCACCCCCTACCCTGCCCC]
##   [8] 111482    29    99      -  4  0  0 [CCCCAGAGCCCCACACACCCCTCCGCCC
C]
##   [9]  29660    28    94      -  5  3  0 [CCCCCACCCCAACGCCCACCCTCCACCC]
##   ...     ...   ...   ...    ... ... ... ...
## [184] 114017    35    48      -  4  3  0 [CCCCAACAACGCGCCC...GGAGCACCGC
AACCC]
## [185] 121116    46    48      -  4  2  0 [CTGTGCCCGCCCCAGG...TCTGACCCTG
GCCCC]
## [186]   8193    34    47      -  4  1  1 [CATCCCAGCCCCGTCTCCACAAAGGCAGA
TCCCC]
## [187]  15962    37    47      +  4  3  0 [GGGGATGTGAGGGCTG...AGGGAGGCAT
TGAGG]
## [188]  22278    26    47      -  4  0  2 [CCCCATGTCTGCTCCAGCCCCTCCTC]
## [189]  59036    33    47      +  4  3  0 [GGCCCAGGAGAAAGAGGGGCTGGGCTCCT
GGGG]
## [190] 109797    20    47      -  3  1  0 [CTCCTCCCTGCTCCCTGCCC]
## [191] 115193    26    47      -  3  1  0 [CCCTCCTAGGTGGCCACCCACTGCCC]
## [192] 123830    26    47      +  3  1  0 [GGAGAGCCCAGCGGGGATGGGAAGGG]
```

7. Export all PQS into a GFF3-formatted file.

```
export(as(pqs, "GRanges"), "test.gff", version = "3")
```

The contents of the GFF3 file are as follows (the first three records only):

```
##gff-version 3
##source-version rtracklayer 1.46.0
##date 2019-10-29
chr1 pqsfinder G_quartet 778 806 85 + .
   nt=4;nb=1;nm=0;rl1=4;rl2=4;rl3=4;ll1=1;ll2=6;ll3=1
chr1 pqsfinder G_quartet 1071 1109 71 - .
   nt=5;nb=2;nm=1;rl1=5;rl2=8;rl3=5;ll1=6;ll2=4;ll3=1
```

8. Export all PQS into a FASTA format file.

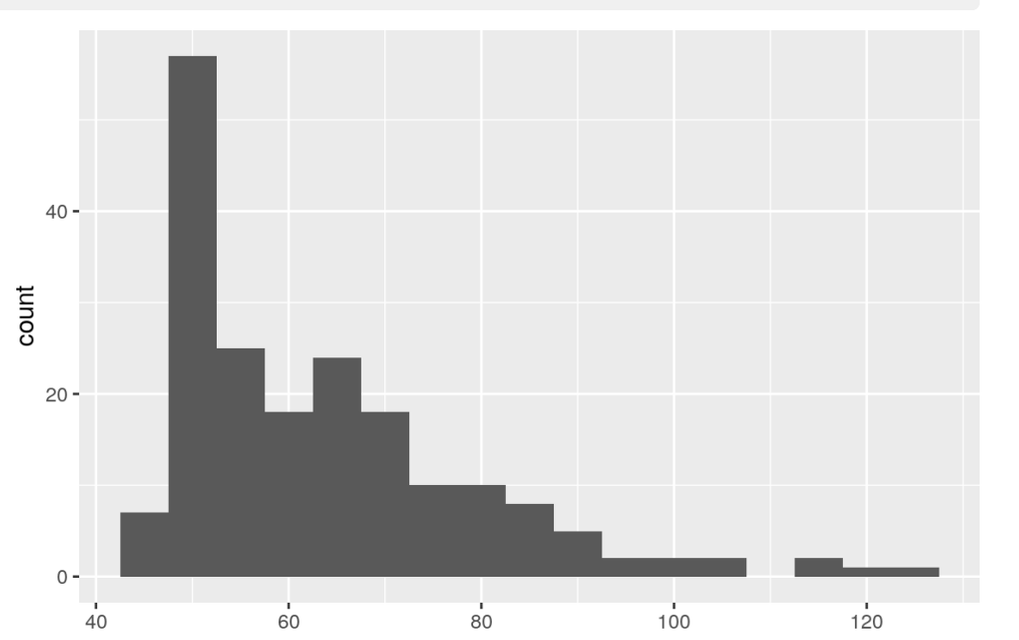
```
writeXStringSet(as(pqs, "DNAStringSet"), file = "test.fa", format = "fast
a")
```

The contents of the FASTA file are as follows (the first three records only):

```
>pqsfinder;G_quartet;start=778;end=806;strand=+;score=85;nt=4;nb=1;nm=0;rl
1=4;rl2=4;rl3=4;ll1=1;ll2=6;ll3=1;
GGGGAGGGGGAGCAAGGGGTGTAAGAGGG
>pqsfinder;G_quartet;start=1071;end=1109;strand=-;score=71;nt=5;nb=2;nm=1;
rl1=5;rl2=8;rl3=5;ll1=6;ll2=4;ll3=1;
CCCCCTCTAGTCCCAAACCTAAGCCCACACTCTGTCCCC
>pqsfinder;G_quartet;start=1846;end=1874;strand=-;score=54;nt=3;nb=0;nm=0;
rl1=3;rl2=3;rl3=3;ll1=8;ll2=1;ll3=8;
CCCTTCACCTTCCCTCCCTGTCGTCTCCC
```

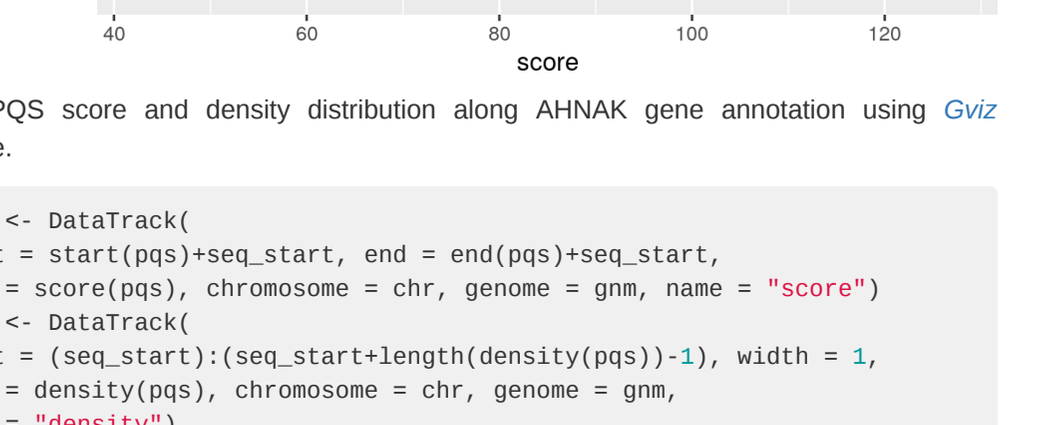
9. Show histogram for score distribution of detected PQS.

```
sf <- data.frame(score = score(pqs))
ggplot(sf) + geom_histogram(mapping = aes(x = score), binwidth = 5)
```



10. Show PQS score and density distribution along AHNAK gene annotation using *Gviz* package.

```
strack <- DataTrack(
  start = start(pqs)+seq_start, end = end(pqs)+seq_start,
  data = score(pqs), chromosome = chr, genome = gnm, name = "score")
dtrack <- DataTrack(
  start = (seq_start):(seq_start+length(density(pqs))-1), width = 1,
  data = density(pqs), chromosome = chr, genome = gnm,
  name = "density")
atrack <- GenomeAxisTrack()
suppressWarnings(plotTracks(c(gtrack, strack, dtrack, atrack), type = "h"
))
```



The stacked plot of the score and density distribution might help to assess the singularity of PQS. Higher density values indicates low-complexity regions (full of guanines), so it is expected to contain high-scoring PQS. On the other hand, a high-scoring PQS in low-density region might be an interesting target.

5 Customizing the detection algorithm

The underlying detection algorithm is almost fully customizable, it can even be set up to find fundamentally different types of G-quadruplexes. The very first option how to change the detection behavior is to tune scoring bonuses, penalizations and factors. Supported options are summarized in the table bellow:

Option name	Description
`tetrad_bonus`	G-tetrad bonus, regardless the tetrade contains mismatches or not.
`mismatch_penalty`	Penalization for a mismatch in tetrad.
`bulge_penalty`	Penalization for a bulge.
`bulge_len_factor`	Penalization factor of a bulge length.
`bulge_len_exponent`	Exponent of a bulge length.
`loop_mean_factor`	Penalization factor of a loop length mean.
`loop_mean_exponent`	Exponent of a loop length mean.

5.1 Customizing the scoring function

A more complicated way to influence the algorithm output is to implement a custom scoring function and pass it throught the `custom_scoring_fn` options. Before you start experimenting with this feature, please consider the fact that custom scoring function can **influence the overall algorithm performance very negatively**, particularly on long sequences. The best use case of this feature is rapid prototyping of novel scoring techniques, which can be later implemented efficiently, for example in the next version of this package. Thus, if you have any suggestions how to further improve the default scoring system (DSS), please let us know, we would highly appreciate that.

Basically, the custom scoring function should take the following 10 arguments:

- `subject` - input DNAString object,
- `score` - positive PQS score assigned by DSS, if enabled,
- `start` - PQS start position,
- `width` - PQS width,
- `loop_1` - loop #1 start position,
- `run_2` - run #2 start position,
- `loop_2` - loop #2 start position,
- `run_3` - run #3 start position,
- `loop_3` - loop #3 start position,
- `run_4` - run #4 start position.

The function will return a new score as a single integer value. Please note that if `use_default_scoring` is enabled, the custom scoring function is evaluated **after** the DSS but **only if** the DSS resulted in positive score (for performance reasons). On the other hand, when `use_default_scoring` is disabled, custom scoring function is evaluated on every PQS.

Example: Imagine you would like to assign a particular type of quadruplex a more favourable score. For example, you might want to reflect that G-quadruplexes with all loops containing just a single cytosine tend to be more stable than similar ones with different nucleotide at the same place. This can be easily implemented by the following custom scoring function:

```
c_loop_bonus <- function(subject, score, start, width, loop_1,
                         run_2, loop_2, run_3, loop_3, run_4) {
  l1 <- run_2 - loop_1
  l2 <- run_3 - loop_2
  l3 <- run_4 - loop_3
  if (l1 == l2 && l1 == l3 && subject[loop_1] == DNAString("C") &&
      subject[loop_1] == subject[loop_2] &&
      subject[loop_1] == subject[loop_3]) {
    score <- score + 20
  }
  return(score)
}
```

Without the custom scoring function, the two PQS found in the example sequence will have the same score.

```
seq <- DNAString("GGGCGGGCGGGCGGGAAAAAAAAAAAAAAGGGAGGGAGGGAGGG")
pqsfinder(seq)
```

```
##   PQS views on a 43-letter DNAString subject
## subject: GGGCGGGCGGGCGGGAAAAAAAAAAAAAAGGGAGGGAGGGAGGG
## quadruplexes:
##     start width score strand nt nb nm
## [1]     1    15    73      +  3  0  0 [GGGCGGGCGGGCGGG]
## [2]    29    15    73      +  3  0  0 [GGGAGGGAGGGAGGG]
```

However, if the custom scoring function presented above is applied, the two PQS are clearly distinguishable by score:

```
pqsfinder(seq, custom_scoring_fn = c_loop_bonus)
```

```
##   PQS views on a 43-letter DNAString subject
## subject: GGGCGGGCGGGCGGGAAAAAAAAAAAAAAGGGAGGGAGGGAGGG
## quadruplexes:
##     start width score strand nt nb nm
## [1]     1    15    93      +  3  0  0 [GGGCGGGCGGGCGGG]
## [2]    29    15    73      +  3  0  0 [GGGAGGGAGGGAGGG]
```

5.2 Complete replacement of the default scoring system

There might be use cases when it is undesirable to have the default scoring system (DSS) enabled. In this example we show how to change the detection algorithm behavior to find quite a different type of sequence motif - an interstrand G-quadruplex (isG4) (Kudlicki 2016). Unlike standard intramolecular G-quadruplex, isG4 can be defined by interleaving runs of guanines and cytosines respectively. Its canonical form can be described by a regular expression $G_nN_aC_nN_bG_nN_cC_n$.

To detect isG4s by the `pqsfinder` function, it is essential to change three options. At first, disable the DSS by setting `use_default_scoring` to `FALSE`. Second, specify a custom regular expression defining one run of the quadruplex by setting `run_re` to `G{3,6}|C{3,6}`. The last step is to define a custom scoring function validating each PQS:

```
isG4 <- function(subject, score, start, width, loop_1,
                 run_2, loop_2, run_3, loop_3, run_4) {
  r1 <- loop_1 - start
  r2 <- loop_2 - run_2
  r3 <- loop_3 - run_3
  r4 <- start + width - run_4

  if (!(r1 == r2 && r1 == r3 && r1 == r4))
    return(0)

  run_1_s <- subject[start:start+r1-1]
  run_2_s <- subject[run_2:run_2+r2-1]
  run_3_s <- subject[run_3:run_3+r3-1]
  run_4_s <- subject[run_4:run_4+r4-1]

  if (length(grep("^G+$", run_1_s)) && length(grep("^C+$", run_2_s)) &&
      length(grep("^G+$", run_3_s)) && length(grep("^C+$", run_4_s)))
    return(r1 * 20)
  else
    return(0)
}
```

Let's see how it all works together:

```
pqsfinder(DNAString("AAAAGGGATCCCTAAGGGGTCCC"), strand = "+",
          use_default_scoring = FALSE, run_re = "G{3,6}|C{3,6}",
          custom_scoring_fn = isG4)
```

```
##   PQS views on a 23-letter DNAString subject
## subject: AAAAGGGATCCCTAAGGGGTCCC
## quadruplexes:
##     start width score strand nt nb nm
## [1]     5    19    60      +  0  0  0 [GGGATCCCTAAGGGGTCCC]
```

6 Session info

Here is the output of `sessionInfo()` on the system on which this document was compiled:

```
## R version 3.6.1 (2019-07-05)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 18.04.3 LTS
##
## Matrix products: default
## BLAS:   /home/biocbuild/bbs-3.10-bioc/R/lib/libRblas.so
## LAPACK: /home/biocbuild/bbs-3.10-bioc/R/lib/libRlapack.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=C
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
##  [1] grid      stats4    parallel  stats     graphics  grDevices utils
##  [8] datasets  methods   base
##
## other attached packages:
##  [1] ggplot2_3.2.1                     BSgenome.Hsapiens.UCSC.hg38_1.4.1
##  [3] BSgenome_1.54.0                   rtracklayer_1.46.0
##  [5] Gviz_1.30.0                       GenomicRanges_1.38.0
##  [7] GenomeInfoDb_1.22.0               pqsfinder_2.2.0
##  [9] Biostrings_2.54.0                 XVector_0.26.0
## [11] IRanges_2.20.0                    S4Vectors_0.24.0
## [13] BiocGenerics_0.32.0               BiocStyle_2.14.0
##
## loaded via a namespace (and not attached):
##  [1] ProtGenerics_1.18.0         bitops_1.0-6
##  [3] matrixStats_0.55.0          bit64_0.9-7
##  [5] RColorBrewer_1.1-2          progress_1.2.2
##  [7] httr_1.4.1                  tools_3.6.1
##  [9] backports_1.1.5             R6_2.4.0
## [11] rpart_4.1-15                Hmisc_4.2-0
## [13] DBI_1.0.0                   lazyeval_0.2.2
## [15] colorspace_1.4-1            nnet_7.3-12
## [17] withr_2.1.2                 tidyselect_0.2.5
## [19] gridExtra_2.3               prettyunits_1.0.2
## [21] bit_1.1-14                  curl_4.2
## [23] compiler_3.6.1              Biobase_2.46.0
## [25] htmlTable_1.13.2            DelayedArray_0.12.0
## [27] labeling_0.3                bookdown_0.14
## [29] scales_1.0.0                checkmate_1.9.4
## [31] askpass_1.1                 rappdirs_0.3.1
## [33] stringr_1.4.0               digest_0.6.22
## [35] Rsamtools_2.2.0             foreign_0.8-72
## [37] rmarkdown_1.16              dichromat_2.0-0
## [39] base64enc_0.1-3             pkgconfig_2.0.3
## [41] htmltools_0.4.0             ensembldb_2.10.0
## [43] dbplyr_1.4.2                htmlwidgets_1.5.1
## [45] rlang_0.4.1                 rstudioapi_0.10
## [47] RSQLite_2.1.2               BiocParallel_1.20.0
## [49] acepack_1.4.1               dplyr_0.8.3
## [51] VariantAnnotation_1.32.0    RCurl_1.95-4.12
## [53] magrittr_1.5                GenomeInfoDbData_1.2.2
## [55] Formula_1.2-3               Matrix_1.2-17
## [57] Rcpp_1.0.2                  munsell_0.5.0
## [59] stringi_1.4.3               yaml_2.2.0
## [61] SummarizedExperiment_1.16.0 zlibbioc_1.32.0
## [63] BiocFileCache_1.10.0        blob_1.2.0
## [65] crayon_1.3.4                lattice_0.20-38
## [67] splines_3.6.1               GenomicFeatures_1.38.0
## [69] hms_0.5.1                   zeallot_0.1.0
## [71] knitr_1.25                  pillar_1.4.2
## [73] biomaRt_2.42.0              XML_3.98-1.20
## [75] glue_1.3.1                  evaluate_0.14
## [77] biovizBase_1.34.0           latticeExtra_0.6-28
## [79] data.table_1.12.6           BiocManager_1.30.9
## [81] vctrs_0.2.0                 gtable_0.3.0
## [83] openssl_1.4.1               purrr_0.3.3
## [85] assertthat_0.2.1            xfun_0.10
## [87] AnnotationFilter_1.10.0     survival_2.44-1.1
## [89] tibble_2.1.3                GenomicAlignments_1.22.0
## [91] AnnotationDbi_1.48.0        memoise_1.1.0
## [93] cluster_2.1.0
```

References

Chambers, Vicki S, Giovanni Marsico, Jonathan M Boutell, Marco Di Antonio, Geoffrey P Smith, and Shankar Balasubramanian. 2015. "High-Throughput Sequencing of DNA G-Quadruplex Structures in the Human Genome." *Nature Biotechnology* 33 (8). Springer Nature:877–81. https://doi.org/10.1038/nbt.3295.

Kejnovsky, Eduard, Viktor Tokan, and Matej Lexa. 2015. "Transposable Elements and G-Quadruplexes." *Chromosome Research* 23. https://doi.org/10.1007/s10577-015-9491-7.

Kudlicki, Andrzej S. 2016. "G-Quadruplexes Involving Both Strands of Genomic Dna Are Highly Abundant and Colocalize with Functional Sites in the Human Genome." *PLoS ONE* 11 (1). https://doi.org/10.1371/journal.pone.0146174.

Maizels, Nancy, and Lucas T. Gray. 2013. "The G4 Genome." *PLoS Genet* 9 (4). Public Library of Science:1–6. https://doi.org/10.1371/journal.pgen.1003468.