Searching Biological Sequences for Research

Erik S. Wright

October 31, 2025

Contents

1 Introduction

Sequence searching is an essential part of biology research. The word *research* even originates from a word in Old French meaning 'to search'. Yet, the sheer amount of biological sequences to comb through can make (re)search feel like finding a needle in a haystack. To avoid heading out on a wild goose chase, it's important to master the ins and outs of searching. The goal of this vignette is to help you leave no stone unturned as you scout out homologous sequences. Mixed metaphors aside, understanding how to properly use DECIPHER's search functions is critical for optimizing performance. The search functions' versatility makes them highly customizable, but this also places the onus of understanding on the user. This vignette is intended to bring you up to speed on how to best apply the search functions in your research.

2 Getting Started

Fast sequence searching is performed by breaking the query and target sequences into k-mers. Behind the scenes, the search function has to rapidly find significant paths among many k-mer matches shared by both sequences. This process is especially difficult in the presence of many spurious matches and repeats (Fig. 1). The `IndexSeqs` and

SearchIndex functions introduced in this vignette employ many techniques to increase the accuracy and speed of search, such as masking problematic regions. The examples below illustrate how to best apply these techniques to solve different search problems, including searching translated sequences, extending k-mer matches, aligning search hits, and mapping reads.

2.1 Startup

To get started we need to load the DECIPHER package, which automatically loads a few other required packages.

```
> library(DECIPHER)
```

Help for a function can be accessed through:

```
> ? SearchIndex
```

Once DECIPHER is installed, the code in this tutorial can be obtained via:

```
> browseVignettes("DECIPHER")
```

2.2 Gathering the evidence

There are umpteen reasons to search through biological sequences. For the purposes of this vignette, we are going to focus on finding homologous proteins in a genome. In this case, our pattern (query) is the protein sequence and the subject (target) is a 6-frame translation of the genome. Feel free to follow along with your own (nucleotide or protein) sequences or use those in the vignette:

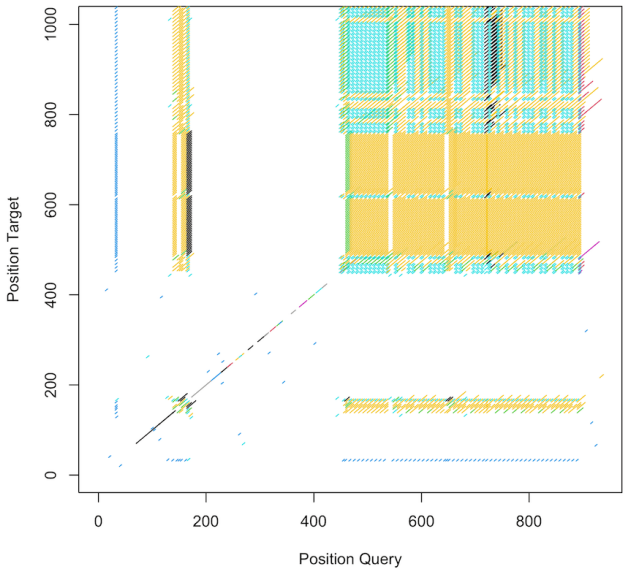

Figure 1: K-mer matches between a query and target sequence colored by their length. The presence of repeats can be seen in rectangular blocks with many k-mer matches.

```
> # specify the path to your file of pattern (query) sequences:
> fas1 <- "<<path to pattern FASTA file>>"
> # OR use the example protein sequences:
> fas1 <- system.file("extdata",
          "PlanctobacteriaNamedGenes.fas.gz",
          package="DECIPHER")
> # read the sequences into memory
> pattern <- readAAStringSet(fas1)
> pattern
AAStringSet object of length 2497:
       width seq                                                names
   [1]   227 MAGPKHVLLVSEHWDLFFQTKE...VGYLFSDDGDKKFSQQDTKLS A0A0H3MDW1|Root;N...
   [2]   394 MKRNPHFVSLTKNYLFADLQKR...GKREDILAACERLQMAPALQS O84395|Root;2;6;1...
   [3]   195 MAYGTRYPTLAFHTGGIGESDD...GFCLTALGFLNFENAEPAKVN Q9Z6M7|Root;4;1;1...
   [4]   437 MMLRGVHRIFKCFYDVVLVCAF...TASFDRTWRALKSYIPLYKNS Q46222|Root;2;4;9...
   [5]   539 MSFKSIFLTGGVVSSLGKGLTA...FIEFIRAAKAYSLEKANHEHR Q59321|Root;6;3;4...
   ...   ... ...
[2493]  1038 MFEEVLQESFDEREKKVLKFWQ...EGTDWDLNGEPTKIIIKKSEY Q6MDY1|Root;6;1;1...
[2494]   102 MVQIVSQDNFADSIASGLVLVD...VERSVGLKDKDSLVKLISKHQ Q9PJK3|Root;NoEC;...
```

```
[2495]   224 MKPQDLKLPYFWEDRCPKIENH...NLWRSKGEKIFCTEFVKRVGI Q9PL91|Root;2;1;1...
[2496]   427 MLRRLFVSTFLIFGMVSLYAKD...KIVIGLGEKRFPSWGGFPNNQ Q256H8|Root;NoEC;...
[2497]   344 MLTLGLESSCDETACALVDAKG...GIHPCARYHWESISASLSPLP Q822Y4|Root;2;3;1...
```

Protein search is more accurate than nucleotide search, so we are going to import a genome and perform 6-frame translation to get the subject sequences. Feel free to carry on without translating the sequences if you are searching nucleotides or otherwise would prefer to skip translation. Note that `SearchIndex` only searches the nucleotides in the direction they are provided, so if you desire to search both strands then you will need to combine with the `reverseComplement` as shown below.

```
> # specify the path to your file of subject (target) sequences:
> fas2 <- "<<path to subject FASTA file>>"
> # OR use the example subject genome:
> fas2 <- system.file("extdata",
          "Chlamydia_trachomatis_NC_000117.fas.gz",
          package="DECIPHER")
> # read the sequences into memory
> genome <- readDNAStringSet(fas2)
> genome
DNAStringSet object of length 1:
      width seq                                               names
[1] 1042519 GCGGCCGCCCGGGAAATTGCTA...GTTGGCTGGCCCTGACGGGGTA NC_000117.1 Chlam...
> genome <- c(genome, reverseComplement(genome)) # two strands
> subject <- subseq(rep(genome, each=3), rep(1:3, 2)) # six frames
> subject <- suppressWarnings(translate(subject)) # 6-frame translation
> subject
AAStringSet object of length 6:
     width seq                                               names
[1] 347506 AAAREIAKRWEQRVRDLQDKGAA...GCVHK*VRGSFRSEQVGWP*RG NC_000117.1 Chlam...
[2] 347506 RPPGKLLKDGSKELEIYKIKVLH...AAYTSECADHLEANKLAGPDGV NC_000117.1 Chlam...
[3] 347505 GRPGNC*KMGAKS*RSTR*RCCT...WLRTQVSARII*KRTSWLALTG NC_000117.1 Chlam...
[4] 347506 YPVRASQLVRF*MIRALTCVRSH...QHLYLVDL*LFAPIF*QFPGRP NC_000117.1 Chlam...
[5] 347506 TPSGPANLFASK*SAHSLVYAAI...STFIL*ISNSLLPSFSNFPGGR NC_000117.1 Chlam...
[6] 347505 PRQGQPTCSLLNDPRTHLCTQPS...AAPLSCRSLTLCSHLLAISRAA NC_000117.1 Chlam...
```

3 Searching for hits between pattern and subject sequences

Once the sequences are imported, we need to build an *InvertedIndex* object from the `subject` sequences. We can accomplish this with `IndexSeqs` by specifying the k-mer length (*K*). If you don't know what value to use for *K*, then you can specify *sensitivity*, *percentIdentity*, and *patternLength* in lieu of *K*. Here, we want to ensure we find 99% (0.99) of sequences with at least 70% identity to a pattern with 300 or more residues. Note that *sensitivity* is defined as a fraction, whereas *percentIdentity* is defined as a percentage.

```
> index <- IndexSeqs(subject,
          sensitivity=0.99,
```

```
         percentIdentity=70,
         patternLength=300,
         processors=1)
================================================================================

Time difference of 0.72 secs
> index
  An InvertedIndex built with:
   * Amino acid sequences: 6
   * Total k-mers: 1,228,562
   * Alphabet: A, C, DE, FWY, G, H, ILMV, N, P, Q, RK, ST
   * K-mer size: 5
   * Step size: 1
```

Printing the `index` shows that we created an *InvertedIndex* object containing over 1 million 5-mers in a reduced amino acid alphabet with 12 symbols. Before we can find homologous hits to our protein sequences with `SearchIndex`, we must decide how many hits we desire. The default is to return the best hit (above *minScore*) per subject sequence, but this may return too many hits. Instead we could limit the number of hits by specifying a positive value for `perPatternLimit` or `perSubjectLimit` (unlimited is `0`). Note that we do not need to specify a value for *minScore*, because it is automatically set based on the size of the *InvertedIndex*.

```
> hits <- SearchIndex(pattern,
         index,
         perPatternLimit=100,
         processors=1)
================================================================================

Time difference of 7.48 secs
> dim(hits)
[1] 1917    4
> head(hits)
  Pattern Subject     Score     Position
1       1       6  468.4091 1, 12, 1....
2       2       2  901.9045 1, 394, ....
3       3       3  390.1394 3, 67, 1....
4       4       6  278.6762 18, 22, ....
5       5       3 1224.8793 1, 539, ....
6       6       6 1206.7279 1, 538, ....
```

The result of our search is a *data.frame* with four columns: Pattern (index in `pattern`), Subject (index in `subject`), Score, and Position (of k-mer matches). The Position column is optional and can be disabled by setting *scoreOnly* to `TRUE`. We can take a closer look at the number of hits per protein, their scores, and locations (Fig. 2):

```
> layout(matrix(1:4, nrow=2))
> hist(hits$Score,
         breaks=100, xlab="Score", main="Distribution of scores")
> plot(NA, xlim=c(0, max(width(subject))), ylim=c(1, 6),
         xlab="Genome position", ylab="Genome frame",
         main="Location of k-mer matches")
> segments(sapply(hits$Position, `[`, i=3), # third row
         hits$Subject,
         sapply(hits$Position, `[`, i=4), # fourth row
         hits$Subject)
> plot(hits$Score,
         sapply(hits$Position, function(x) sum(x[2,] - x[1,] + 1)),
         xlab="Score", ylab="Sum of k-mer matches",
         main="Matches versus score", log="xy")
> plot(table(tabulate(hits$Pattern, nbins=length(pattern))),
         xlab="Hits per pattern sequence", ylab="Frequency",
         main="Number of hits per query")
```

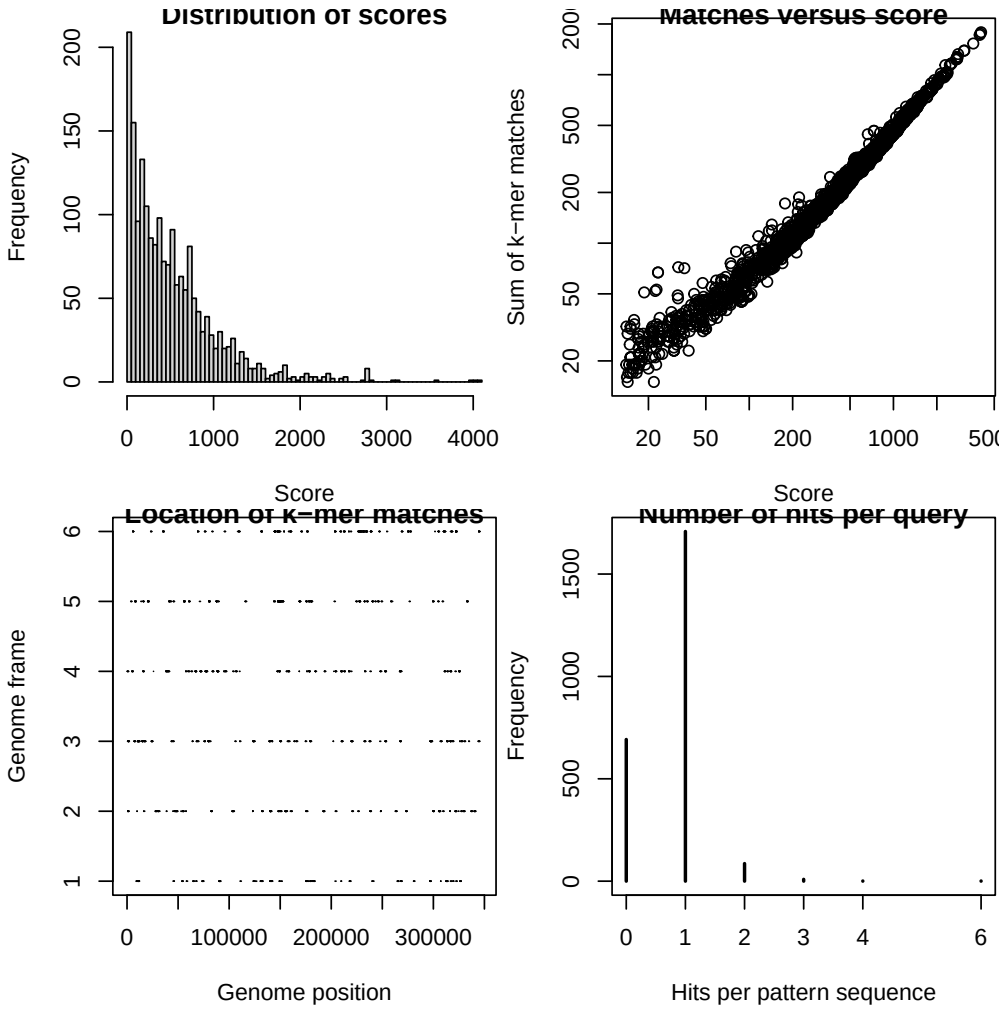

Figure 2: Summary of hits found between a set of proteins and the genome's 6-frame translation.

The calculated *Score* for each search hit is defined by the negative log-odds of observing the hit by chance. We see that most scores were near zero, but there were many high scoring hits. Hits tended to be clustered along specific frames of the genome, with some genome regions devoid of hits. Also, most pattern (protein) sequences were found at zero or one location in the genome. As expected, a hit's score is correlated with the length of k-mer matches, although greater separation between matches lowers the score. One protein was found many times more than all the others. We can easily figure out which protein was found the most times:

```
> w <- which.max(tabulate(hits$Pattern))
> hits[hits$Pattern == w,]
    Pattern Subject    Score     Position
823    1065       1 25.27856 43, 51, ....
824    1065       2 45.04873 27, 31, ....
825    1065       3 46.61342 43, 50, ....
826    1065       4 47.26039 27, 31, ....
827    1065       5 18.74354 43, 50, ....
828    1065       6 30.13226 153, 158....
> names(pattern)[w]
[1] "Q9Z8Q8|Root;7;4;2;11;metN"
```

Likely these hits are to multiple paralogous genes on the genome, as can be seen by the wide distribution of scores.

4 Aligning the search hits between pattern and subject

So far, we've identified the location and score of search hits without alignment. Aligning the hits would provide us with their local start and stop boundaries, percent identity, and the locations of any insertions or deletions. Thankfully, alignment is elementary once we've completed our search.

```
> aligned <- AlignPairs(pattern=pattern,
          subject=subject,
          pairs=hits,
          processors=1)
================================================================================

Time difference of 0.44 secs
> head(aligned)
  Pattern PatternStart PatternEnd Subject SubjectStart SubjectEnd Matches
1       1            1        227       6       108250     108476     222
2       2            1        394       2       148163     148556     394
3       3            1        195       3       141952     142146     173
4       4           13        435       6       268926     269346     222
5       5            1        539       3        68143      68681     539
6       6            1        544       6       304850     305393     544
  Mismatches AlignmentLength    Score PatternGapPosition PatternGapLength
1          5             227 1350.159
2          0             394 2432.598
3         22             195 1153.484
4        199             423 1485.138
5          0             539 3385.075
6          0             544 3125.491
```

```
  SubjectGapPosition SubjectGapLength
1
2
3
4           218, 372             1, 1
5
6
```

The `AlignPairs` function returns a *data.frame* containing the Pattern (i.e., `pattern` index), Subject (i.e., `subject` index), their start and end positions, the number of matched and mismatched positions, the alignment length and its score, as well as the location of any gaps in the *pattern* or *subject*. We can use this information to calculate a percent identity, which can be defined a couple of different ways (Fig. 3).

```
> PID1 <- aligned$Matches/(aligned$Matches + aligned$Mismatches)
> PID2 <- aligned$Matches/aligned$AlignmentLength
> layout(matrix(1:4, ncol=2))
> plot(hits$Score, PID2,
         xlab="Hit score",
         ylab="Matches / (Aligned length)")
> plot(hits$Score, aligned$Score,
         xlab="Hit score",
         ylab="Aligned score")
> plot(aligned$Score, PID1,
         xlab="Aligned score",
         ylab="Matches / (Matches + Mismatches)")
> plot(PID1, PID2,
         xlab="Matches / (Matches + Mismatches)",
         ylab="Matches / (Aligned length)")
```

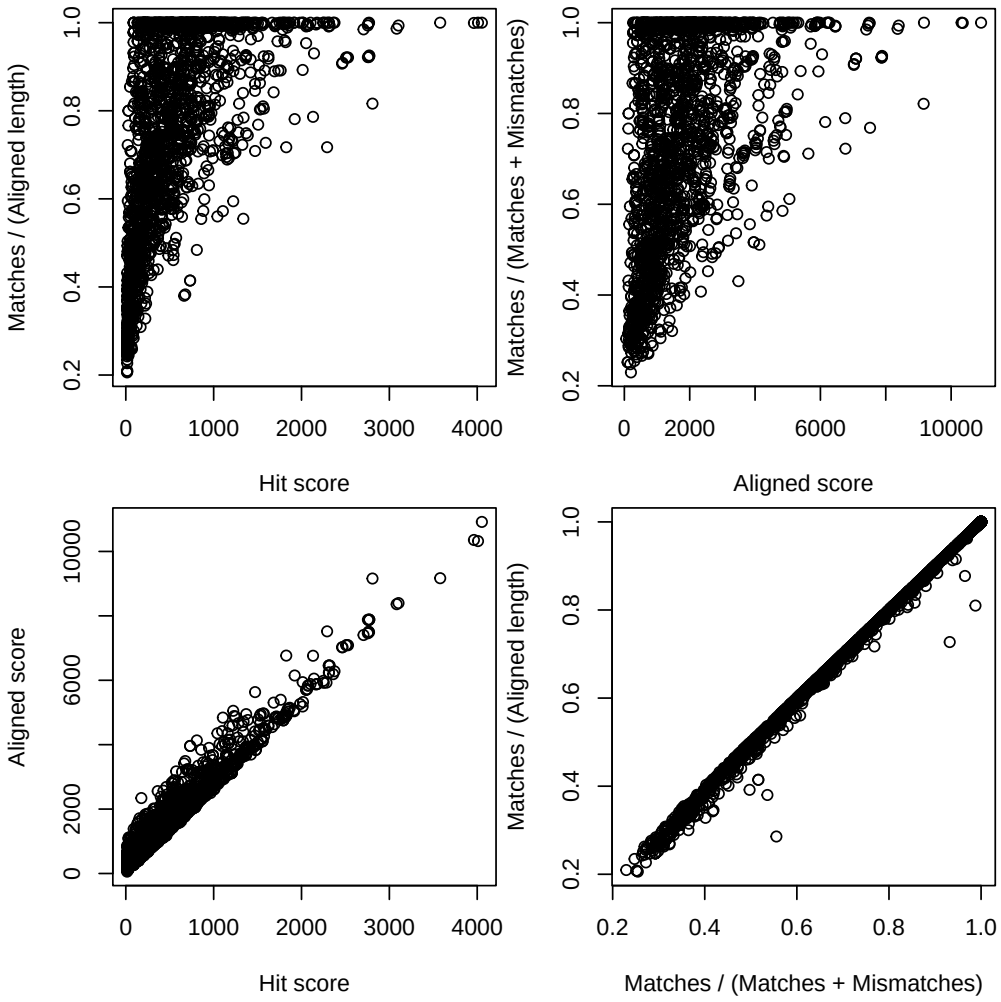

Figure 3: Scatterplots of different scores and methods of formulating percent identity.

By default, `AlignPairs` gives us everything we need to align the sequences except the alignments themselves. If needed, we can easily request the alignments be included in the output.

```
> alignments <- AlignPairs(pattern=pattern,
          subject=subject,
          pairs=hits,
          type="both",
          processors=1)
================================================================================

Time difference of 0.45 secs
> patterns <- alignments[[2]]
> subjects <- alignments[[3]]
> c(patterns[1], subjects[1]) # view the first pairwise alignment
AAStringSet object of length 2:
    width seq
[1]   227 MAGPKHVLLVSEHWDLFFQTKELLNPEEYRCTIG...GAYGSRIVTLRGVGYLFSDDGDKKFSQQDTKLS
[2]   227 MAGPKHVLLVSENWDLFFQTKELLNPEEYRCTIG...GAYGSRIVTLRGVGYLFSDDGDKKFSQQDTKLS
```

5 Controlling for false discoveries

`SearchIndex` returns hits with scores above *minScore*. However, it is often useful to compute an expect value (E-value) representing the number of times we expect to see a hit at least as high scoring in a database of the same size. A lesser known fact is that E-values are a function of the substitution matrix, *gapOpening* penalty, *gapExtension* penalty, and other search parameters, so E-values must be empirically determined.

Parameters for determining E-values are typically pre-estimated on an independent database. However, it is intuitive that the composition of the target database might have an effect on the E-values. For this reason, reported E-values are often biased and miscalibrated across different programs [1]. It is therefore preferable to develop an approach for calibrating E-values on the specific target database that is being searched. Below are two alternative approaches to controlling for false discoveries in sequence search.

5.1 Option 1: Calibrating an expect value (E-value) from hit scores

There are two straightforward ways to calibrate E-values: (1) create an equivalent database of random sequences with matched composition to the input, or (2) search for the reverse of the sequences under the assumption that reverse hits are unexpected (i.e., false positives). The second approach is more conservative, because we will find more hits than expected if its underlying assumption is not true. Here, we will try the second approach:

```
> revhits <- SearchIndex(reverse(pattern), # reverse the query
          index, # keep the same target database
          minScore=10, # set low to get many hits
          processors=1)
================================================================================

Time difference of 7.05 secs
> dim(revhits)
[1] 14  4
```

Next, our goal is to fit the distribution of background scores (i.e., reverse hits), which is reasonably well-modeled by an exponential distribution. We will bin the reverse hits' scores into intervals of one score unit between

`10` and `100`. Then we will use the fact that the integral of $e^{-rate*x}$ (with respect to x) is $e^{-rate*x}$ to estimate the number of background hits at each score cutoff. Note how there is an outlying point that violated the assumption reversed sequences should not have strong hits (Fig. 4). We can use the *sum of absolute error* (L1 norm) rather than the *sum of squared error* (L2 norm) to make the fit more robust to outliers. We will perform the fit in log-space to emphasize points across many orders of magnitude.

```
> X <- 10:100 # score bins
> Y <- tabulate(.bincode(revhits$Score, X), length(X) - 1)
> Y <- Y/length(pattern) # average per query
> w <- which(Y > 0) # needed to fit in log-space
> plot(X[w], Y[w],
         log="y",
         xlab="Score",
         ylab="Average false positives per query")
> fit <- function(rate) # integrate from bin start to end
         sum(abs((log((exp(-X[w]*rate) -
                 exp(-X[w + 1]*rate))*length(subject)) -
                 log(Y[w]))))
> o <- optimize(fit, c(0.01, 2)) # optimize rate
> lines(X[-length(X)], (exp(-X[-length(X)]*o$minimum) -
         exp(-(X[-1])*o$minimum))*length(subject))
> rate <- o$minimum
> print(rate)
[1] 0.5117453
```

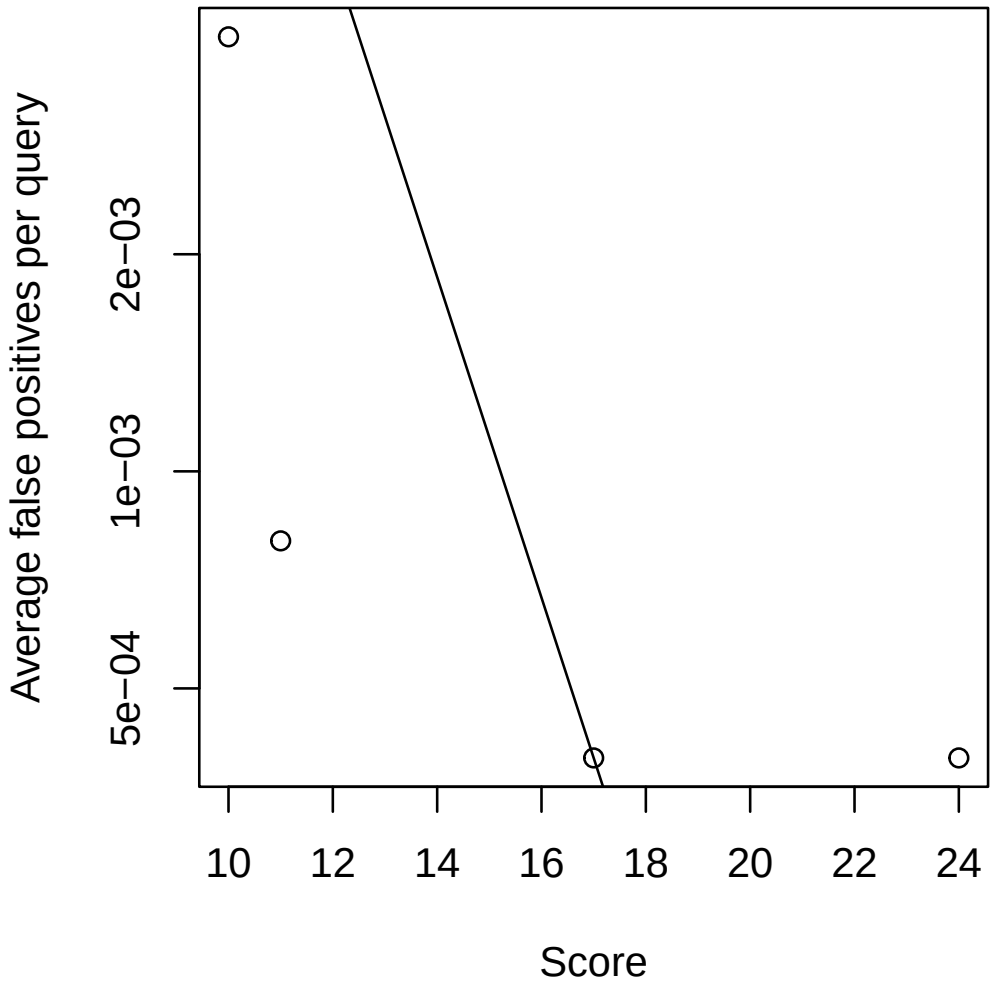

Figure 4: Fitting an exponential distribution to the score background.

Now that we've optimized the *rate* parameter, it is feasible to convert our original scores into E-values. We are interested in the number of false positive hits expected across all queries at every value of `Score`. This differs from the standard definition of E-value, which is defined on a per query basis. However, since we are performing multiple queries it is preferable to apply a multiple testing correction for the number of searches. We can convert our original scores to E-values, as well as define score thresholds for a given number of acceptable false positives across all `pattern` sequences:

```
> # convert each Score to an E-value
> Evalue <- exp(-rate*hits$Score)*length(subject)*length(pattern)
> # determine minimum Score for up to 1 false positive hit expected
> log(1/length(subject)/length(pattern))/-rate
[1] 18.78787
```

As can be seen, for this particular combination of dataset and parameters, a score threshold of `22` is sufficient to only permit one combined false positive across all queries. Since E-value is a function of the dataset's size and the specific search parameters, you should calibrate the E-value for each set of searches performed. Once you have calibrated the *rate*, it is straightforward to find only those hits that are statistically significant:

```
> # determine minimum Score for 0.05 (total) false positive hits expected
> threshold <- log(0.05/length(subject)/length(pattern))/-rate
> hits <- hits[hits$Score > threshold,]
> dim(hits)
[1] 1816    4
```

5.2 Option 2: The knockoff approach for limiting false discoveries

An alternative and simpler approach to limiting false discoveries is to use the results on knockoffs to limit the false discovery rate. To use the reverse hits for this purpose, we will rank the combined set of results and allow any hits above a pre-determined false discovery rate.

```
> FDR <- 0.001 # maximum allowed false discovery rate
> N <- nrow(hits)
> ranking <- order(c(hits$Score, revhits$Score), decreasing=TRUE)
> ranking <- ranking[cumprod(cumsum(ranking > N) <= seq_along(ranking)*FDR) == 1L]
> hits <- hits[sort(ranking[ranking <= N]),] # filtered hits
> nrow(hits) # number of search hits after controlling FDR
[1] 1816
```

6 Maximizing search sensitivity to find distant hits

We've already employed some strategies to improve search sensitivity: choosing a small value for k-mer length and step size, searching amino acids rather than nucleotides, and masking low complexity regions and repeats. Although k-mer search is very fast, sometimes k-mers alone are insufficient to find distant homologs. In these cases, search sensitivity can be improved by providing the `subject` (target) sequences, which causes `SearchIndex` to extend k-mer matches to their left and right. This is as simple as adding a single argument:

```
> # include the target sequences to increase search sensitivity
> hits <- SearchIndex(pattern,
          index,
          subject, # optional parameter
          perPatternLimit=100,
          processors=1)
```

```
================================================================================

Time difference of 332.48 secs
> dim(hits)
[1] 2760    4
> head(hits)
  Pattern Subject     Score     Position
1       1       6  636.2766 1, 13, 1....
2       2       2 1156.7987 1, 394, ....
3       3       3  530.2673 1, 71, 1....
4       4       6  589.2361 13, 42, ....
5       5       3 1607.9917 1, 539, ....
6       6       6 1506.3808 1, 544, ....
```

We can see that search took longer when providing `subject` sequences, but the number of hits also increased. The *dropScore* parameter, which controls the degree of extension, can be adjusted to balance sensitivity and speed. In this manner, high sensitivity can be achieved by providing `subject` sequences in conjunction with a low value of k-mer length and *dropScore*.

7 Mapping long reads to a genome

Read alignment is one of the most common bioinformatics tasks. The higher error rates of long read sequences inspired new mapping approaches with greater error tolerance than traditional short read mapping algorithms. DECIPHER's search functions are well suited to locating the position of significant matches in a genome. As an example, we will try mapping error-prone long reads to a genome. There are two ways to search both strands with nucleotide sequences: (1) search the forward and reverse complement of the query (pattern) sequences with *SearchIndex*, or (2) build an inverted index from the forward and reverse complement of the target (subject) sequences with *IndexSeqs*. The first approach will usually have a lower memory footprint, but the second approach only requires a single call to `SearchIndex`. Long reads could overlap with repeat-rich or low-complexity regions of the genome, so we will disable masking of those regions.

```
> index <- IndexSeqs(genome,
          K=11,
          maskRepeats=FALSE,
          maskLCRs=FALSE,
          processors=1)
================================================================================

Time difference of 0.76 secs
> index
  An InvertedIndex built with:
   * Nucleotide sequences: 2
   * Total k-mers: 2,085,018
   * Alphabet: A, C, G, T
   * K-mer size: 11
   * Step size: 1
```

Here, we will use a set of simulated long reads with known mapping positions on the genome. We can see that the header for each sequence contains the strand and starting location of the read on the genome.

```
> fas3 <- system.file("extdata",
          "Simulated_ONT_Long_Reads.fas.gz",
          package="DECIPHER")
> reads <- readDNAStringSet(fas3)
> head(names(reads))
[1] "186478"  "283731"  "1012724" "978503"  "304154"  "346391"
```

In this case, we are only interested in unambiguous mappings, so we will compare the top two hits for each sequence to discard multi-mapped reads. We can subtract the score for the second best hit from the score for the top hit to approximate how well a read mapped to a single location on the genome. Also, we will disable masking of the reads in the same manner as we did for the genome above.

```
> maps <- SearchIndex(reads,
          index,
          perPatternLimit=2, # two hits per read
          perSubjectLimit=0, # unlimited
          maskRepeats=FALSE,
          maskLCRs=FALSE,
          processors=1)
================================================================================

Time difference of 0.29 secs
> dim(maps)
[1] 59  4
> head(maps)
  Pattern Subject     Score     Position
1       1       1  3129.880 7, 19, 1....
2       2       1  5096.578 14, 30, ....
3       3       1  4568.316 1, 18, 1....
4       4       1  5411.576 14, 25, ....
5       5       1 15628.914 20, 33, ....
6       6       1  1104.886 45, 58, ....
> # when > 1 hit, subtract 2nd highest score from top hit
> o <- order(maps$Pattern, maps$Score, decreasing=TRUE)
> w <- which(duplicated(maps$Pattern[o]))
> maps$Score[o[w - 1]] <- maps$Score[o[w - 1]] - maps$Score[o[w]]
> maps <- maps[-o[w],] # remove 2nd best hits
> nrow(maps) == length(reads)
[1] TRUE
```

It is possible to see that all the simulated reads mapped to the first subject sequence, which was expected because only reads on the forward strand were simulated. Now, it is possible to compare the predicted mapping locations with the expected mapping locations. The "Position" column contains matrices with four rows giving the start and end positions of anchors in the pattern and subject, respectively. The start of the predicted mapping location can be calculated from the first column of the matrix and compared to the actual mapping location in the sequences header. Since the genome is over a million nucleotides long, any read that maps near the expected position is a positive result.

```
> pos <- sapply(maps$Position, function(x) x[3] - x[1] + 1)
> offset <- abs(pos - as.integer(names(reads)))
> table(offset)
```

```
offset
    0     1     2     3     4     5     6    18 22054
    9    19    14     4     3     2     2     1     1
> maps$Score[which.max(offset)]
[1] 0
```

All reads mapped to locations nearby the correct location except one, which was far from the actual mapping location. This read had a mapping score of zero, because this read mapped equally well to multiple locations on the genome. Our strategy of subtracting second best scores worked as intended to eliminate this multi-mapped read. In practice, we would see reads mapped to both strands of the genome and we could use `AlignPairs` to obtain more information if desired.

8 Recommended search settings

It is best to tune function parameters for your particular search task. Most default parameters can be left alone. Shown below are recommended alternative settings for common use cases to serve as a starting point:

Function(s)	Parameter	Long read mapping	Short read mapping	Nucleotide search	Protein search
`IndexSeqs`	*K*	11 (10 to 12)	12 (11 to 13)	9 (8 to 10)	5 (4 to 6)
`IndexSeqs`	*maskRepeats*	`FALSE`	`FALSE`		
`IndexSeqs`	*maskLCRs*	`FALSE`	`FALSE`		
`SearchIndex`	*subject*			supply	supply
`SearchIndex`	*perSubjectLimit*	`2`	`2`	`1` or more	`1` or more
`SearchIndex`	*perPatternLimit*	`2`	`2`	typically » `1`	typically » `1`
All	*processors*	`NULL`	`NULL`	`NULL`	`NULL`

Note that none of these functions removes duplicate sequences. If your *pattern* or *subject* contain many similar or identical sequences, reducing redundancy with `unique` or `Clusterize` may improve speed.

9 Session Information

All of the output in this vignette was produced under the following conditions:

- R Under development (unstable) (2025-10-20 r88955), `x86_64-pc-linux-gnu`
- Running under: `Ubuntu 24.04.3 LTS`
- Matrix products: default
- BLAS: `/home/biocbuild/bbs-3.23-bioc/R/lib/libRblas.so`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0`
- Base packages: base, datasets, grDevices, graphics, methods, stats, stats4, utils
- Other packages: BiocGenerics 0.57.0, Biostrings 2.79.1, DECIPHER 3.7.0, IRanges 2.45.0, S4Vectors 0.49.0, Seqinfo 1.1.0, XVector 0.51.0, generics 0.1.4
- Loaded via a namespace (and not attached): DBI 1.2.3, KernSmooth 2.23-26, compiler 4.6.0, crayon 1.5.3, tools 4.6.0

References

[1] Lu, Y. Y., Noble, W. S., & Keich, U. A BLAST from the past: revisiting blastp's *E*-value. Bioinformatics, 40(12), btae729. doi:10.1093/bioinformatics/btae729, 2024.