Sequence manipulation and scanning

Benjamin Jean-Marie Tremblay*

5 May 2026

Abstract

Sequences stored as XStringSet objects (from the Biostrings package) can be used by several functions in the universalmotif package. These functions are demonstrated here and fall into two categories: sequence manipulation and motif scanning. Sequences can be generated, shuffled, and background frequencies of any order calculated. Scanning can be done simply to find locations of motif hits above a certain threshold, or to find instances of enriched motifs.

Contents

1 Introduction **2**

2 Basic sequence handling **2**
2.1 Creating random sequences . . . 2
2.2 Calculating sequence background . . . 3
2.3 Clustering sequences by k-let composition . . . 4

3 Shuffling **5**
3.1 Shuffling sequences . . . 5
3.2 Local shuffling . . . 7
3.3 Composition-matched backgrounds with `match_bkg()` . . . 8
3.4 Generating ground-truth positive sequences with `implant_motifs()` . . . 9
3.5 Motif pair co-occurrence with `motif_coocc()` . . . 9

4 Sequence scanning and enrichment **10**
4.1 Choosing a logodds threshold . . . 10
4.2 Regular and higher order scanning . . . 14
4.3 A faster alternative: `scan_sequences_lite()` . . . 18
4.4 Visualizing motif hits across sequences . . . 20
4.5 Enrichment analyses . . . 23
4.6 A faster alternative: `enrich_motifs_lite()` . . . 24
4.7 *De novo* motif discovery . . . 25
4.8 Positional enrichment with `motif_peaks()` . . . 25
4.9 Proximity enrichment with `motif_proximity()` . . . 26
4.10 Fixed and variable-length gapped motifs . . . 28
4.11 Detecting low complexity regions and sequence masking . . . 28

5 Motif discovery with MEME **30**

6 Miscellaneous string utilities **32**

Session info **33**

*benjamin.tremblay@uwaterloo.ca

1 Introduction

This vignette goes through generating your own sequences from a specified background model, shuffling sequences whilst maintaining a certain k-let size, and the scanning of sequences and scoring of motifs. For an introduction to sequence motifs, see the introductory vignette. For a basic overview of available motif-related functions, see the motif manipulation vignette. For a discussion on motif comparisons and P-values, see the motif comparisons and P-values vignette. For a worked example that strings these scanning, discovery, and enrichment pieces together on a real ChIP-seq dataset, see the ChIP-seq workflow vignette.

2 Basic sequence handling

2.1 Creating random sequences

The `Biostrings` package offers an excellent suite of functions for dealing with biological sequences. The `universalmotif` package hopes to help extend these by providing the `create_sequences()` and `shuffle_sequences()` functions. The first of these, `create_sequences()`, generates a set of letters in random order, then passes these strings to the `Biostrings` package to generate the final `XStringSet` object. The number and length of sequences can be specified. The probabilities of individual letters can also be set.

The `freqs` option of `create_sequences()` also takes higher order backgrounds. In these cases the sequences are constructed in a Markov-style manner, where the probability of each letter is based on which letters precede it.

```
library(universalmotif)
library(Biostrings)

## Create some DNA sequences for use with an external program (default
## is DNA):

sequences.dna <- create_sequences(seqnum = 500,
                                  freqs = c(A=0.3, C=0.2, G=0.2, T=0.3))
## writeXStringSet(sequences.dna, "dna.fasta")
sequences.dna
#> DNAStringSet object of length 500:
#>       width sequence
#>   [1]   100 ACGAACGCAGTAATTTCACCAATTGTATCATCT...AGTTCGAGCTAAGACATTATTCAGTTTGTCAG
#>   [2]   100 TCAGCAATCACACTAGACGGATGGATTCTACTG...TGACAAGCTATAATCCTCGTAGTATGATACGG
#>   [3]   100 AGTACATGCACGTAATTCTCTCAGTAGCTTGAT...CTAGTTGGCTCGAAGGCTAATTCCAACAGCTA
#>   [4]   100 GTTATTAAAATCTCCTAAAATCCGAAAAGAATA...TTTGTGGCCTGGAGTTTTCTGGATGAATTAAT
#>   [5]   100 TTCTAGCCGTTTAATATACACCTTATTGAAATC...ATCGCTTATTCTACTGATTAGCTGTTCCTCAT
#>   ...   ... ...
#> [496]   100 ATGGGATCGGTTAATAAAGTTAGTCAATGTAGA...AAGGTACCAGAGAGGATATTTGATAATACTGT
#> [497]   100 TAAAGACAACTAGTATGTTAGCGTTCAACGCTG...GGAAACCTTAAAGTAATATCTTATCCATCCTA
#> [498]   100 ACCAGGGCCATGAGGCACTATATGGGGTAGCAC...GATCATCCGACTGTCGCAAGTCATGAGTTATT
#> [499]   100 TTCTGAGTGGTATATTCGCTGCTCTCAGGACCT...TCTTTCCGTGCTATTTACGATTGACTAGATGA
#> [500]   100 GTTACATACGCCAGACCCGGTATAAATCGAGCC...TTCTAATTATATATGAGAAGAATCAGGGTGCC

## Amino acid:

create_sequences(alphabet = "AA")
#> AAStringSet object of length 100:
#>       width sequence
```

```
#>   [1]   100 VHDWTDHIPDNKRREKSWWADWAQSSSKKSNYH...EWLLSGMFFGFKITDLEYAQQKLQFFEYQCQV
#>   [2]   100 QHAIIHAFGYRKGWNFDRNKCPEHNGHLECYYN...VFSRMCNAHAKKYHAPSWQMIIYQFYCVPPEV
#>   [3]   100 AHGDNVLEIHTPHWECSCWVRYVLMGQERCYIH...CVALNARYHNMNFCDNIAQCLQETHWYKRGNE
#>   [4]   100 RCMGPTCVVQSAEIQRQVNKKHWCWIVHHNCES...EDVPYQPGTCRSHKEIADFRNTCEMCADYSGD
#>   [5]   100 MPQEPVRKFCLMNWNMGWTIQKAEQSTCNLPIE...EREKVWSTAHELTDACPIMSRAGACNENAYFC
#>   ...   ... ...
#>  [96]   100 CKDKNLNFAPTMINFTNWWVPVGKAGGRYQFSN...IKLIQLNWVIMIDHNCDYKYGMYHLLTCSSKK
#>  [97]   100 NQYSFSADMELMEYVQVYYIDYIKRGHCCAVMW...SEKCCCHRDVLTVPGMTSWCYLVVFNHTDVVA
#>  [98]   100 LLYIACMWCRFMITAVKVHWTRPLETPYPVSAE...WDFARTNTRCQGIMSPAALNWGAIDPYTFPGW
#>  [99]   100 LYHILIQIDTRLTFPKMHPYHMNCKRFDRYAVK...QQYGNPESHKEPPYIEVYGSSGPTTHQWKKVQ
#> [100]   100 APEQLPARWQWLYVNRKHEQVMIQDVSMYTLVE...KARKNGATKMYGPSVQYGGLGILNLFHRWEYR

## Any set of characters can be used

create_sequences(alphabet = paste0(letters, collapse = ""))
#> BStringSet object of length 100:
#>       width sequence
#>   [1]   100 ydjstsfjwylvwyofqwcfahzdizpqqucuh...iyahgeegtrrngifdlnejaxeqshxirlrd
#>   [2]   100 zcynytwqxqzmghzveraolwsspgjiuqvut...mmkapzzwrpgxcgdfdcsqetmtzvxpkdmt
#>   [3]   100 sfxpugkbfipvwvnbvmmrybfrcypqyzago...eowmnnkwpvvwefxdniwmcmokyjkydypg
#>   [4]   100 rrjzfnpaqpaiclcogtieaxorcskqrsaon...kwuhzciypfftqudozokipscvodctsjh
#>   [5]   100 sbyzzsyiolcqnoobasctbaygwevluuwki...lpubqzpwbpeidvlercpcpbhjhdqmuxgx
#>   ...   ... ...
#>  [96]   100 idhocfsypqoyqhxapdwokxzhagzgxcics...ezzmbefsakdusdrnawolfycigpxhbeiv
#>  [97]   100 mdezilrclcawangtmjzvsymklidfmscgp...nldtckxntuqzyhjbeeesaatmvdvtxxhk
#>  [98]   100 mxtsbfasemegmkqtnybruragmvpxijwlm...qqnbuefuvyhvqlncrxsjeckidhdqxrlg
#>  [99]   100 ohvnrcuwphcygqzkftqdnxzgynzoqlmaf...ryvjsqttmtszmdkypizlnzscworrskhx
#> [100]   100 ndfkydnktsqryglqfnferayninsdrbzxt...axfwlufwxqexhsqlyrlfrxkgcywboruv
```

2.2 Calculating sequence background

Sequence backgrounds can be retrieved for DNA and RNA sequences with `oligonucleotideFrequency()` from `Biostrings`. Unfortunately, no such `Biostrings` function exists for other sequence alphabets. The `universalmotif` package provides `get_bkg()` to remedy this. Similarly, the `get_bkg()` function can calculate higher order backgrounds for any alphabet as well. It is recommended to use `oligonucleotideFrequency()` for very long (e.g. billions of characters) DNA and RNA sequences whenever possible though, as it is much faster than `get_bkg()`.

```
library(universalmotif)

## Background of DNA sequences:
dna <- create_sequences()
get_bkg(dna, k = 1:2)
#> DataFrame with 20 rows and 3 columns
#>            klet     count probability
#>     <character> <numeric>   <numeric>
#> 1             A      2484   0.2484000
#> 2             C      2479   0.2479000
#> 3             G      2523   0.2523000
#> 4             T      2514   0.2514000
#> 5            AA       633   0.0639394
#> ...         ...       ...         ...
#> 16           GT       630   0.0636364
#> 17           TA       594   0.0600000
```

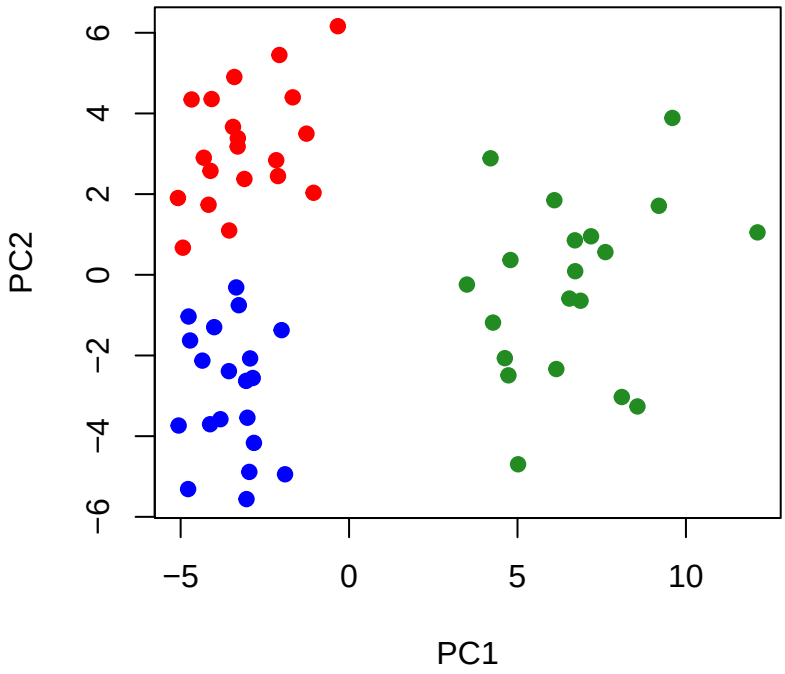
```
#> 18           TC       631   0.0637374
#> 19           TG       642   0.0648485
#> 20           TT       630   0.0636364

## Background of non DNA/RNA sequences:
qwerty <- create_sequences("QWERTY")
get_bkg(qwerty, k = 1:2)
#> DataFrame with 42 rows and 3 columns
#>            klet     count probability
#>     <character> <numeric>   <numeric>
#> 1             E      1705      0.1705
#> 2             Q      1651      0.1651
#> 3             R      1636      0.1636
#> 4             T      1621      0.1621
#> 5             W      1681      0.1681
#> ...         ...       ...         ...
#> 38           YQ       266   0.0268687
#> 39           YR       278   0.0280808
#> 40           YT       261   0.0263636
#> 41           YW       279   0.0281818
#> 42           YY       308   0.0311111
```

2.3 Clustering sequences by k-let composition

One way to compare sequences is by k-let composition. The following example illustrates how one could go about doing this using only the `universalmotif` package and base graphics.

```
library(universalmotif)

## Generate three random sets of sequences:
s1 <- create_sequences(seqnum = 20,
  freqs = c(A = 0.3, C = 0.2, G = 0.2, T = 0.3))
s2 <- create_sequences(seqnum = 20,
  freqs = c(A = 0.4, C = 0.4, G = 0.1, T = 0.1))
s3 <- create_sequences(seqnum = 20,
  freqs = c(A = 0.2, C = 0.3, G = 0.3, T = 0.2))

## Create a function to get properly formatted k-let counts:
get_klet_matrix <- function(seqs, k, groupName) {
  bkg <- get_bkg(seqs, k = k, merge.res = FALSE)
  bkg <- bkg[, c("sequence", "klet", "count")]
  bkg <- reshape(bkg, idvar = "sequence", timevar = "klet",
    direction = "wide")
  suppressWarnings(as.data.frame(cbind(Group = groupName, bkg)))
}

## Calculate k-let content (up to you what size k you want!):
s1 <- get_klet_matrix(s1, 4, 1)
s2 <- get_klet_matrix(s2, 4, 2)
s3 <- get_klet_matrix(s3, 4, 3)

# Combine everything into a single object:
sAll <- rbind(s1, s2, s3)
```

```
## Do the PCA:
sPCA <- prcomp(sAll[, -(1:2)])

## Plot the PCA:
plot(sPCA$x, col = c("red", "forestgreen", "blue")[sAll$Group], pch = 19)
```



This example could be improved by using `tidyr::pivot_wider()` instead of `reshape()` (the former is much faster), and plotting the PCA using the `ggfortify` package to create a nicer `ggplot2` plot. Feel free to play around with different ways of plotting the data! Additionally, you could even try using t-SNE instead of PCA (such as via the `Rtsne` package).

3 Shuffling

3.1 Shuffling sequences

When performing *de novo* motif searches or motif enrichment analyses, it is common to do so against a set of background sequences. In order to properly identify consistent patterns or motifs in the target sequences, it is important that there be maintained a certain level of sequence composition between the target and background sequences. This reduces results which are derived purely from base differential letter frequency biases.

In order to avoid these results, typically it is desirable to use a set of background sequences which preserve a certain `k`-let size (such as dinucleotide or trinucleotide frequencies in the case of DNA sequences). Though for some cases a set of similar sequences may already be available for use as background sequences, usually background sequences are obtained by shuffling the target sequences, while preserving a desired `k`-let size. For this purpose, a commonly used tool is `uShuffle` (Jiang et al. 2008). The `universalmotif` package aims to provide its own `k`-let shuffling capabilities for use within `R` via `shuffle_sequences()`.

The `universalmotif` package offers three different methods for sequence shuffling: `euler`, `markov` and `linear`. The first method, `euler`, can shuffle sequences while preserving any desired `k`-let size. Furthermore, 1-letter counts will always be maintained. However, due to the nature of the method, the first and last letters will remain unshuffled. This method is based on the initial random Eulerian walk algorithm proposed by Altschul and Erickson (1985) and the subsequent cycle-popping algorithm detailed by Propp and Wilson

(1998) for quickly and efficiently finding Eulerian walks.

The second method, `markov`, can only guarantee that the approximate `k`-let frequency will be maintained, but not that the original letter counts will be preserved. The `markov` method involves determining the original `k`-let frequencies, then creating a new set of sequences which will have approximately similar `k`-let frequency. As a result the counts for the individual letters will likely be different. Essentially, it involves a combination of determining `k`-let frequencies followed by `create_sequences()`. This type of pseudo-shuffling is discussed by Fitch (1983).

The third method, `linear`, preserves the original 1-letter counts exactly, but uses a more crude shuffling technique. In this case the sequence is split into sub-sequences every `k`-let (of any size), which are then re-assembled randomly. This means that while shuffling the same sequence multiple times with `method = "linear"` will result in different sequences, they will all have started from the same set of `k`-length sub-sequences (just re-assembled differently).

```
library(universalmotif)
library(Biostrings)
data(ArabidopsisPromoters)

## Potentially starting off with some external sequences:
# ArabidopsisPromoters <- readDNAStringSet("ArabidopsisPromoters.fasta")

euler <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "euler")
markov <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "markov")
linear <- shuffle_sequences(ArabidopsisPromoters, k = 2, method = "linear")
k1 <- shuffle_sequences(ArabidopsisPromoters, k = 1)
```

Let us compare how the methods perform:

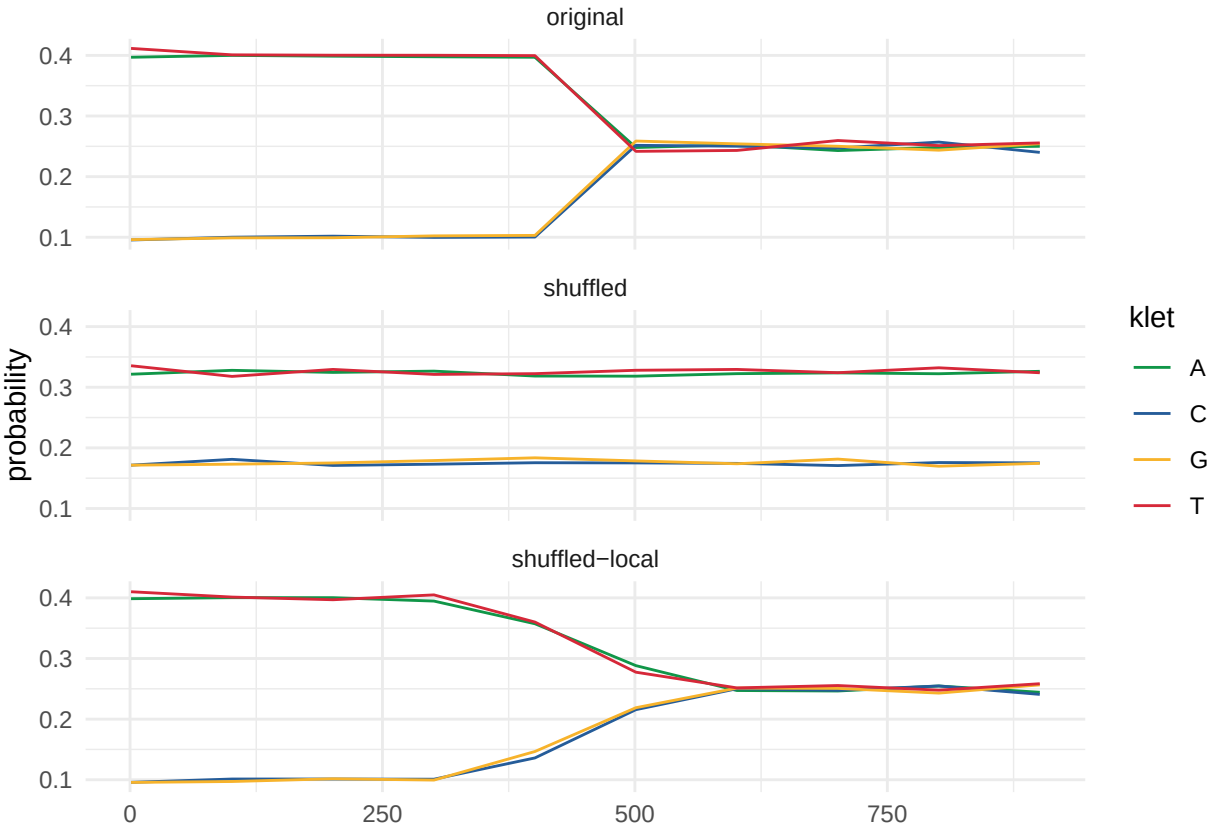
```
o.letter <- get_bkg(ArabidopsisPromoters, 1)
e.letter <- get_bkg(euler, 1)
m.letter <- get_bkg(markov, 1)
l.letter <- get_bkg(linear, 1)

data.frame(original=o.letter$count, euler=e.letter$count,
  markov=m.letter$count, linear=l.letter$count, row.names = DNA_BASES)
#>   original euler markov linear
#> A    17384 17384  17310  17384
#> C     8081  8081   8084   8081
#> G     7583  7583   7616   7583
#> T    16952 16952  16990  16952

o.counts <- get_bkg(ArabidopsisPromoters, 2)
e.counts <- get_bkg(euler, 2)
m.counts <- get_bkg(markov, 2)
l.counts <- get_bkg(linear, 2)

data.frame(original=o.counts$count, euler=e.counts$count,
  markov=m.counts$count, linear=l.counts$count,
  row.names = get_klets(DNA_BASES, 2))
#>    original euler markov linear
#> AA     6893  6893   6878   6497
#> AC     2614  2614   2599   2703
#> AG     2592  2592   2590   2624
#> AT     5276  5276   5220   5543
#> CA     3014  3014   3001   2902
```

```
#> CC     1376  1376   1393   1316
#> CG     1051  1051   1016   1117
#> CT     2621  2621   2668   2740
#> GA     2734  2734   2770   2670
#> GC     1104  1104   1116   1150
#> GG     1176  1176   1207   1202
#> GT     2561  2561   2519   2554
#> TA     4725  4725   4647   5299
#> TC     2977  2977   2967   2903
#> TG     2759  2759   2793   2635
#> TT     6477  6477   6566   6095
```

3.2 Local shuffling

If you have a fairly heterogeneous sequence and wish to preserve the presence of local "patches" of differential sequence composition, you can set `window = TRUE` in the `shuffle_sequences()` function. In the following example, the sequence of interest has an AT rich first half followed by a second half with an even background. The impact on this specific sequence composition is observed after regular and local shuffling, using the per-window functionality of `get_bkg()` (via `window = TRUE`). Fine-tune the window size and overlap between windows with `window.size` and `window.overlap`.

```
library(Biostrings)
library(universalmotif)
library(ggplot2)

myseq <- DNAStringSet(paste0(
  create_sequences(seqlen = 500, freqs = c(A=0.4, T=0.4, C=0.1, G=0.1)),
  create_sequences(seqlen = 500)
))

myseq_shuf <- shuffle_sequences(myseq)
myseq_shuf_local <- shuffle_sequences(myseq, window = TRUE)

myseq_bkg <- get_bkg(myseq, k = 1, window = TRUE)
myseq_shuf_bkg <- get_bkg(myseq_shuf, k = 1, window = TRUE)
myseq_shuf_local_bkg <- get_bkg(myseq_shuf_local, k = 1, window = TRUE)

myseq_bkg$group <- "original"
myseq_shuf_bkg$group <- "shuffled"
myseq_shuf_local_bkg$group <- "shuffled-local"

myseq_all <- suppressWarnings(as.data.frame(
  rbind(myseq_bkg, myseq_shuf_bkg, myseq_shuf_local_bkg)
))

ggplot(myseq_all, aes(x = start, y = probability, colour = klet)) +
  geom_line() +
  theme_minimal() +
  scale_colour_manual(values = universalmotif:::DNA_COLOURS) +
  xlab(element_blank()) +
  facet_wrap(~group, ncol = 1)
#> Warning: `label` cannot be a <ggplot2::element_blank> object.
```



3.3 Composition-matched backgrounds with `match_bkg()`

`shuffle_sequences()` randomises each input sequence in place, preserving that sequence's own k-let composition. This is appropriate enough when the input set is roughly homogeneous, but it does leave a confounder behind: if the input sequences carry a systematic composition bias relative to the genome (GC-rich ChIP-seq peaks, AT-rich promoters, and so on), then shuffling alone will not control for it. `match_bkg()` takes a different tack, sampling real sequences from a larger universe so that they match the input on GC fraction and length (the binned approach that HOMER uses).

```
library(universalmotif)
library(Biostrings)
data(ArabidopsisPromoters)

## Pretend the first 10 promoters are "target" and the next 40 are the
## universe; in real usage `universe` would be extracted from a
## BSgenome over a larger pool of genomic regions.
target   <- ArabidopsisPromoters[1:10]
universe <- ArabidopsisPromoters[11:50]

set.seed(1)
bkg <- match_bkg(target, universe, n.per.target = 2)
length(bkg)   # 20 = 10 targets x 2 matches each
#> [1] 20
```

The result is an `XStringSet` you can pass directly to other functions:

```
enrich_motifs_lite(motifs, target, bkg.sequences = bkg)
motif_finder(target, bkg.sequences = bkg)
```

Use `plot_match_bkg()` to visually verify the match landed:

```
plot_match_bkg(target, bkg)
```

This simply overlays the GC-fraction and length density curves for the target and the matched background, in a two-panel ggplot.

`match_bkg()` complements `shuffle_sequences()` rather than replacing it (and the same is true the other way around). A shuffle is what you want when you are after a per-sequence, k-let-preserving null, whereas a matched background is the better choice when you need to control for composition bias relative to some genomic universe. For reproducibility, do remember to call `set.seed()` before `match_bkg()`.

3.4 Generating ground-truth positive sequences with `implant_motifs()`

When you are benchmarking a discovery, scanning, or enrichment pipeline, you need positive sequences that carry known motif instances at known positions, so that recall and precision can be scored against an answer key. `implant_motifs()` is there to produce exactly those: it takes a set of motifs together with some host sequences, samples each instance column-by-column from the motif's PPM, and then overwrites the bases at the chosen position. Three insertion modes are available (a fixed `n.per.seq` count, a Poisson `rate` per base pair, or an explicit list of `positions`), and a few optional knobs (`centre.bias`, `min.spacing`, `strand`) let you mimic ChIP-seq-style central enrichment or enforce non-overlapping placement.

```
library(universalmotif)
data(ArabidopsisMotif)
hosts <- create_sequences("DNA", seqnum = 100, seqlen = 500)
## Plant one instance per sequence and get both the sequences and the
## ground-truth positions in a single call. Use set.seed() for
## reproducibility.
set.seed(1)
res   <- implant_motifs(ArabidopsisMotif, hosts, n.per.seq = 1,
                        return.indices = TRUE)
out   <- res$sequences
truth <- res$indices
head(truth)
## Quantify recovery
hits <- scan_sequences(ArabidopsisMotif, out, threshold = 0.85,
                       threshold.type = "logodds")
## Visualise positional density (qvalue = 1 keeps the motif even when,
## as here, the implants are placed uniformly)
peaks <- motif_peaks(as.data.frame(hits), seq.length = 500, qvalue = 1)
plot_motif_peaks(peaks)
```

With `return.indices = FALSE` (the default) you get back only the modified `XStringSet`. With `return.indices = TRUE` you instead get a list of two elements: `sequences` (the modified `XStringSet`) and `indices` (a `data.frame` with columns `sequence.i`, `motif.i`, `start`, `width`, `strand`, and `planted`).

3.5 Motif pair co-occurrence with `motif_coocc()`

`enrich_motifs()` and `enrich_motifs_lite()` test whether each motif is enriched on its own. Transcription factors, though, often bind as heterodimers, or cooperate at composite cis-regulatory modules, so a natural next question is whether any pair of motifs co-occurs in the same sequences more often than chance would predict. This is what `motif_coocc()` is for. For every pair of input motifs it builds the 2x2 contingency table of per-sequence presence and absence, runs a one-sided Fisher's exact test, and then applies a BH correction across all of the tested pairs.

```
library(universalmotif)
data(ArabidopsisMotif)
```

```r
m_other <- create_motif("CACGTGCACGTG", name = "Ebox12")
seqs <- create_sequences("DNA", seqnum = 200, seqlen = 500)
## (In real use, hand the function your peaks or promoters here.)
co <- motif_coocc(list(ArabidopsisMotif, m_other), seqs, pvalue = 1e-3)
co
```

There are two input paths. The default one (shown above) scans internally via `scan_sequences_lite()`, and so is DNA/RNA only. Alternatively, you can hand the function a precomputed hit table:

```r
hits <- scan_sequences_lite(motifs, seqs, pvalue = 1e-3)
co   <- motif_coocc(motifs, hits = hits, n.sequences = length(seqs))
```

The hit-table path accepts any alphabet (amino acid, or custom), since once a hit table exists the co-occurrence is really just set arithmetic on the `(motif.i, sequence.i)` integers.

Setting `max.distance = N` additionally reports two descriptive columns: `both.clustered` (the subset of co-occurring sequences that have an `(A, B)` hit pair within `N` bp of each other) and `median.distance` (the median nearest-pair spacing across those sequences). The Fisher p-value itself is unchanged by this, since it always tests the same thing ("do A and B occur in the same sequences more often than chance?") on the unfiltered 2x2 table. The two spatial columns are best thought of as interpretive aids for flagging heterodimer-like arrangements, rather than as a separate statistical test in their own right.

4 Sequence scanning and enrichment

There are many motif-programs available with sequence scanning capabilities, such as HOMER and tools from the MEME suite. The `universalmotif` package does not aim to supplant these, but rather provide convenience functions for quickly scanning a few sequences without needing to leave the `R` environment. Furthermore, these functions allow for taking advantage of the higher-order (`multifreq`) motif format described here.

Two scanning-related functions are provided: `scan_sequences()` / `scan_sequences_lite()` and `enrich_motifs()` / `enrich_motifs_lite()`. The latter simply runs `scan_sequences()` / `scan_sequences_lite()` twice on a set of target and background sequences. Given a motif of length `n`, `scan_sequences()` / `scan_sequences_lite()` considers every possible `n`-length subsequence in a sequence and scores it using the PWM format. If the match surpasses the minimum threshold, it is reported. This is the case regardless of whether one is scanning with a regular motif, or using the higher-order (`multifreq`) motif format (the `multifreq` matrix is converted to a PWM) with the older `scan_sequences()`.

4.1 Choosing a logodds threshold

Before scanning a set of sequences, one must first decide the minimum logodds threshold for retrieving matches. This decision is not always the same between scanning programs out in the wild, nor is it usually made clear what the cutoff is or how it is decided. As a result, `universalmotif` aims to be as transparent as possible in this regard by allowing for complete control of the threshold. For more details on PWMs, see the introductory vignette.

Logodds thresholds

One way is to set a cutoff between 0 and 1, then multiply it by the highest possible PWM score to get a threshold. The `matchPWM()` function from the `Biostrings` package for example uses a default of 0.8 (shown as `"80%"`). This is quite arbitrary of course, and every motif will end up with a different threshold. For high information content motifs, there is really no right or wrong threshold, as they tend to have fewer non-specific positions. This means that incorrect letters in a match will be more punishing. To illustrate this, contrast the following PWMs:

```
library(universalmotif)
m1 <- create_motif("TATATATATA", nsites = 50, type = "PWM", pseudocount = 1)
m2 <- matrix(c(0.10,0.27,0.23,0.19,0.29,0.28,0.51,0.12,0.34,0.26,
               0.36,0.29,0.51,0.38,0.23,0.16,0.17,0.21,0.23,0.36,
               0.45,0.05,0.02,0.13,0.27,0.38,0.26,0.38,0.12,0.31,
               0.09,0.40,0.24,0.30,0.21,0.19,0.05,0.30,0.31,0.08),
             byrow = TRUE, nrow = 4)
m2 <- create_motif(m2, alphabet = "DNA", type = "PWM")
m1["motif"]
#>           T         A         T         A         T         A         T
#> A -5.672425  1.978626 -5.672425  1.978626 -5.672425  1.978626 -5.672425
#> C -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425
#> G -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425 -5.672425
#> T  1.978626 -5.672425  1.978626 -5.672425  1.978626 -5.672425  1.978626
#>           A         T         A
#> A  1.978626 -5.672425  1.978626
#> C -5.672425 -5.672425 -5.672425
#> G -5.672425 -5.672425 -5.672425
#> T -5.672425  1.978626 -5.672425
m2["motif"]
#>            S           H           C           N           N           N
#> A -1.3219281  0.09667602 -0.12029423 -0.3959287  0.2141248  0.1491434
#> C  0.5260688  0.19976951  1.02856915  0.6040713 -0.1202942 -0.6582115
#> G  0.8479969 -2.33628339 -3.64385619 -0.9434165  0.1110313  0.5897160
#> T -1.4739312  0.66371661 -0.05889369  0.2630344 -0.2515388 -0.4102840
#>            R           N           N           V
#> A  1.0430687 -1.0732490  0.4436067  0.04222824
#> C -0.5418938 -0.2658941 -0.1202942  0.51171352
#> G  0.0710831  0.5897160 -1.0588937  0.29598483
#> T -2.3074285  0.2486791  0.3103401 -1.65821148
```

In the first example, sequences which do not have a matching base in every position are punished heavily. The maximum logodds score in this case is approximately 20, and for each incorrect position the score is reduced approximately by 5.7. This means that a threshold of zero would allow for at most three mismatches. At this point, it is up to you how many mismatches you would deem appropriate.

P-values

This thinking becomes impossible for the second example. In this case, mismatches are much less punishing, to the point that one could ask: what even constitutes a mismatch? The answer to this question is usually much more difficult in such cases. An alternative to manually deciding upon a threshold is to instead start with the maximum P-value one would consider appropriate for a match. If, say, we want matches with a P-value of at most 0.001, then we can use `motif_pvalue()` to calculate the appropriate threshold (see the comparisons and P-values vignette for details on motif P-values).

```
motif_pvalue(m2, pvalue = 0.001)
#> [1] 4.859
```

Multiple testing-corrected P-values

This P-value can be further refined to correct for multiple testing (and becomes a Q-value). There are three available corrections that can be set in `scan_sequences()`: Bonferroni ("bonferroni"), Benjamini & Hochberg ("BH"), and the false discovery rate ("fdr") based on the empirical null distribution of motif hits in a set of sequences. They are excellently explained in Noble (2009), and these explanations will be briefly regurgitated here.

To begin to understand how these different corrections are implemented, consider the following motif, sequences, example P-value for an example motif hit, and the theoretical maximum number of motif hits:

```
library(universalmotif)
data(ArabidopsisMotif)
data(ArabidopsisPromoters)

(Example.Score <- score_match(ArabidopsisMotif, "TTCTCTTTTTTTTTT"))
#> [1] 16.81
(Example.Pvalue <- motif_pvalue(ArabidopsisMotif, Example.Score))
#> [1] 6.612819e-07

(Max.Possible.Hits <- sum(width(ArabidopsisPromoters) - ncol(ArabidopsisMotif) + 1))
#> [1] 49300
```

The first correction method, Bonferroni, is by far the simplest. To calculate it, take the P-value of a motif hit and multiply it by the theoretical maximum number of hits:

```
(Example.bonferroni <- Example.Pvalue * Max.Possible.Hits)
#> [1] 0.0326012
```

As you can imagine, the level of punishment the P-value receives corresponds to the size of the sequences you are scanning. If you are scanning an entire genome, then you can expect this to be very punishing and only return near-perfect matches (or no matches). However, for smaller sets of sequences, this correction can be more appropriate.

Next, Benjamini & Hochberg. To perform this correction, the P-value is divided by its fractional rank in the list of P-values for all theoretically possible hits sorted in ascending order (this assumes that P-values are independent and uniformly distributed on [0, 1] under the null hypothesis). It is important to note that this means the correction cannot be calculated before the sequences have been scanned for the motif, and P-values have been calculated for all returned hits. When requesting this type of Q-value for the minimum threshold of score, `scan_sequences()` instead calculates the threshold from the input Q-value as a P-value, then filters the final results after Q-values have been calculated. Returning to our example:

```
(Scan.Results <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters,
  threshold = 0.8, threshold.type = "logodds", calc.qvals = FALSE))
#> DataFrame with 20 rows and 14 columns
#>               motif   motif.i    sequence sequence.i     start      stop
#>         <character> <integer> <character>  <integer> <integer> <integer>
#> 1   YTTTYTTTTTYTTTY         1   AT1G05670         47        68        82
#> 2   YTTTYTTTTTYTTTY         1   AT1G19510         45       402       416
#> 3   YTTTYTTTTTYTTTY         1   AT1G49840         27       899       913
#> 4   YTTTYTTTTTYTTTY         1   AT2G22500         14       946       960
#> 5   YTTTYTTTTTYTTTY         1   AT2G22500         14       948       962
#> ...             ...       ...         ...        ...       ...       ...
#> 16  YTTTYTTTTTYTTTY         1   AT3G23170         34       603       617
#> 17  YTTTYTTTTTYTTTY         1   AT4G19520          3       792       806
#> 18  YTTTYTTTTTYTTTY         1   AT4G19520          3       793       807
#> 19  YTTTYTTTTTYTTTY         1   AT4G27652         20       879       893
#> 20  YTTTYTTTTTYTTTY         1   AT4G27652         20       881       895
#>         score           match thresh.score min.score max.score score.pct
#>     <numeric>     <character>    <numeric> <numeric> <numeric> <numeric>
#> 1      15.407 GTTTCTTTTTTCTTT      15.0272   -125.07    18.784   82.0219
#> 2      17.405 TTTTCTTTTTCTTTT      15.0272   -125.07    18.784   92.6586
#> 3      15.177 CTTTTTGTTTTTTTC      15.0272   -125.07    18.784   80.7975
#> 4      15.827 TCCTCTCTTTCTCTC      15.0272   -125.07    18.784   84.2579
```

```
#> 5      15.908 CTCTCTTTCTCTCTT      15.0272   -125.07    18.784   84.6891
#> ...       ...                ...         ...       ...       ...       ...
#> 16     15.734 GTTTCTTCTTTTTTT      15.0272   -125.07    18.784   83.7628
#> 17     15.352 TTTTTTTTTTTTTTT      15.0272   -125.07    18.784   81.7291
#> 18     15.352 TTTTTTTTTTTTTTT      15.0272   -125.07    18.784   81.7291
#> 19     16.410 TTTTCTCTTTTTTTT      15.0272   -125.07    18.784   87.3616
#> 20     16.810 TTCTCTTTTTTTTTT      15.0272   -125.07    18.784   89.4911
#>          strand      pvalue
#>     <character>   <numeric>
#> 1             + 3.95595e-06
#> 2             + 2.44369e-07
#> 3             + 5.01977e-06
#> 4             + 2.53853e-06
#> 5             + 2.39165e-06
#> ...         ...         ...
#> 16            + 2.83419e-06
#> 17            + 4.33848e-06
#> 18            + 4.33848e-06
#> 19            + 1.23950e-06
#> 20            + 6.61282e-07
```

First we sort and calculate the fractional ranks of our P-values, and then divide the P-values:

```
Pvalues <- Scan.Results$pvalue
Pvalues.Ranks <- rank(Pvalues) / Max.Possible.Hits
Qvalues.BH <- Pvalues / Pvalues.Ranks
(Example.BH <- Qvalues.BH[Scan.Results$match == "TTCTCTTTTTTTTTT"][1])
#> [1] 0.00652024
```

Finally, calculating the false discovery rate from the empirical distribution of scores. This method requires some additional steps, as we must obtain the observed and null distributions of hits in our sequences. Then for each hit, divide the number of hits with a score equal to or greater in the null distribution by the number of hits with a score equal to or greater in the observed distribution. Along the way we must be wary of the nonmonotonicity of the final Q-values (meaning that as scores get smaller the Q-value does not always increase), and thus always select the minimum available Q-value as the score increases. To get the null distribution of hits, we can simply use the P-values associated with each score as these are analytically calculated from the null based on the background probabilities (see `?motif_pvalue`).

```
Scan.Results <- Scan.Results[order(Scan.Results$score, decreasing = TRUE), ]
Observed.Hits <- 1:nrow(Scan.Results)
Null.Hits <- Max.Possible.Hits * Scan.Results$pvalue
Qvalues.fdr <- Null.Hits / Observed.Hits
Qvalues.fdr <- rev(cummin(rev(Qvalues.fdr)))
(Example.fdr <- Qvalues.fdr[Scan.Results$match == "TTCTCTTTTTTTTTT"][1])
#> [1] 0.00652024
```

Similarly to Benjamini & Hochberg, these can only be known after scanning has occurred.

To summarize, we can compare the initial P-value with the different corrections:

```
knitr::kable(
  data.frame(
    What = c("Score", "P-value", "bonferroni", "BH", "fdr"),
    Value = format(
      c(Example.Score, Example.Pvalue, Example.bonferroni, Example.BH, Example.fdr),
      scientific = FALSE
```

```
    )
  ),
  format = "markdown", caption = "Comparing P-value correction methods"
)
```

Table 1: Comparing P-value correction methods

What	Value
Score	16.8100000000000
P-value	0.0000006612819
bonferroni	0.0326011986749
BH	0.0065202397350
fdr	0.0065202397350

Use your best judgement as to which method is most appropriate for your specific use case.

4.2 Regular and higher order scanning

Furthermore, the `scan_sequences()` function offers the ability to scan using the `multifreq` slot, if available. This allows one to take into account inter-positional dependencies, and get matches which more faithfully represent the original sequences from which the motif originated.

```
library(universalmotif)
library(Biostrings)
data(ArabidopsisPromoters)

## A 2-letter example:

motif.k2 <- create_motif("CWWWWCC", nsites = 6)
sequences.k2 <- DNAStringSet(rep(c("CAAAACC", "CTTTTCC"), 3))
motif.k2 <- add_multifreq(motif.k2, sequences.k2)
```

Regular scanning:

```
scan_sequences(motif.k2, ArabidopsisPromoters, RC = TRUE,
               threshold = 0.9, threshold.type = "logodds")
#> DataFrame with 94 rows and 15 columns
#>           motif   motif.i    sequence sequence.i     start      stop     score
#>     <character> <integer> <character>  <integer> <integer> <integer> <numeric>
#> 1         motif         1   AT1G03850          4       203       209      9.08
#> 2         motif         1   AT1G03850          4       334       328      9.08
#> 3         motif         1   AT1G03850          4       713       707      9.08
#> 4         motif         1   AT1G05670         47       706       700      9.08
#> 5         motif         1   AT1G06160         48       498       492      9.08
#> ...         ...       ...         ...        ...       ...       ...       ...
#> 90        motif         1   AT5G22690         46        81        87      9.08
#> 91        motif         1   AT5G22690         46       362       368      9.08
#> 92        motif         1   AT5G24660         49       146       140      9.08
#> 93        motif         1   AT5G58430         16       332       338      9.08
#> 94        motif         1   AT5G58430         16       343       349      9.08
#>           match thresh.score min.score max.score score.pct      strand
#>     <character>    <numeric> <numeric> <numeric> <numeric> <character>
#> 1       CTAATCC        8.172   -19.649      9.08       100           +
#> 2       CTTTTCC        8.172   -19.649      9.08       100           -
```

```
#> 3       CTTAACC        8.172   -19.649      9.08       100           -
#> 4       CTTTACC        8.172   -19.649      9.08       100           -
#> 5       CTAAACC        8.172   -19.649      9.08       100           -
#> ...         ...          ...       ...       ...       ...         ...
#> 90      CAATACC        8.172   -19.649      9.08       100           +
#> 91      CAAATCC        8.172   -19.649      9.08       100           +
#> 92      CATTACC        8.172   -19.649      9.08       100           -
#> 93      CATAACC        8.172   -19.649      9.08       100           +
#> 94      CAAATCC        8.172   -19.649      9.08       100           +
#>          pvalue    qvalue
#>       <numeric> <numeric>
#> 1   0.000976562         1
#> 2   0.000976562         1
#> 3   0.000976562         1
#> 4   0.000976562         1
#> 5   0.000976562         1
#> ...         ...       ...
#> 90  0.000976562         1
#> 91  0.000976562         1
#> 92  0.000976562         1
#> 93  0.000976562         1
#> 94  0.000976562         1
```

Using 2-letter information to scan:

```
scan_sequences(motif.k2, ArabidopsisPromoters, use.freq = 2, RC = TRUE,
               threshold = 0.9, threshold.type = "logodds")
#> DataFrame with 8 rows and 15 columns
#>         motif   motif.i    sequence sequence.i     start      stop     score
#>   <character> <integer> <character>  <integer> <integer> <integer> <numeric>
#> 1       motif         1   AT1G19510         45       960       966    17.827
#> 2       motif         1   AT1G49840         27       959       965    17.827
#> 3       motif         1   AT1G77210         32       184       190    17.827
#> 4       motif         1   AT1G77210         32       954       960    17.827
#> 5       motif         1   AT2G37950         15       751       757    17.827
#> 6       motif         1   AT3G57640         33       917       923    17.827
#> 7       motif         1   AT4G12690         12       938       944    17.827
#> 8       motif         1   AT4G14365         35       977       983    17.827
#>         match thresh.score min.score max.score score.pct      strand
#>   <character>    <numeric> <numeric> <numeric> <numeric> <character>
#> 1     CTTTTCC      16.0443   -16.842    17.827       100           +
#> 2     CTTTTCC      16.0443   -16.842    17.827       100           +
#> 3     CAAAACC      16.0443   -16.842    17.827       100           +
#> 4     CAAAACC      16.0443   -16.842    17.827       100           +
#> 5     CAAAACC      16.0443   -16.842    17.827       100           +
#> 6     CTTTTCC      16.0443   -16.842    17.827       100           +
#> 7     CAAAACC      16.0443   -16.842    17.827       100           +
#> 8     CTTTTCC      16.0443   -16.842    17.827       100           +
#>        pvalue    qvalue
#>     <numeric> <numeric>
#> 1 1.90735e-06 0.0236988
#> 2 1.90735e-06 0.0236988
#> 3 1.90735e-06 0.0236988
#> 4 1.90735e-06 0.0236988
```

```
#> 5 1.90735e-06 0.0236988
#> 6 1.90735e-06 0.0236988
#> 7 1.90735e-06 0.0236988
#> 8 1.90735e-06 0.0236988
```

Furthermore, sequence scanning can be further refined to avoid overlapping hits. Consider:

```
motif <- create_motif("AAAAAA")

## Leave in overlapping hits:

scan_sequences(motif, ArabidopsisPromoters, RC = TRUE, threshold = 0.9,
               threshold.type = "logodds")
#> DataFrame with 491 rows and 15 columns
#>           motif   motif.i    sequence sequence.i     start      stop     score
#>     <character> <integer> <character>  <integer> <integer> <integer> <numeric>
#> 1         motif         1   AT1G03850          4        56        51    11.934
#> 2         motif         1   AT1G03850          4        57        52    11.934
#> 3         motif         1   AT1G03850          4        58        53    11.934
#> 4         motif         1   AT1G03850          4        59        54    11.934
#> 5         motif         1   AT1G03850          4       243       248    11.934
#> ...         ...       ...         ...        ...       ...       ...       ...
#> 487       motif         1   AT5G64310         22       589       594    11.934
#> 488       motif         1   AT5G64310         22       590       595    11.934
#> 489       motif         1   AT5G64310         22       591       596    11.934
#> 490       motif         1   AT5G64310         22       592       597    11.934
#> 491       motif         1   AT5G64310         22       696       701    11.934
#>           match thresh.score min.score max.score score.pct      strand
#>     <character>    <numeric> <numeric> <numeric> <numeric> <character>
#> 1        AAAAAA      10.7406   -39.948    11.934       100           -
#> 2        AAAAAA      10.7406   -39.948    11.934       100           -
#> 3        AAAAAA      10.7406   -39.948    11.934       100           -
#> 4        AAAAAA      10.7406   -39.948    11.934       100           -
#> 5        AAAAAA      10.7406   -39.948    11.934       100           +
#> ...         ...          ...       ...       ...       ...         ...
#> 487      AAAAAA      10.7406   -39.948    11.934       100           +
#> 488      AAAAAA      10.7406   -39.948    11.934       100           +
#> 489      AAAAAA      10.7406   -39.948    11.934       100           +
#> 490      AAAAAA      10.7406   -39.948    11.934       100           +
#> 491      AAAAAA      10.7406   -39.948    11.934       100           +
#>           pvalue    qvalue
#>        <numeric> <numeric>
#> 1   0.000244141 0.0494745
#> 2   0.000244141 0.0494745
#> 3   0.000244141 0.0494745
#> 4   0.000244141 0.0494745
#> 5   0.000244141 0.0494745
#> ...         ...       ...
#> 487 0.000244141 0.0494745
#> 488 0.000244141 0.0494745
#> 489 0.000244141 0.0494745
#> 490 0.000244141 0.0494745
#> 491 0.000244141 0.0494745
```

```
## Only keep the highest scoring hit amongst overlapping hits:

scan_sequences(motif, ArabidopsisPromoters, RC = TRUE, threshold = 0.9,
               threshold.type = "logodds", no.overlaps = TRUE)
#> DataFrame with 220 rows and 15 columns
#>           motif   motif.i    sequence sequence.i     start      stop     score
#>     <character> <integer> <character>  <integer> <integer> <integer> <numeric>
#> 1         motif         1   AT1G03850          4        56        51    11.934
#> 2         motif         1   AT1G03850          4       243       248    11.934
#> 3         motif         1   AT1G03850          4       735       740    11.934
#> 4         motif         1   AT1G05670         47        32        27    11.934
#> 5         motif         1   AT1G05670         47        78        73    11.934
#> ...         ...       ...         ...        ...       ...       ...       ...
#> 216       motif         1   AT5G64310         22       251       246    11.934
#> 217       motif         1   AT5G64310         22       342       347    11.934
#> 218       motif         1   AT5G64310         22       586       591    11.934
#> 219       motif         1   AT5G64310         22       592       597    11.934
#> 220       motif         1   AT5G64310         22       696       701    11.934
#>           match thresh.score min.score max.score score.pct      strand
#>     <character>    <numeric> <numeric> <numeric> <numeric> <character>
#> 1        AAAAAA      10.7406   -39.948    11.934       100           -
#> 2        AAAAAA      10.7406   -39.948    11.934       100           +
#> 3        AAAAAA      10.7406   -39.948    11.934       100           +
#> 4        AAAAAA      10.7406   -39.948    11.934       100           -
#> 5        AAAAAA      10.7406   -39.948    11.934       100           -
#> ...         ...          ...       ...       ...       ...         ...
#> 216      AAAAAA      10.7406   -39.948    11.934       100           -
#> 217      AAAAAA      10.7406   -39.948    11.934       100           +
#> 218      AAAAAA      10.7406   -39.948    11.934       100           +
#> 219      AAAAAA      10.7406   -39.948    11.934       100           +
#> 220      AAAAAA      10.7406   -39.948    11.934       100           +
#>           pvalue    qvalue
#>        <numeric> <numeric>
#> 1   0.000244141 0.0494745
#> 2   0.000244141 0.0494745
#> 3   0.000244141 0.0494745
#> 4   0.000244141 0.0494745
#> 5   0.000244141 0.0494745
#> ...         ...       ...
#> 216 0.000244141 0.0494745
#> 217 0.000244141 0.0494745
#> 218 0.000244141 0.0494745
#> 219 0.000244141 0.0494745
#> 220 0.000244141 0.0494745
```

Finally, the results can be returned as a `GRanges` object for further manipulation:

```
scan_sequences(motif.k2, ArabidopsisPromoters, RC = TRUE,
               threshold = 0.9, threshold.type = "logodds",
               return.granges = TRUE)
#> GRanges object with 94 ranges and 11 metadata columns:
#>         seqnames    ranges strand |       motif   motif.i sequence.i     score
#>            <Rle> <IRanges>  <Rle> | <character> <integer>  <integer> <numeric>
#>    [1] AT1G03850   203-209      + |       motif         1          4      9.08
```

```
#>     [2] AT1G03850   328-334      - |       motif         1          4      9.08
#>     [3] AT1G03850   707-713      - |       motif         1          4      9.08
#>     [4] AT1G05670   700-706      - |       motif         1         47      9.08
#>     [5] AT1G06160   956-962      + |       motif         1         48      9.08
#>     ...       ...       ...    ... .         ...       ...        ...       ...
#>    [90] AT5G22690   362-368      + |       motif         1         46      9.08
#>    [91] AT5G22690     52-58      - |       motif         1         46      9.08
#>    [92] AT5G24660   140-146      - |       motif         1         49      9.08
#>    [93] AT5G58430   332-338      + |       motif         1         16      9.08
#>    [94] AT5G58430   343-349      + |       motif         1         16      9.08
#>             match thresh.score min.score max.score score.pct      pvalue
#>       <character>    <numeric> <numeric> <numeric> <numeric>   <numeric>
#>     [1]     CTAATCC        8.172   -19.649      9.08       100 0.000976562
#>     [2]     CTTTTCC        8.172   -19.649      9.08       100 0.000976562
#>     [3]     CTTAACC        8.172   -19.649      9.08       100 0.000976562
#>     [4]     CTTTACC        8.172   -19.649      9.08       100 0.000976562
#>     [5]     CTAATCC        8.172   -19.649      9.08       100 0.000976562
#>     ...         ...          ...       ...       ...       ...         ...
#>    [90]     CAAATCC        8.172   -19.649      9.08       100 0.000976562
#>    [91]     CATTACC        8.172   -19.649      9.08       100 0.000976562
#>    [92]     CATTACC        8.172   -19.649      9.08       100 0.000976562
#>    [93]     CATAACC        8.172   -19.649      9.08       100 0.000976562
#>    [94]     CAAATCC        8.172   -19.649      9.08       100 0.000976562
#>           qvalue
#>        <numeric>
#>     [1]         1
#>     [2]         1
#>     [3]         1
#>     [4]         1
#>     [5]         1
#>     ...       ...
#>    [90]         1
#>    [91]         1
#>    [92]         1
#>    [93]         1
#>    [94]         1
#>    -------
#>    seqinfo: 50 sequences from an unspecified genome
```

4.3 A faster alternative: `scan_sequences_lite()`

For DNA or RNA scans that do not need multifreq scoring, gapped motifs, q-values, exhaustive P-values, or any threshold type other than a P-value, the lighter `scan_sequences_lite()` is generally faster, and it parallelises rather better across motifs. Internally it uses the same C++ scanner, computes its per-hit P-values via the same FIMO-style dynamic-programming algorithm, and returns the same set of hits; it simply skips the extra work that `scan_sequences()` has to do in order to support its broader feature set.

The default surface of `scan_sequences_lite()` is deliberately minimal (just six formals: `motifs`, `sequences`, `pvalue = 1e-4`, `RC = TRUE`, `nthreads = 1`, and `return.granges = NULL`). When `GenomicRanges` is installed the default output is a `GRanges`, and otherwise it falls back to a `data.frame`. The coordinates always satisfy `start <= end` regardless of strand (following the BED / GFF / `GRanges` convention), and on `-`-strand hits the `match` column holds the reverse complement of the sequence substring, that is, the matched orientation. Greedy hit-level deduplication is available via `no.overlaps = TRUE`, and is also exposed as a standalone `dedup_hits()` function for post-processing.

```r
## scan_sequences_lite() with all defaults
hits <- scan_sequences_lite(ArabidopsisMotif, ArabidopsisPromoters)

## Optional deduplication of overlapping hits
hits <- scan_sequences_lite(ArabidopsisMotif, ArabidopsisPromoters,
                            no.overlaps = TRUE)
```

When `scan_sequences()` is called with arguments that happen to map cleanly onto `scan_sequences_lite()`'s feature set, it emits a one-line hint pointing you at the faster function. (This can be silenced with `options(universalmotif.suggest.scan_sequences_lite = FALSE)`.)

Speed comparison vs `motifmatchr` and `yamtk`

To put these trade-offs in some perspective, the table below records the end-to-end wall-clock time for the four scanners on a single shared fixture: the first 50 motifs of HOCOMOCOv11, scanned against random DNA at five different sizes, with a P-value cutoff of 5×10^{-5}, both strands, and a uniform background. `motifmatchr::matchMotifs()` appears in two output modes, its default `out = "matches"` (one result per `(motif, sequence)`, which is strictly less work than the other three tools do) and `out = "positions"` (per position, and so directly comparable to the others). The hit sets from the CLI tool `yamtk`, `scan_sequences()`, `scan_sequences_lite()`, and `motifmatchr` in `positions` mode all agree to within about 5% (the small drift comes from `yamtk` building its PWM from a continuous PPM, whereas `universalmotif` rounds the PCM). Times are reported in seconds, and each cell shows `nthreads = 1 / nthreads = 4` wherever threading applies. These benchmarks were run on an MBP M5 under R 4.4.1.

Workload	yamtk	mm matches[1]	mm positions[2]	scan_sequences_lite	scan_sequences[3]
10 mot × 500 seq × 500 bp	0.03 / 0.01	0.05	5.67	0.30 / 0.13	0.39 / 0.39
50 mot × 500 seq × 500 bp	0.12 / 0.03	0.17	28.19	1.48 / 0.44	1.62 / 1.56
200 mot × 1000 seq × 500 bp	0.71 / 0.20	0.71	237.32	6.62 / 1.91	7.02 / 6.46
50 mot × 100 seq × 50 kb	1.34 / 0.37	0.30	6.09	3.71 / 1.11	3.79 / 2.18
200 mot × 100 seq × 50 kb	5.17 / 1.49	1.12	24.83	14.92 / 4.40	15.50 / 9.01

[1] `motifmatchr::matchMotifs(..., out = "matches")` (default): returns a sparse `(motif × sequence)` yes/no matrix. Strictly less work than the per-position scanners; a fair comparison only when you really only need to know whether each motif matched each sequence at all.

[2] `motifmatchr::matchMotifs(..., out = "positions")`: returns the full set of per-position hits as a list of `IRangesList`. This is the fair comparison against `yamtk`, `scan_sequences()`, and `scan_sequences_lite()`. `motifmatchr` is single-threaded from the R API in both modes.

[3] `scan_sequences(..., calc.pvals = TRUE)`, which performs the same work per hit as `scan_sequences_lite()`. The older `mapply`-based dispatch in `scan_sequences()` parallelises poorly for short-sequence workloads; this is the main cost that `scan_sequences_lite()` eliminates.

A few practical takeaways:

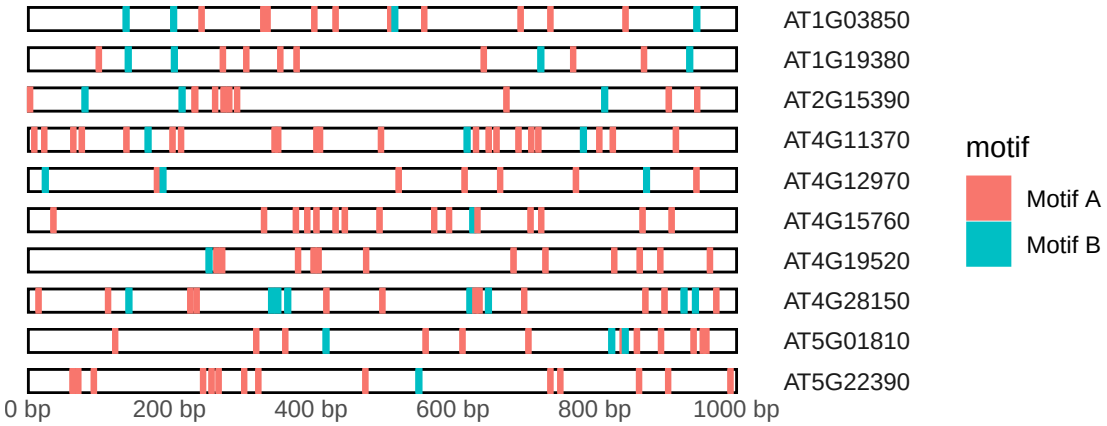
- `yamtk` (a dedicated CLI tool written in C) is the fastest scanner at every scale. The `universalmotif` package does not aim to match it; the goal is to stay within a small multiple while keeping everything inside `R`.
- `scan_sequences_lite()` reaches within ~3× of `yamtk scan` at 4 threads on long-sequence workloads, and is 2-3× faster than `scan_sequences(calc.pvals = TRUE)` at 4 threads across every configuration tested.
- For equivalent-work (per-position) scanning, `scan_sequences_lite()` is also substantially faster than `motifmatchr::matchMotifs(out = "positions")` at every scale: 5-100× depending on the input.
- `motifmatchr` is faster than the per-position scanners only in its default `out = "matches"` mode, where it skips per-position reporting entirely. Choose that mode when "did this motif match this sequence at all?" is the question you are actually asking.
- For new code that does not need any of `scan_sequences()`'s advanced features, I would generally reach for `scan_sequences_lite()`.

4.4 Visualizing motif hits across sequences

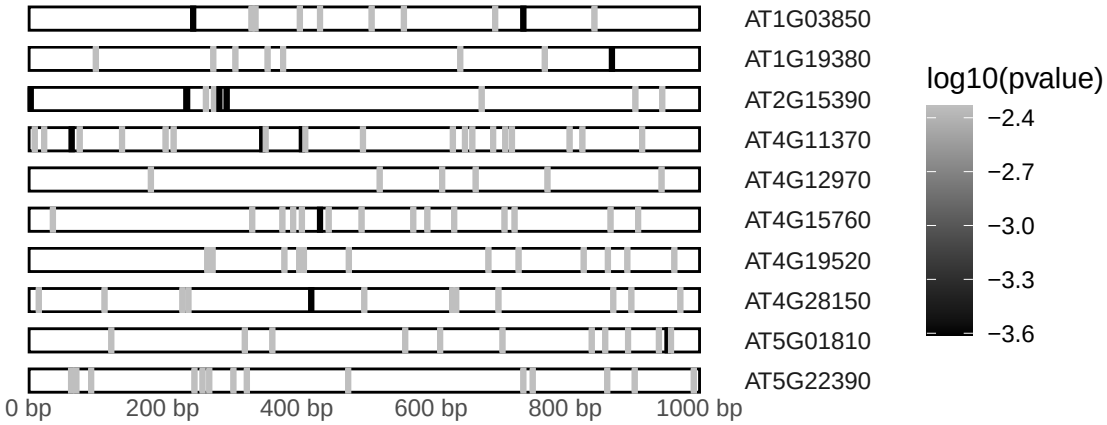
A few suggestions for different ways of plotting hits across sequences are presented here.

Using the `ggbio` package, it is rather trivial to generate nice visualisations of the output of `scan_sequences()`. This requires having the `GenomicRanges` and `ggbio` packages installed, and outputting the `scan_sequences()` result as a `GRanges` object (via `return.granges = TRUE`).

```
library(universalmotif)
library(GenomicRanges)
library(ggbio)

data(ArabidopsisPromoters)

motif1 <- create_motif("AAAAAA", name = "Motif A")
motif2 <- create_motif("CWWWWCC", name = "Motif B")

res <- scan_sequences(c(motif1, motif2), ArabidopsisPromoters[1:10],
  return.granges = TRUE, calc.pvals = TRUE, no.overlaps = TRUE,
  threshold = 0.2, threshold.type = "logodds")

## Just plot the motif hits:
autoplot(res, layout = "karyogram", aes(fill = motif, color = motif)) +
  theme(
    strip.background = element_rect(fill = NA, colour = NA),
    panel.background = element_rect(fill = NA, colour = NA)
  )
#> Scale for x is already present.
#> Adding another scale for x, which will replace the existing scale.
#> Scale for x is already present.
#> Adding another scale for x, which will replace the existing scale.
```



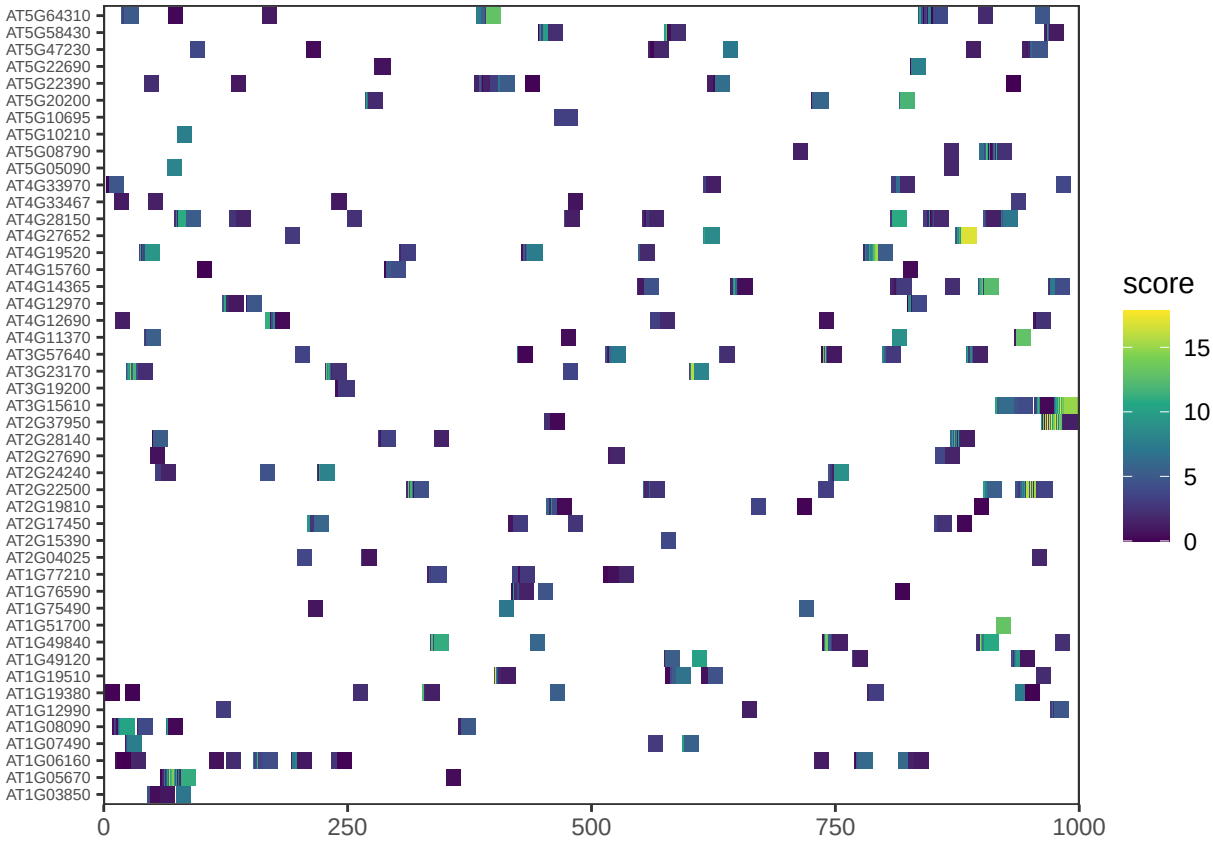

```
## Plot Motif A hits by P-value:
autoplot(res[res$motif.i == 1, ], layout = "karyogram",
  aes(fill = log10(pvalue), colour = log10(pvalue))) +
  scale_fill_gradient(low = "black", high = "grey75") +
  scale_colour_gradient(low = "black", high = "grey75") +
  theme(
    strip.background = element_rect(fill = NA, colour = NA),
    panel.background = element_rect(fill = NA, colour = NA)
  )
#> Scale for x is already present.
#> Adding another scale for x, which will replace the existing scale.
#> Scale for x is already present.
#> Adding another scale for x, which will replace the existing scale.
```



Alternatively, just a simple heatmap with only `ggplot2`.

```
library(universalmotif)
library(ggplot2)

data(ArabidopsisMotif)
data(ArabidopsisPromoters)

res <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters,
  threshold = 0, threshold.type = "logodds.abs")
res <- suppressWarnings(as.data.frame(res))
res$x <- mapply(function(x, y) mean(c(x, y)), res$start, res$stop)
```

```
ggplot(res, aes(x, sequence, fill = score)) +
  scale_fill_viridis_c() +
  scale_x_continuous(expand = c(0, 0), limits = c(0, 1000)) +
  xlab(element_blank()) +
  ylab(element_blank()) +
  geom_tile(width = ncol(ArabidopsisMotif)) +
  theme_bw() +
  theme(panel.grid = element_blank(), axis.text.y = element_text(size = 6))
#> Warning: Removed 2 rows containing missing values or values outside the scale range
#> (`geom_tile()`).
#> Warning: `label` cannot be a <ggplot2::element_blank> object.
#> `label` cannot be a <ggplot2::element_blank> object.
```

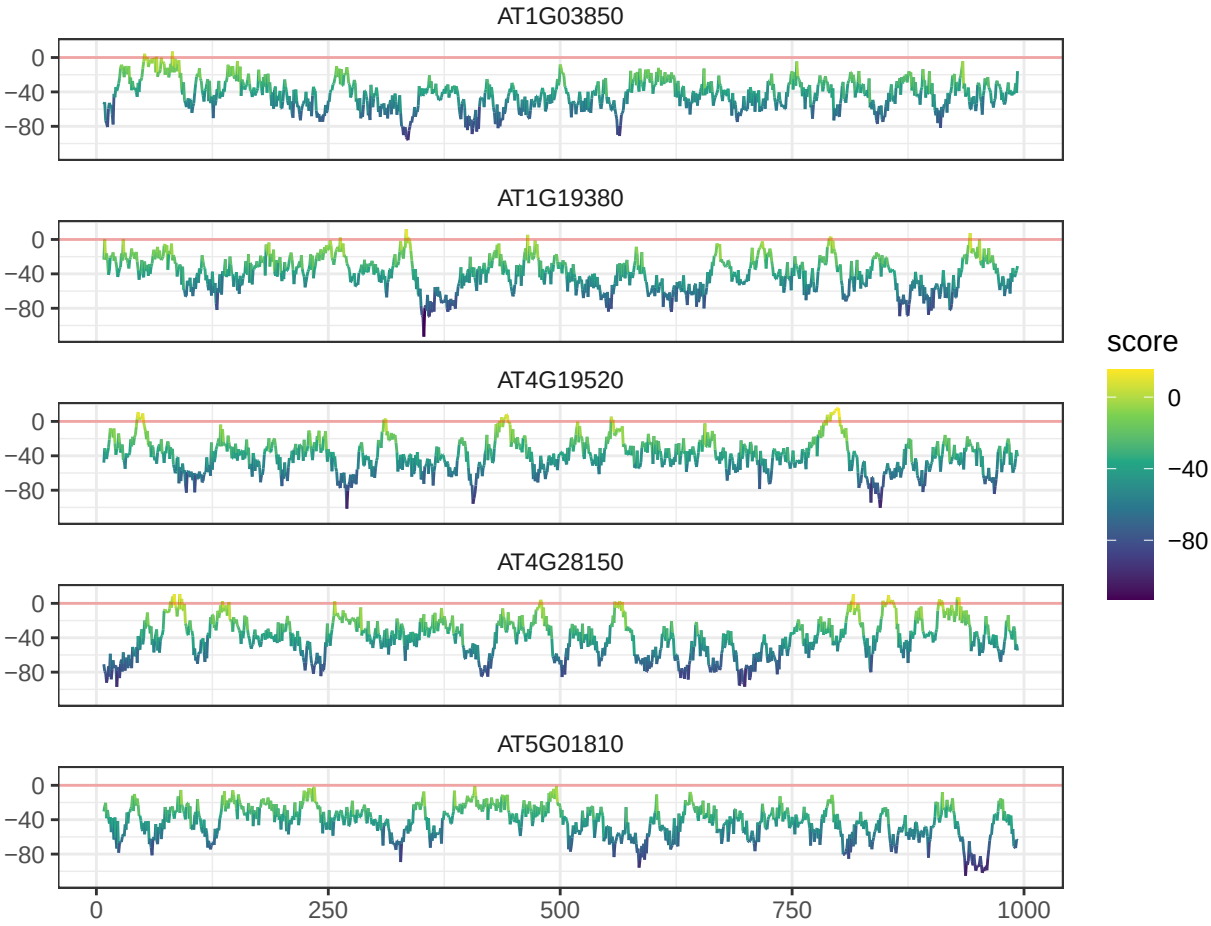


Using packages such as **ggExtra** or **ggpubr**, one could even plot marginal histogram or density plots above or below to illustrate any motif positional preference within the sequences. (Though keep in mind that the hit coordinates and sequence lengths would need to be normalized if not all sequences were of the same length, as they are here.)

Finally, the distribution of all possible motif scores could be shown as a line plot across the sequences.

```
library(universalmotif)
library(ggplot2)

data(ArabidopsisMotif)
data(ArabidopsisPromoters)

res <- scan_sequences(ArabidopsisMotif, ArabidopsisPromoters[1:5],
  threshold = -Inf, threshold.type = "logodds.abs")
```

```
res <- suppressWarnings(as.data.frame(res))
res$position <- mapply(function(x, y) mean(c(x, y)), res$start, res$stop)

ggplot(res, aes(position, score, colour = score)) +
  geom_line() +
  geom_hline(yintercept = 0, colour = "red", alpha = 0.3) +
  theme_bw() +
  scale_colour_viridis_c() +
  facet_wrap(~sequence, ncol = 1) +
  xlab(element_blank()) +
  ylab(element_blank()) +
  theme(strip.background = element_blank())
#> Warning: `label` cannot be a <ggplot2::element_blank> object.
#> `label` cannot be a <ggplot2::element_blank> object.
```



4.5 Enrichment analyses

The `universalmotif` package offers the ability to search for enriched motif sites in a set of sequences via `enrich_motifs()` / `enrich_motifs_lite()`. There is little complexity to this, as it simply runs `scan_sequences()` twice: once on a set of target sequences, and once on a set of background sequences. After which the results between the two sequences are collated and run through enrichment tests. The background sequences can be given explicitly, or else `enrich_motifs()` / `enrich_motifs_lite()` will create background sequences on its own by using `shuffle_sequences()` on the target sequences.

Let us consider the following basic example:

```
library(universalmotif)
data(ArabidopsisMotif)
data(ArabidopsisPromoters)

enrich_motifs(ArabidopsisMotif, ArabidopsisPromoters, shuffle.k = 3,
              threshold = 0.001, RC = TRUE)
#> DataFrame with 1 row and 15 columns
#>              motif   motif.i motif.consensus target.hits target.seq.hits
#>        <character> <integer>     <character>   <integer>       <integer>
#> 1 YTTTYTTTTTYTTTY         1 YTYTYTTYTTYTTTY         244              50
#>   target.seq.count  bkg.hits bkg.seq.hits bkg.seq.count        Pval        Qval
#>          <integer> <integer>    <integer>     <integer>   <numeric>   <numeric>
#> 1               50       151           49            50 1.72467e-06 1.72467e-06
#>          Eval pct.target.seq.hits pct.bkg.seq.hits target.enrichment
#>     <numeric>           <numeric>        <numeric>         <numeric>
#> 1 3.44934e-06                 100               98           1.02041
```

Here we can see that the motif is significantly enriched in the target sequences. The `Pval` was calculated by calling `stats::fisher.test()`.

One final point: always keep in mind the `threshold` parameter, as this will ultimately decide the number of hits found. (A bad threshold can lead to a false negative.)

4.6 A faster alternative: `enrich_motifs_lite()`

For the common case (DNA/RNA motifs, a P-value hit threshold, Fisher's exact test, and Benjamini-Hochberg q-values), `enrich_motifs_lite()` offers a stripped-down counterpart to `enrich_motifs()`. It is built on top of `scan_sequences_lite()`, and so inherits its speed advantages.

```
library(universalmotif)
data(ArabidopsisMotif)
data(ArabidopsisPromoters)

enrich_motifs_lite(ArabidopsisMotif, ArabidopsisPromoters,
              pvalue = 1e-3, qvalue = 0.1, rng.seed = 1, test = "sites")
#>              motif motif.i       consensus target.seq.n target.seq.hits
#> 1 YTTTYTTTTTYTTTY       1 YTYTYTTYTTYTTTY           50              50
#>   target.site.hits bkg.seq.n bkg.seq.hits bkg.site.hits enrichment
#> 1              683        50           48           252   2.710317
#>   log2.enrichment       pvalue       qvalue
#> 1        1.438462 5.602412e-47 5.602412e-47
```

Note the `test = "sites"` here. The two functions default to different tests: `enrich_motifs()` defaults to `mode = "total.hits"` (a per-site comparison), while `enrich_motifs_lite()` defaults to `test = "seqs"` (a per-sequence comparison). Setting `test = "sites"` makes `enrich_motifs_lite()` mirror the `enrich_motifs()` call above. The per-sequence test would be a poor fit for this particular example anyway: the promoters are long (1000 bp) and the motif is a low-complexity poly-pyrimidine that appears in nearly every sequence of both the target and the shuffled background, so per-sequence presence barely differs between them and the enrichment is only marginally significant.

The output is a plain `data.frame` with 13 columns: `motif`, `motif.i`, `consensus`, `target.seq.n`, `target.seq.hits`, `target.site.hits`, `bkg.seq.n`, `bkg.seq.hits`, `bkg.site.hits`, `enrichment`, `log2.enrichment`, `pvalue`, `qvalue`.

Two test modes are supported:

- `test = "seqs"` (default): Fisher's exact on per-sequence hit presence, equivalent to `enrich_motifs(mode = "seq.hits")`.
- `test = "sites"`: Fisher's exact on per-position rates, equivalent to `enrich_motifs(mode = "total.hits")`.

When an `enrich_motifs()` call uses only features that map cleanly onto `enrich_motifs_lite()`, a one-line hint is again emitted pointing you at the leaner function. (Silence it with `options(universalmotif.suggest.enrich_motifs_ = FALSE)`.)

4.7 *De novo* motif discovery

`motif_finder()` is a *de novo* motif discovery function. For each motif width in a configurable range, it enumerates the over-represented k-mer seeds (using a Fisher's exact test on per-sequence presence against a background set), aligns the Hamming-distance-1 neighbours of the best seed to form a PPM, refines that PPM over two re-scan passes, accepts the motif if its Fisher's exact p-value passes `stop.pvalue`, and then masks the covered positions and repeats, up to `nmotifs` times per width. Once every width has been processed, the motifs are deduplicated across widths, BH-adjusted, and IC-trimmed at the flanks.

```
library(universalmotif)
library(Biostrings)

set.seed(1)
seqs <- create_sequences(seqnum = 100, seqlen = 200, rng.seed = 1)
## Plant a motif in 80 of the 100 sequences:
planted <- DNAString("TTGACATA")
for (i in seq_len(80)) {
  pos <- sample.int(200 - 8, 1)
  subseq(seqs[[i]], start = pos, width = 8) <- planted
}

motifs <- motif_finder(seqs, rng.seed = 1, min.width = 6, max.width = 10)
motifs[, c("name", "consensus", "width", "seqs_pos", "seqs_neg", "pvalue", "qvalue")]
```

The return shape is a `universalmotif_df`: one row per discovered motif, with the motif S4 objects sitting in the `motif` list-column, and the standard `to_df()` slot columns (`name`, `consensus`, `alphabet`, and so on) joined to the discovery-specific stats (`rank`, `width`, `seqs_pos`, `seqs_neg`, `sites_pos`, `sites_neg`, `n_pos`, `n_neg`, `pvalue`, `qvalue`). A call to `to_list()` recovers a plain list of `universalmotif` S4 objects, which then plug directly into `view_motifs_lite()`, `compare_motifs_lite()`, `write_meme()`, and the rest of the package.

The discovery widths default to 6 through 15, the per-width motif cap to 10, and the result-level q-value filter to 1e-3; set `qvalue = 1` if you would rather have all of the discovered motifs returned regardless of significance. Unless they are supplied explicitly through `bkg.sequences`, the background sequences are auto-generated by k=2 shuffling of the input.

4.8 Positional enrichment with `motif_peaks()`

`motif_peaks()` tests whether motif hits cluster non-uniformly along the input sequences. It implements an analytical, CentriMo-style test: under the null hypothesis that hit positions are uniform over `[1, seq.length]`, the number of hits falling in a candidate window follows a binomial distribution. The function reports the most-enriched window for each motif, Bonferroni-corrected across the windows tested and then BH-adjusted across the motifs.

Two modes are supported:

- `mode = "central"` (default): only centre-of-sequence windows are tested. Appropriate when input sequences are centred on a reference position (e.g. ChIP-seq peak summits).

- `mode = "local"`: the window centre is also varied, scanning the whole sequence for any positionally-enriched region.

```
library(universalmotif)
library(Biostrings)

set.seed(1)
seqs <- create_sequences(seqnum = 200, seqlen = 500, rng.seed = 1)
planted <- DNAString("TTGACATA")
## Plant in 150 of 200 sequences, centred on position 250 (+/- 4 bp).
for (i in seq_len(150)) {
  pos <- 246L + sample.int(8, 1) - 1L
  subseq(seqs[[i]], start = pos, width = 8) <- planted
}
m <- create_motif("TTGACATA", name = "test")

## Scan sequences, then test for central enrichment.
hits  <- scan_sequences_lite(m, seqs, pvalue = 1e-3, return.granges = TRUE)
peaks <- motif_peaks(hits)
peaks[, c("motif", "best.window", "best.center", "hits.in", "nhits",
          "enrichment", "pvalue", "qvalue")]

## Visualise the per-motif positional distribution + best window.
plot_motif_peaks(peaks)
```

The returned `data.frame` carries one row per significant motif, together with a list-column `centers` holding the best-hit centre positions per sequence. `plot_motif_peaks()` consumes that column directly, so there is no need to thread your original scan results back through it.

The input is polymorphic: `motif_peaks()` will accept either a `data.frame` or a `GRanges` coming out of `scan_sequences()` or `scan_sequences_lite()`. When the input is a `GRanges` that has its `seqlengths()` set (which `scan_sequences_lite()` does for you automatically), `seq.length` is inferred; otherwise you will need to pass it explicitly.

4.9 Proximity enrichment with `motif_proximity()`

`motif_peaks()` assumes equal-length sequences centred on a reference position. When you instead have motif hits scattered across whole chromosomes together with a set of *anchor* coordinates (peak summits, TSSs, SNPs, or another factor's binding sites), `motif_proximity()` tests whether the hits sit closer to those anchors than chance. It is the genome-wide sibling of `motif_peaks()`, sharing the same binomial engine but measuring distance to the nearest anchor rather than position within a fixed window.

Three null models are available through `method`:

- `"binomial"` (default): for each candidate distance `d`, the fraction of the scanned genome lying within `d` of an anchor is the success probability of a binomial test on the hit count. This is fast, but the uniform-hit null cannot correct for composition bias.
- `"permutation"`: the anchors (not the hits) are randomised within the scanned genome to build an empirical null. Because the real hits stay fixed, this absorbs composition and mappability bias, and so it is the method to prefer when such bias is plausible. Call `set.seed()` beforehand for reproducible p-values.
- `"ks"`: a threshold-free Kolmogorov-Smirnov test of the per-hit distance-to-nearest-anchor against its analytical null.

`count = "hits"` (the default) collapses overlapping hits of a motif to one representative before counting hits near anchors, whereas `count = "anchors"` instead counts the fraction of anchors that have a nearby hit.

You can supply a hit table directly, or let `motif_proximity()` scan for you by passing `motifs` and `sequences` (an `XStringSet`, or a `BSgenome`, in which case only the chromosomes carrying anchors are scanned):

```
library(universalmotif)
library(Biostrings)
library(GenomicRanges)

## Treat three created sequences as small chromosomes of unequal length.
set.seed(1)
chrs <- c(create_sequences(seqnum = 1, seqlen = 6000, rng.seed = 1),
          create_sequences(seqnum = 1, seqlen = 4000, rng.seed = 2),
          create_sequences(seqnum = 1, seqlen = 3000, rng.seed = 3))
names(chrs) <- paste0("chr", 1:3)

## Plant a motif a few bp downstream of a set of anchors on chr1.
anchor.pos <- seq(500, 5500, by = 250)
planted <- DNAString("TTGACATA")
for (p in anchor.pos)
  subseq(chrs[["chr1"]], start = p + 5L, width = 8) <- planted
anchors <- GRanges("chr1", IRanges(anchor.pos, width = 1), strand = "+")

m <- create_motif("TTGACATA", name = "anchored")

## Either pass a hit table ...
hits <- scan_sequences_lite(m, chrs, pvalue = 1e-3, return.granges = TRUE)
motif_proximity(hits, anchors)

## ... or let motif_proximity() scan internally (scan.pvalue mirrors the
## pvalue used above, so the two calls are equivalent).
motif_proximity(motifs = m, sequences = chrs, anchors = anchors,
                scan.pvalue = 1e-3)
```

The universe (the null support) defaults to the whole chromosomes implied by `seqlengths(hits)`, and only chromosomes carrying at least one anchor are kept. For masked or partial scans, pass the scanned regions explicitly through `universe`; otherwise the within-distance fraction is overstated and the test becomes anti-conservative.

Anchor width matters as much as their positions. The near zone is each anchor widened by the distance on either side, so wide anchors enlarge the within-distance fraction, drive the enrichment toward one, and collapse a hit's distance to zero whenever it falls inside the region. If the question is enrichment near a point (summits, TSSs, SNPs), use 1 bp anchors and reduce regions to their reference position first, for example `resize(peaks, width = 1, fix = "center")` or the called summit. Feeding raw, wide peaks instead can silently gut the test even when the signal is strong. Keep regions only when you genuinely mean "within or around these intervals", and then choose `dist.thresholds` relative to their width.

Because the anchors are a `GRanges`, their strand can drive a directional test. With stranded anchors such as TSSs, `orientation = "upstream"` or `"downstream"` builds a strand-aware one-sided zone, so a motif enriched downstream of the feature is distinguished from one enriched upstream (the motif above was planted downstream of the `+` anchors):

```
motif_proximity(hits, anchors, orientation = "downstream")
motif_proximity(hits, anchors, orientation = "upstream")
```

`plot_motif_proximity()` shows the per-hit distances to the nearest anchor, faceted by motif, with the most-enriched zone shaded. When the anchors are stranded the distances are signed, so upstream and downstream fall on opposite sides of zero:

```
res <- motif_proximity(hits, anchors)
plot_motif_proximity(res)
```

4.10 Fixed and variable-length gapped motifs

`universalmotif` class motifs can be gapped, which can be used by `scan_sequences()` and `enrich_motifs()`. Note that gapped motif support is currently limited to these two functions. All other functions will ignore the gap information, and even discard them in functions such as `merge_motifs()`.

First, obtain the component motifs:

```
library(universalmotif)
data(ArabidopsisPromoters)

m1 <- create_motif("TTTATAT", name = "PartA")
m2 <- create_motif("GGTTCGA", name = "PartB")
```

Then, combine them and add the desired gap. In this case, a gap will be added between the two motifs which can range in size from 4-6 bases.

```
m <- cbind(m1, m2)
m <- add_gap(m, gaploc = ncol(m1), mingap = 4, maxgap = 6)
m
#>
#>        Motif name:   PartA/PartB
#>          Alphabet:   DNA
#>              Type:   PCM
#>           Strands:   +-
#>          Total IC:   28
#>       Pseudocount:   0
#>         Consensus:   TTTATAT..GGTTCGA
#>      Target sites:   1
#>     Gap locations:   7-8
#>         Gap sizes:   4-6
#>
#>   T T T A T A T    G G T T C G A
#> A 0 0 0 1 0 1 0 .. 0 0 0 0 0 0 1
#> C 0 0 0 0 0 0 0 .. 0 0 0 0 1 0 0
#> G 0 0 0 0 0 0 0 .. 1 1 0 0 0 1 0
#> T 1 1 1 0 1 0 1 .. 0 0 1 1 0 0 0
```

Now, it can be used directly in `scan_sequences()` or `enrich_motifs()`:

```
scan_sequences(m, ArabidopsisPromoters, threshold = 0.4, threshold.type = "logodds")
#> No hits found.
#> DataFrame with 0 rows and 15 columns
```

4.11 Detecting low complexity regions and sequence masking

Highly-repetitive low complexity regions can oftentimes cause problems during *de novo* motif discovery, leading to obviously false motifs being returned. One way to get around this issue is to preemptively remove or mask these regions. The `universalmotif` package includes a few functions which can help carry out this task.

Using `mask_seqs()`, one can mask a specific pattern of letters in `XStringSet` objects. Consider the following sequences:

```
library(universalmotif)
library(Biostrings)

Ex.seq <- DNAStringSet(c(
  A = "GTTGAAAAAAAAAAAAAAAACAGACGT",
  B = "TTAGATGGCCCATAGCTTATACGGCAA",
  C = "AATAAAATGCTTAGGAAATCGATTGCC"
))
```

We can easily mask portions that contain, say, stretches of at least 8 As:

```
mask_seqs(Ex.seq, "AAAAAAAA")
#> DNAStringSet object of length 3:
#>     width sequence                                          names
#> [1]    27 GTTG----------------CAGACGT                      A
#> [2]    27 TTAGATGGCCCATAGCTTATACGGCAA                       B
#> [3]    27 AATAAAATGCTTAGGAAATCGATTGCC                       C
```

Alternatively, instead of masking a known stretch of letters, one can find low complexity regions using `sequence_complexity()`, and then mask specific regions in the sequences using `mask_ranges()`. The `sequence_complexity()` function has several complexity metrics available: the Wootton-Federhen (Wootton and Federhen 1993) and Trifonov (Trifonov 1990) algorithms (and their approximations) are well described in Orlov and Potapov (2004), and DUST in Morgulis et al. (2006). See `?sequence_complexity` for more details.

```
(Ex.DUST <- sequence_complexity(Ex.seq, window.size = 10, method = "DUST",
    return.granges = TRUE))
#> GRanges object with 15 ranges and 1 metadata column:
#>        seqnames    ranges strand | complexity
#>           <Rle> <IRanges>  <Rle> |  <numeric>
#>    [1]        A      1-10      * |   0.857143
#>    [2]        A      6-15      * |   4.000000
#>    [3]        A     11-20      * |   4.000000
#>    [4]        A     16-25      * |   0.428571
#>    [5]        A     21-27      * |   0.000000
#>    ...      ...       ...    ... .        ...
#>   [11]        C      1-10      * |   0.285714
#>   [12]        C      6-15      * |   0.000000
#>   [13]        C     11-20      * |   0.000000
#>   [14]        C     16-25      * |   0.000000
#>   [15]        C     21-27      * |   0.000000
#>   -------
#>   seqinfo: 3 sequences from an unspecified genome
```

Using the DUST algorithm, we can see there are a couple of regions which spike in the complexity score (for this particular algorithm, more complex sequences converge towards zero). Now it is only a matter of filtering for those regions and using `mask_ranges()`.

```
(Ex.DUST <- Ex.DUST[Ex.DUST$complexity >= 3])
#> GRanges object with 2 ranges and 1 metadata column:
#>       seqnames    ranges strand | complexity
#>          <Rle> <IRanges>  <Rle> |  <numeric>
#>   [1]        A      6-15      * |          4
#>   [2]        A     11-20      * |          4
#>   -------
#>   seqinfo: 3 sequences from an unspecified genome
```

```
mask_ranges(Ex.seq, Ex.DUST)
#> DNAStringSet object of length 3:
#>     width sequence                                          names
#> [1]    27 GTTGA---------------CAGACGT                      A
#> [2]    27 TTAGATGGCCCATAGCTTATACGGCAA                       B
#> [3]    27 AATAAAATGCTTAGGAAATCGATTGCC                       C
```

Now these sequences could be used directly with `scan_sequences()` or written to a fasta file using `Biostrings::writeXStringSet()` for use with an external *de novo* motif discovery program such as MEME.

5 Motif discovery with MEME

Note: In the time since the inception of the `run_meme()` function, Sabrina Nystrom (a contributor to `universalmotif`) has created the `memes` package as an interface to much of the `MEME` suite. It is fully interoperable with the `universalmotif` package and provides a much more convenient way to run `MEME` programs from within `R`. Install it from Bioconductor with `BiocManager::install("memes")`.

The `universalmotif` package provides a simple wrapper to the powerful motif discovery tool `MEME` (Bailey and Elkan 1994). To run an analysis with `MEME`, all that is required is a set of `XStringSet` class sequences (defined in the `Biostrings` package), and `run_meme()` will take care of running the program and reading the output for use within `R`.

The first step is to check that R can find the `MEME` binary in your `$PATH` by running `run_meme()` without any parameters. If successful, you should see the default `MEME` help message in your console. If not, then you'll need to provide the complete path to the `MEME` binary. There are two options:

```
library(universalmotif)

## 1. Once per session: via `options()`

options(meme.bin = "/path/to/meme/bin/meme")

run_meme(...)

## 2. Once per run: via `run_meme()`

run_meme(..., bin = "/path/to/meme/bin/meme")
```

Now we need to get some sequences to use with `run_meme()`. At this point we can read sequences from disk or extract them from one of the `Bioconductor BSgenome` packages.

```
library(universalmotif)
data(ArabidopsisPromoters)

## 1. Read sequences from disk (in fasta format):

library(Biostrings)

# The following `read*()` functions are available in Biostrings:
# DNA: readDNAStringSet
# DNA with quality scores: readQualityScaledDNAStringSet
# RNA: readRNAStringSet
# Amino acid: readAAStringSet
# Any: readBStringSet
```

```
sequences <- readDNAStringSet("/path/to/sequences.fasta")

run_meme(sequences, ...)

## 2. Extract from a `BSgenome` object:

library(GenomicFeatures)
library(TxDb.Athaliana.BioMart.plantsmart28)
library(BSgenome.Athaliana.TAIR.TAIR9)

# Let us retrieve the same promoter sequences from ArabidopsisPromoters:
gene.names <- names(ArabidopsisPromoters)

# First get the transcript coordinates from the relevant `TxDb` object:
transcripts <- transcriptsBy(TxDb.Athaliana.BioMart.plantsmart28,
                             by = "gene")[gene.names]

# There are multiple transcripts per gene, we only care for the first one
# in each:

transcripts <- lapply(transcripts, function(x) x[1])
transcripts <- unlist(GRangesList(transcripts))

# Then the actual sequences:

# Unfortunately this is a case where the chromosome names do not match
# between the two databases

seqlevels(TxDb.Athaliana.BioMart.plantsmart28)
#> [1] "1"  "2"  "3"  "4"  "5"  "Mt" "Pt"
seqlevels(BSgenome.Athaliana.TAIR.TAIR9)
#> [1] "Chr1" "Chr2" "Chr3" "Chr4" "Chr5" "ChrM" "ChrC"

# So we must first rename the chromosomes in `transcripts`:
seqlevels(transcripts) <- seqlevels(BSgenome.Athaliana.TAIR.TAIR9)

# Finally we can extract the sequences
promoters <- getPromoterSeq(transcripts,
                            BSgenome.Athaliana.TAIR.TAIR9,
                            upstream = 1000, downstream = 0)

run_meme(promoters, ...)
```

Once the sequences are ready, there are a few important options to keep in mind. One is whether to conserve the output from `MEME`. The default is not to, but this can be changed by setting the relevant option:

```
run_meme(sequences, output = "/path/to/desired/output/folder")
```

The second important option is the search function (`objfun`). Some search functions such as the default `classic` do not require a set of background sequences, whilst some do (such as `de`). If you choose one of the latter, then you can either let `MEME` create them for you (it will shuffle the target sequences) or you can provide them via the `control.sequences` parameter.

Finally, choose how you'd like the data imported into `R`. Once the `MEME` program exits, `run_meme()` will import the results into `R` with `read_meme()`. At this point you can decide if you want just the motifs themselves

(`readsites = FALSE`) or if you'd like the original sequence sites as well (`readsites = TRUE`, the default). Doing the latter gives you the option of generating higher order representations for the imported `MEME` motifs as shown here:

```
motifs <- run_meme(sequences)
motifs.k23 <- mapply(add_multifreq, motifs$motifs, motifs$sites)
```

There are a wealth of other `MEME` options available, such as the number of desired motifs (`nmotifs`), the width of desired motifs (`minw`, `maxw`), the search mode (`mod`), assigning sequence weights (`weights`), using a custom alphabet (`alph`), and many others. See the output from `run_meme()` for a brief description of the options, or visit the online manual for more details.

6 Miscellaneous string utilities

Since biological sequences are usually contained in `XStringSet` class objects, `sequence_complexity()`, `get_bkg()` and `shuffle_sequences()` are designed to work with such objects. For cases when strings are not `XStringSet` objects, the following functions are available:

- `calc_complexity()`: alternative to `sequence_complexity()`
- `count_klets()`: alternative to `get_bkg()`
- `shuffle_string()`: alternative to `shuffle_sequences()`

```
library(universalmotif)

string <- "DASDSDDSASDSSA"

calc_complexity(string)
#> [1] 0.7823323

count_klets(string, 2)
#>   klets counts
#> 1    AA      0
#> 2    AD      0
#> 3    AS      2
#> 4    DA      1
#> 5    DD      1
#> 6    DS      3
#> 7    SA      2
#> 8    SD      3
#> 9    SS      1

shuffle_string(string, 2)
#> [1] "DDSDSSDSASDASA"
```

A few other utilities have also been made available (based on the internal code of other `universalmotif` functions) that work on simple character vectors:

- `calc_windows()`: calculate the coordinates for sliding windows from 1 to any number `n`
- `get_klets()`: get a list of all possible `k`-lets for any sequence alphabet
- `slide_fun()`: apply a function over sliding windows across a single string
- `window_string()`: retrieve characters from sliding windows of a single string

```
library(universalmotif)

calc_windows(n = 12, window = 4, overlap = 2)
#>   start stop
```

```
#> 1     1    4
#> 2     3    6
#> 3     5    8
#> 4     7   10
#> 5     9   12

get_klets(c("A", "S", "D"), 2)
#> [1] "AA" "AS" "AD" "SA" "SS" "SD" "DA" "DS" "DD"

slide_fun("ABCDEFGH", charToRaw, raw(2), window = 2, overlap = 1)
#>      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
#> [1,]   41   42   43   44   45   46   47
#> [2,]   42   43   44   45   46   47   48

window_string("ABCDEFGH", window = 2, overlap = 1)
#> [1] "AB" "BC" "CD" "DE" "EF" "FG" "GH"
```

Session info

```
#> R version 4.6.1 (2026-06-24)
#> Platform: x86_64-pc-linux-gnu
#> Running under: Ubuntu 24.04.5 LTS
#>
#> Matrix products: default
#> BLAS:   /home/biocbuild/bbs-3.24-bioc/R/lib/libRblas.so
#> LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.12.0  LAPACK version 3.12.0
#>
#> locale:
#>  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
#>  [3] LC_TIME=en_GB              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
#>
#> time zone: America/New_York
#> tzcode source: system (glibc)
#>
#> attached base packages:
#> [1] stats4    stats     graphics  grDevices utils     datasets  methods
#> [8] base
#>
#> other attached packages:
#>  [1] ggbio_1.61.0
#>  [2] TFBSTools_1.51.0
#>  [3] dplyr_1.2.1
#>  [4] ggtree_4.3.0
#>  [5] ggplot2_4.0.3
#>  [6] MotifDb_1.55.0
#>  [7] BSgenome.Athaliana.TAIR.TAIR9_1.3.1000
#>  [8] BSgenome_1.81.1
#>  [9] BiocIO_1.23.3
#> [10] rtracklayer_1.73.0
```

```
#> [11] GenomeInfoDb_1.49.2
#> [12] GenomicRanges_1.65.4
#> [13] Biostrings_2.81.9
#> [14] Seqinfo_1.3.2
#> [15] XVector_0.53.0
#> [16] IRanges_2.47.5
#> [17] S4Vectors_0.51.10
#> [18] BiocGenerics_0.59.13
#> [19] generics_0.1.4
#> [20] universalmotif_1.31.65
#>
#> loaded via a namespace (and not attached):
#>   [1] RColorBrewer_1.1-3          rstudioapi_0.19.0
#>   [3] jsonlite_2.0.0              magrittr_2.0.5
#>   [5] GenomicFeatures_1.65.0      farver_2.1.2
#>   [7] rmarkdown_2.32              fs_2.1.0
#>   [9] vctrs_0.7.3                 memoise_2.0.1
#>  [11] Cairo_1.7-0                 Rsamtools_2.29.0
#>  [13] RCurl_1.98-1.20             base64enc_0.1-6
#>  [15] tinytex_0.61                S4Arrays_1.13.2
#>  [17] htmltools_0.5.9             BiocBaseUtils_1.15.1
#>  [19] curl_8.1.0                  SparseArray_1.13.4
#>  [21] Formula_1.2-6               gridGraphics_0.5-1
#>  [23] htmlwidgets_1.6.4           plyr_1.8.9
#>  [25] cachem_1.1.0                GenomicAlignments_1.49.2
#>  [27] lifecycle_1.0.5             pkgconfig_2.0.3
#>  [29] Matrix_1.7-6                R6_2.6.1
#>  [31] fastmap_1.2.0               MatrixGenerics_1.25.0
#>  [33] digest_0.6.39               aplot_0.3.2
#>  [35] colorspace_2.1-4            TFMPvalue_1.0.0
#>  [37] OrganismDbi_1.55.1          AnnotationDbi_1.75.2
#>  [39] patchwork_1.3.2             Hmisc_5.3-0
#>  [41] RSQLite_3.53.3              seqLogo_1.79.0
#>  [43] labeling_0.4.3              abind_1.4-8
#>  [45] httr_1.4.9                  compiler_4.6.1
#>  [47] bit64_4.8.6                 fontquiver_0.2.1
#>  [49] withr_3.0.3                 backports_1.5.1
#>  [51] htmlTable_2.5.0             S7_0.2.2
#>  [53] BiocParallel_1.47.0         DBI_1.3.0
#>  [55] MASS_7.3-66                 DelayedArray_0.39.8
#>  [57] rappdirs_0.3.4              rjson_0.2.23
#>  [59] gtools_3.9.5                caTools_1.18.4
#>  [61] tools_4.6.1                 splitstackshape_1.4.8.1
#>  [63] foreign_0.8-92              otel_0.3.0
#>  [65] ape_5.8-1                   ggseqlogo_0.2.2
#>  [67] nnet_7.3-21                 glue_1.8.1
#>  [69] restfulr_0.0.17             nlme_3.1-172
#>  [71] grid_4.6.1                  checkmate_2.3.4
#>  [73] cluster_2.1.8.3             reshape2_1.4.5
#>  [75] ade4_1.7-24                 gtable_0.3.6
#>  [77] ensembldb_2.37.3            tidyr_1.3.2
#>  [79] data.table_1.18.6.1         pillar_1.11.1
#>  [81] stringr_1.6.0               yulab.utils_0.2.5
#>  [83] treeio_1.37.1               lattice_0.23-1
```

```
#>  [85] bit_4.6.0                   biovizBase_1.61.0
#>  [87] RBGL_1.89.0                 tidyselect_1.2.1
#>  [89] DirichletMultinomial_1.55.1 fontLiberation_0.1.0
#>  [91] knitr_1.52                  fontBitstreamVera_0.1.1
#>  [93] gridExtra_2.3.1             grImport2_0.3-3
#>  [95] bookdown_0.48               ProtGenerics_1.45.0
#>  [97] SummarizedExperiment_1.43.0 xfun_0.62
#>  [99] Biobase_2.73.2              matrixStats_1.5.0
#> [101] stringi_1.8.9               UCSC.utils_1.9.0
#> [103] lazyeval_0.2.3              ggfun_0.2.2
#> [105] yaml_2.3.12                 evaluate_1.0.5
#> [107] codetools_0.2-20            cigarillo_1.3.1
#> [109] gdtools_0.5.3               tibble_3.3.1
#> [111] graph_1.91.0                BiocManager_1.30.27
#> [113] ggplotify_0.1.4             cli_3.6.6
#> [115] rpart_4.1.27                systemfonts_1.3.2
#> [117] dichromat_2.0-1             Rcpp_1.1.2
#> [119] png_0.1-9                   XML_3.99-0.25
#> [121] parallel_4.6.1              blob_1.3.0
#> [123] jpeg_0.1-11                 AnnotationFilter_1.37.0
#> [125] bitops_1.1-0                pwalign_1.9.1
#> [127] viridisLite_0.4.3           tidytree_0.4.8
#> [129] VariantAnnotation_1.59.4    ggiraph_0.9.6
#> [131] scales_1.4.0                motifStack_1.57.0
#> [133] purrr_1.2.2                 crayon_1.5.3
#> [135] rlang_1.3.0                 KEGGREST_1.53.6
```

References

Altschul, Stephen F., and Bruce W. Erickson. 1985. "Significance of Nucleotide Sequence Alignments: A Method for Random Sequence Permutation That Preserves Dinucleotide and Codon Usage." *Molecular Biology and Evolution* 2: 526–38.

Bailey, T. L., and C. Elkan. 1994. "Fitting a Mixture Model by Expectation Maximization to Discover Motifs in Biopolymers." *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology* 2: 28–36.

Fitch, Walter M. 1983. "Random Sequences." *Journal of Molecular Biology* 163: 171–76.

Jiang, M., J. Anderson, J. Gillespie, and M. Mayne. 2008. "uShuffle: A Useful Tool for Shuffling Biological Sequences While Preserving k-Let Counts." *BMC Bioinformatics* 9.

Morgulis, A., E. M. Gertz, A. A. Schaffer, and R. Agarwala. 2006. "A Fast and Symmetric DUST Implementation to Mask Low-Complexity DNA Sequences." *Journal of Computational Biology* 13: 1028–40.

Noble, William S. 2009. "How Does Multiple Testing Correction Work?" *Nature Biotechnology* 27: 1135–37.

Orlov, Y. L., and V. N. Potapov. 2004. "Complexity: An Internet Resource for Analysis of DNA Sequence Complexity." *Nucleic Acids Research* 32: W628–33.

Propp, J. G., and D. W. Wilson. 1998. "How to Get a Perfectly Random Sample from a Generic Markov Chain and Generate a Random Spanning Tree of a Directed Graph." *Journal of Algorithms* 27: 170–217.

Trifonov, E. N. 1990. "Making Sense of the Human Genome." In *Structure & Methods*, edited by R. H. Sarma, 69–77. Albany: Adenine Press.

Wootton, J. C., and S. Federhen. 1993. "Statistics of Local Complexity in Amino Acid Sequences and Sequence Databases." *Computers & Chemistry* 17: 149–63.