A short introduction to acoustic template matching with `monitoR`

Sasha D. Hafner
Aarhus University, Aarhus, Denmark (sasha.hafner@eng.au.dk)

and

Jonathan Katz
Vermont Cooperative Fish and Wildlife Research Unit, University of Vermont, USA (jonkatz4@gmail.com)

October 31, 2024

Disclaimer

"Although this software program has been used by the U.S. Geological Survey (USGS), no warranty, expressed or implied, is made by the USGS or the U.S. Government as to the accuracy and functioning of the program and related program material nor shall the fact of distribution constitute any such warranty, and no responsibility is assumed by the USGS in connection therewith."

1 Getting started

The motivation behind the development of the `monitoR` package was automated detection and identification of animal vocalizations. Here, we describe how to use `monitoR` functions to detect and identify songs from two bird species from a survey recording. We'll start by loading the package.

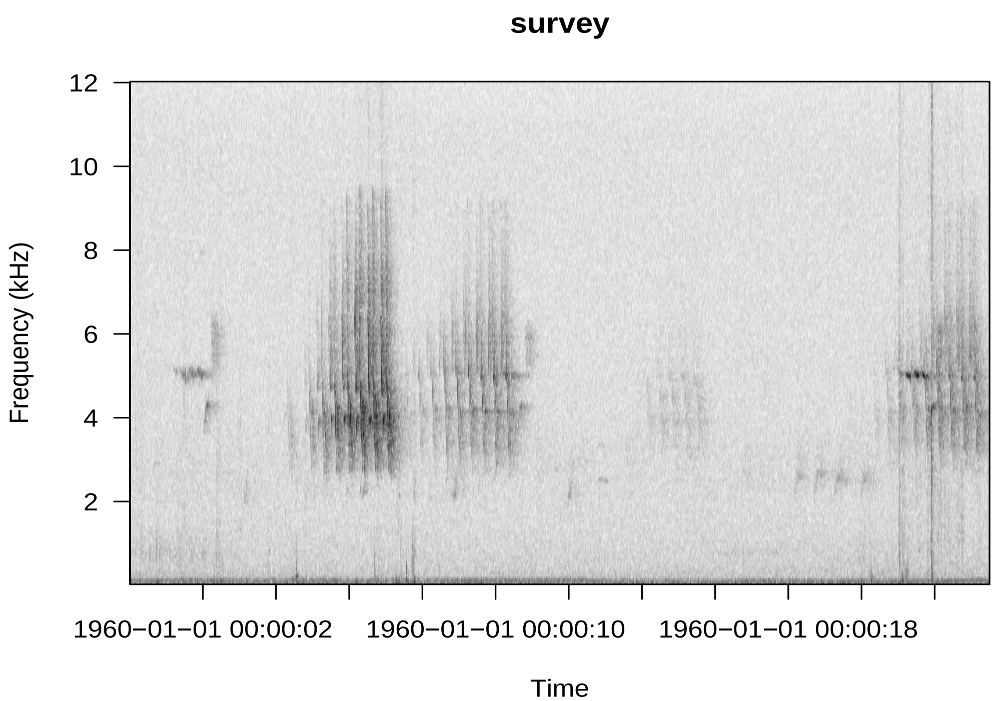
```
library(monitoR)
```

The `monitoR` package includes a 23 second recording of songbirds as a Wave object (defined in the `tuneR` package) named `survey`.

```
data(survey)
survey


#
# Wave Object
#  Number of Samples:      564000
#  Duration (seconds):     23.5
#  Samplingrate (Hertz):   24000
#  Channels (Mono/Stereo): Mono
#  PCM (integer format):   TRUE
#  Bit (8/16/24/32/64):    16
```

A spectrogram of the recordings can be generated using the `viewSpec` function[1].

```
viewSpec(survey)

# Warning in as.character.POSIXt(as.POSIXct(t.bin.ticks + start.time, origin = "1960-01-01",
:  as.character(td, ..)  no longer obeys a 'format' argument; use format(td, ..)  ?
```



We could also play the recording using the `play` function from the tuneR package[2].

```
setWavPlayer("play")
play(survey)
```

You might be see two different types of songs here. The songs around 2, 10, and maybe 22 seconds are from a black-throated green warbler *Setophaga virens*, and there are ovenbird *Seiurus aurocapilla* songs around 6, 9, 15, and 23 seconds. In the following, we show how `monitoR` can be used to detect and identify these songs.

The `monitoR` package also includes short recordings of songs from the same two species in `survey`: `btnw`, which contains a single song from a black-throated green warbler, and `oven` which contains a single song from an ovenbird. As with `survey`, these objects are `Wave` objects.

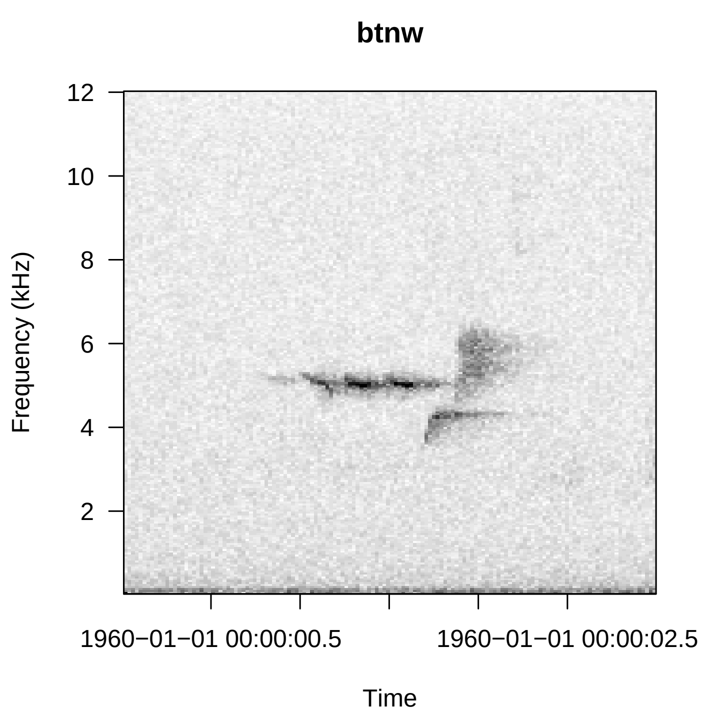
```
data(btnw)
data(oven)
btnw
```

[1] By default, this function just transforms the data and generates a spectrogram. Try setting the `interactive` argument to `TRUE` for more viewing options that are useful for longer recordings.

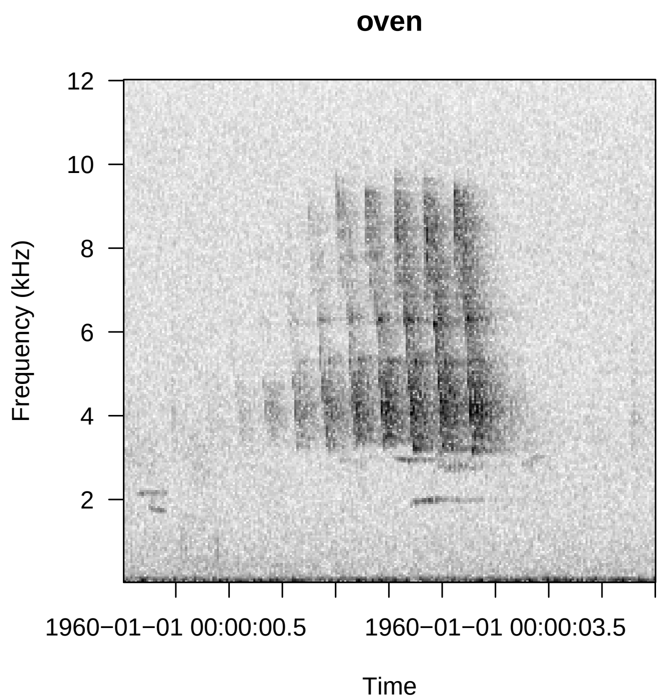
[2] This requires that a command-line wav player is installed. Here, we are using SoX.

```
#
# Wave Object
#   Number of Samples:      72001
#   Duration (seconds):     3
#   Samplingrate (Hertz):   24000
#   Channels (Mono/Stereo): Mono
#   PCM (integer format):   TRUE
#   Bit (8/16/24/32/64):    16

oven

#
# Wave Object
#   Number of Samples:      120001
#   Duration (seconds):     5
#   Samplingrate (Hertz):   24000
#   Channels (Mono/Stereo): Mono
#   PCM (integer format):   TRUE
#   Bit (8/16/24/32/64):    16
```

```
viewSpec(btnw)

# Warning in as.character.POSIXt(as.POSIXct(t.bin.ticks + start.time, origin = "1960-01-01",
:  as.character(td, ..)  no longer obeys a 'format' argument; use format(td, ..)  ?
```



```
viewSpec(oven)

# Warning in as.character.POSIXt(as.POSIXct(t.bin.ticks + start.time, origin = "1960-01-01",
:  as.character(td, ..)  no longer obeys a 'format' argument; use format(td, ..)  ?
```



As `Wave` objects, all of these recordings can be used directly in the template matching functions[3]. Users will typically work with wav files, however, as opposed to mp3 files, or `Wave` objects. To be consistent with this typical approach, we will save the two clips and the survey as wav files[4]. We'll include a made-up date in the name for the survey file, since that is one way to determine "absolute" times for detections[5].

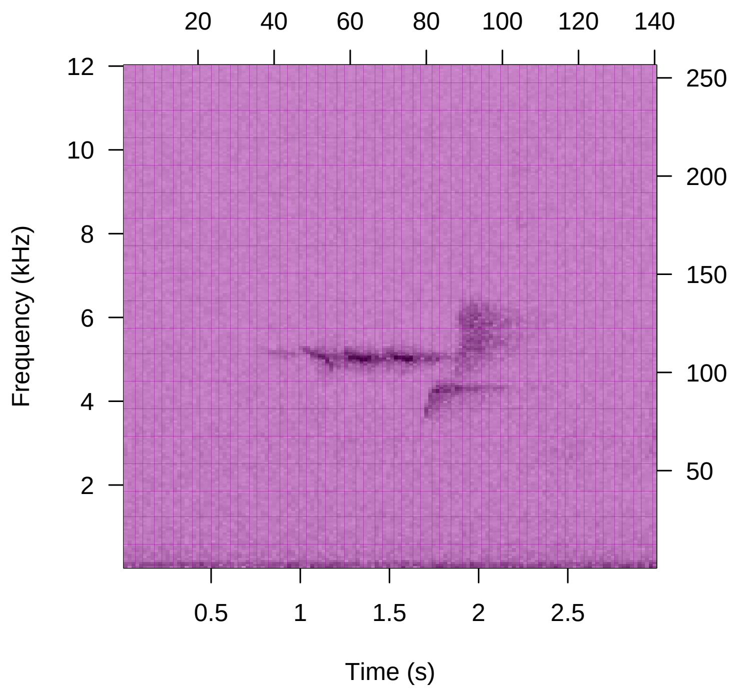
```
btnw.fp <- file.path(tempdir(), "btnw.wav")
oven.fp <- file.path(tempdir(), "oven.wav")
survey.fp <- file.path(tempdir(), "survey2010-12-31_120000_EST.wav")
writeWave(btnw, btnw.fp)
writeWave(oven, oven.fp)
writeWave(survey, survey.fp)
```

[3]But you must set the `write.wav` argument to `TRUE` to do this.
[4]Here, we will use a temporary directory, but for your own recordings a more permanent location is better.
[5]The other option is to use the modification date of the survey file, which clearly would be wrong here, since we just created it.

2 Spectrogram cross correlation

The `monitoR` package provides functions for two kinds of template detection–spectrogram cross correlation and binary point matching. We'll start with the correlation approach, which is slightly simpler. The easiest way to create a correlation template is to call up `makeCorTemplate` with only the `clip` argument specified. This function carries out a Fourier transform, and, by default, saves amplitude (volume) data for each cell within the resulting frequency-by-time matrix.

```
wct1 <- makeCorTemplate(btnw.fp)
```




```
#
# Automatic point selection.
#
# Done.
```

The `makeCorTemplate` function displays a spectrogram of the recording[6], and shows the cells included in the template (all the cells here) in transparent purple[7]. Let's look at a summary of the template we just created[8].

```
wct1


#
```

[6] This plot is more important when templates are produced interactively, as described below.
[7] This color can be selected with the argument `sel.col`.
[8] Just entering the template list name will call up the appropriate `show` method, which just displays a simple summary.

```
# Object of class "corTemplateList"
#  containing  1  templates
#         original.recording sample.rate lower.frequency
# A /tmp/RtmpUZest1/btnw.wav       24000           0.047
#   upper.frequency lower.amp upper.amp duration n.points
# A              12    -97.52         0    2.965    35840
#   score.cutoff
# A          0.4
```

The `w1` object is a template list, of class `corTemplateList`. Here the idea of a list doesn't make a lot of sense, since we only have one template, but in monitoR, templates only exist as part of template lists. Below, we'll create template lists with multiple templates.

The name associated with our single template in the list is `A`, which is the default, and is not very descriptive. The name of a template can be set when it is created, using the `name` argument[9][10].

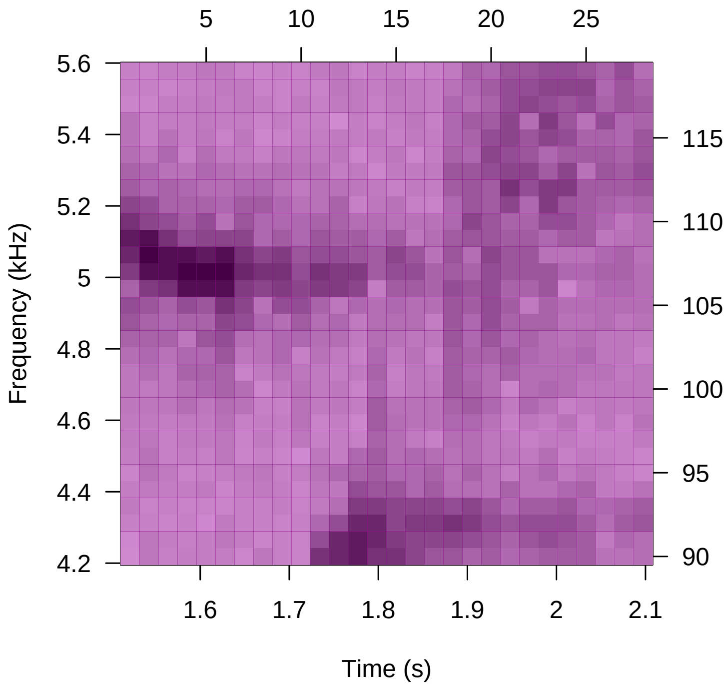
```
wct1 <- makeCorTemplate(btnw.fp, name = "w1")

#
# Automatic point selection.
#
# Done.

wct1

#
# Object of class "corTemplateList"
#  containing  1  templates
#          original.recording sample.rate lower.frequency
# w1 /tmp/RtmpUZest1/btnw.wav       24000           0.047
#    upper.frequency lower.amp upper.amp duration n.points
# w1              12    -97.52         0    2.965    35840
#    score.cutoff
# w1          0.4
```

This template is functional, but it does not take advantage of all the options available in `makeCorTemplate`. Setting time and frequency limits is one of simplest, and may be necessary when making a template from a long recording. We'll make a new template that uses the `t.lim` and `frq.lim` arguments to focus on a particular section within the song.

[9] The `templateNames` function can be used to change template names in an existing template list. For more detailed information, the `templateComment` function can be used to check and add comments to templates.

[10] In the interest of keeping this file from being too big, the figures produced by the calls in most of the following examples will not be shown. For a complete version of this vignette, visit the monitoR website.

```
wct2 <- makeCorTemplate(btnw.fp, t.lim = c(1.5, 2.1), frq.lim = c(4.2, 5.6), name = "w2")
```




```
#
# Automatic point selection.
#
# Done.

wct2

#
# Object of class "corTemplateList"
#  containing  1  templates
#           original.recording sample.rate lower.frequency
# w2 /tmp/RtmpUZest1/btnw.wav       24000           4.219
#    upper.frequency lower.amp upper.amp duration n.points
# w2           5.578    -93.52         0    0.576      840
#    score.cutoff
# w2          0.4
```

Let's make two templates with the ovenbird recording as well.

```
oct1 <- makeCorTemplate(oven.fp, t.lim = c(1, 4), frq.lim = c(1, 11), name = "o1")

#
# Automatic point selection.
```

```
#
# Done.
```

For the next template, we'll use the `dens` argument to reduce the size of the template–that is, the number of points included.

```
oct2 <- makeCorTemplate(oven.fp, t.lim = c(1, 4), frq.lim = c(1, 11), dens = 0.1, name = "o2")

#
# Automatic point selection.
#
# Done.
```

Here, our template will include *approximately* one-tenth of all the spectrogram points.

Template lists with just a single template, can be used for detection. However, `monitoR` is designed to use template lists that contain multiple templates. Templates can be combined together in a single list with `combineCorTemplates`.

```
ctemps <- combineCorTemplates(wct1, wct2, oct1, oct2)
ctemps

#
# Object of class "corTemplateList"
#  containing  4  templates
#          original.recording sample.rate lower.frequency
# w1 /tmp/RtmpUZest1/btnw.wav       24000           0.047
# w2 /tmp/RtmpUZest1/btnw.wav       24000           4.219
# o1 /tmp/RtmpUZest1/oven.wav       24000           1.031
# o2 /tmp/RtmpUZest1/oven.wav       24000           1.031
#    upper.frequency lower.amp upper.amp duration n.points
# w1          12.000    -97.52      0.00    2.965    35840
# w2           5.578    -93.52      0.00    0.576      840
# o1          10.969    -85.94      0.00    2.965    29820
# o2          10.969    -71.29     -2.64    2.965     2970
#    score.cutoff
# w1          0.4
# w2          0.4
# o1          0.4
# o2          0.4
```

The output from `combineCorTemplates` is a template list, just like the output from `makeCorTemplate`. This function can be used to combine templates present in any number of correlation template lists (and each of the lists can contain any number of templates).

We can view all the templates in `ctemps` with the `plot` function.

```
plot(ctemps)
```

We can now use our templates to look for black-throated green warbler and ovenbird songs within the `survey` recording. This is a three-step process: we first calculate correlation score between the templates and each time bin in the survey, then identify "peaks" in the scores, and finally determine which, if any, score peaks exceed the score cutoff. Correlation scores are calculated with `corMatch`.

```
cscores <- corMatch(survey.fp, ctemps)

#
# Starting  w1 . . .
#  Fourier transform on survey . . .
#  Continuing. . .
#
#  Done.
#
# Starting  w2 . . .
#  Done.
#
# Starting  o1 . . .
#  Done.
#
# Starting  o2 . . .
#  Done.
```

This particular example should run in a few seconds, but `corMatch` can take much longer for long surveys [11] . We can get a summary of the `scores` object by entering its name, but it requires additional processing in order to be used for detecting songs.

```
cscores

#
# A "templateScores" object
#
# Based on the survey file:  /tmp/RtmpUZest1/survey2010-12-31_120000_EST.wav
#
# And  4  templates
# Score information
#    min.score max.score n.scores
# w1      0.13      0.46      961
# w2     -0.22      0.73     1073
# o1     -0.07      0.63      961
# o2     -0.10      0.66      961
```

The `corMatch` function returns `templateScores` objects, which contain the correlation scores for each template, but also the complete original templates, along with other information, like the name of the survey file used. To make detections, we need to look within the correlation scores returned by `corMatch` for peaks, and then identify those peaks with values above the score cutoff as detections. These two steps are carried out with `findPeaks`.

```
cdetects <- findPeaks(cscores)

#
```

[11] Set the `parallel` argument to `TRUE` to speed this step up if you are have a multi-core processor and are not using Windows.

```
# Done with  w1
# Done with  w2
# Done with  o1
# Done with  o2
# Done
```

The `findPeaks` function returns `detectionList` objects, which contain all the information originally present in `templateScores` objects, plus detection results. To see a summary of the detections, just enter the name of the object.

```
cdetects

#
# A "detectionList" object
#
# Based on survey file:  /tmp/RtmpUZest1/survey2010-12-31_120000_EST.wav
#
# and  4  templates
#
# Detection information
#    n.peaks n.detections min.peak.score max.peak.score
# w1       6            1    0.297370075      0.4615955
# w2      38            3    0.009121254      0.7291718
# o1       5            3    0.154263730      0.6322787
# o2       5            3    0.210719516      0.6566550
#    min.detection.score max.detection.score
# w1           0.4615955           0.4615955
# w2           0.4057619           0.7291718
# o1           0.4562568           0.6322787
# o2           0.4813885           0.6566550
```

From the `n.peaks` column, we can see there are from five to 38 peaks per template, and from the `n.detections` column, we can see that the templates resulted in from one to three detections. The detections can be extracted with the `getDetections` function.

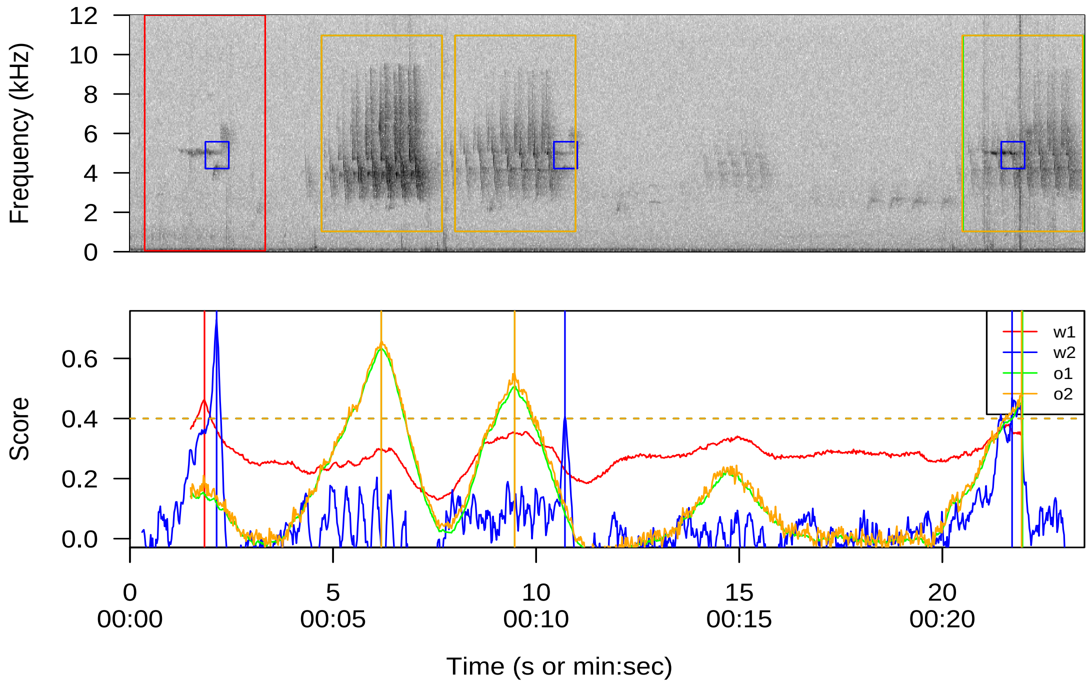
```
getDetections(cdetects)

#    template           date.time      time     score
# 1        w1 2010-12-31 11:59:38  1.834667 0.4615955
# 2        w2 2010-12-31 11:59:38  2.133333 0.7291718
# 3        w2 2010-12-31 11:59:47 10.709333 0.4057619
# 4        w2 2010-12-31 11:59:58 21.717333 0.4691309
# 5        o1 2010-12-31 11:59:42  6.186667 0.6322787
# 6        o1 2010-12-31 11:59:45  9.472000 0.5078371
# 7        o1 2010-12-31 11:59:58 21.973333 0.4562568
# 8        o2 2010-12-31 11:59:42  6.186667 0.6566550
# 9        o2 2010-12-31 11:59:45  9.472000 0.5511877
# 10       o2 2010-12-31 11:59:58 21.952000 0.4813885
```

Whether or not a peak qualifies as a detection depends a parameter called the score cutoff, which can be set separately for each template. If we look at our template list again, we can see the score cutoffs given in the last column of the summary.

```
ctemps

#
# Object of class "corTemplateList"
#  containing  4  templates
#          original.recording sample.rate lower.frequency
# w1 /tmp/RtmpUZest1/btnw.wav       24000           0.047
# w2 /tmp/RtmpUZest1/btnw.wav       24000           4.219
# o1 /tmp/RtmpUZest1/oven.wav       24000           1.031
# o2 /tmp/RtmpUZest1/oven.wav       24000           1.031
#    upper.frequency lower.amp upper.amp duration n.points
# w1          12.000    -97.52      0.00    2.965    35840
# w2           5.578    -93.52      0.00    0.576      840
# o1          10.969    -85.94      0.00    2.965    29820
# o2          10.969    -71.29     -2.64    2.965     2970
#    score.cutoff
# w1          0.4
# w2          0.4
# o1          0.4
# o2          0.4
```

They are currently set to the default value, but there is no reason to think that this value should work for all or even any templates. Instead, score cutoffs need to be determined based on template performance, typically with a subset of the survey or surveys that will ultimately be searched for matching sounds. Here we just have one, very short, survey, so we'll use the entire survey to determine what the score cutoff should be set to for each template. We can see a graphical summary of the results with the generic `plot` function.

```
plot(cdetects)

# 0 to 23.5 seconds
```



Starting with the warbler templates, it is easy to see that template `w2` is much better than template `w1`–its scores are high for the three apparent songs, and relatively low elsewhere, including during ovenbird songs. Changing the score cutoff cannot much improve template `w1`. For template `w2`, the default value works well enough for this survey. But to incorporate a safety factor, we might want to lower it, perhaps to 0.30.

Looking at the ovenbird results, both templates perform similarly, but if we want to detect songs like the faint one around 15 sec, the score cutoff needs to be reduced, perhaps to 0.20.

We could recreate this template list by repeating the steps above, but set the `score.cutoff` argument for `w2` to 0.30 and 0.20 for `o1` and `o2`. Fortunately, using `templateCutoff` is a more efficient option. It can be used as an extractor or replacement function.

```
templateCutoff(ctemps)

#  w1  w2  o1  o2
# 0.4 0.4 0.4 0.4
```

To change the cutoff for templates w2, o1, and o2 in the the template list `cdetects`, we can use:

```
templateCutoff(ctemps)[2:4] <- c(0.3, 0.2, 0.2)
```

Or, this call does the same thing (and is perhaps less prone to mistakes):

```
templateCutoff(ctemps) <- c(w2 = 0.3, o1 = 0.2, o2 = 0.2)
ctemps

#
```

```
# Object of class "corTemplateList"
#  containing  4  templates
#          original.recording sample.rate lower.frequency
# w1 /tmp/RtmpUZest1/btnw.wav       24000           0.047
# w2 /tmp/RtmpUZest1/btnw.wav       24000           4.219
# o1 /tmp/RtmpUZest1/oven.wav       24000           1.031
# o2 /tmp/RtmpUZest1/oven.wav       24000           1.031
#    upper.frequency lower.amp upper.amp duration n.points
# w1          12.000    -97.52      0.00    2.965    35840
# w2           5.578    -93.52      0.00    0.576      840
# o1          10.969    -85.94      0.00    2.965    29820
# o2          10.969    -71.29     -2.64    2.965     2970
#    score.cutoff
# w1          0.4
# w2          0.3
# o1          0.2
# o2          0.2
```

Or, we could even use the special name `default` to specify a default value.

```
templateCutoff(ctemps) <- c(w2 = 0.3, default = 0.2)
ctemps

#
# Object of class "corTemplateList"
#  containing  4  templates
#          original.recording sample.rate lower.frequency
# w1 /tmp/RtmpUZest1/btnw.wav       24000           0.047
# w2 /tmp/RtmpUZest1/btnw.wav       24000           4.219
# o1 /tmp/RtmpUZest1/oven.wav       24000           1.031
# o2 /tmp/RtmpUZest1/oven.wav       24000           1.031
#    upper.frequency lower.amp upper.amp duration n.points
# w1          12.000    -97.52      0.00    2.965    35840
# w2           5.578    -93.52      0.00    0.576      840
# o1          10.969    -85.94      0.00    2.965    29820
# o2          10.969    -71.29     -2.64    2.965     2970
#    score.cutoff
# w1          0.2
# w2          0.3
# o1          0.2
# o2          0.2
```

And now we could call `corMatch` and `findPeaks` again. This approach would work fine, but takes more time and effort than is needed, because it isn't necessary to recalculate the scores, since they are independent of the cutoffs. Instead, we can just change the score cutoffs in our existing `detectionList` object `cdetects`, with the same function `templateCutoff`. The detections are automatically updated when score cutoffs are changed in a `detectionList` object.

```
templateCutoff(cdetects) <- c(w2 = 0.3, default = 0.2)
```

So it isn't even necessary to change `ctemps`[12]. Now let's take a look at `cdetects`.

```
cdetects

#
# A "detectionList" object
#
# Based on survey file:  /tmp/RtmpUZest1/survey2010-12-31_120000_EST.wav
#
# and  4  templates
#
# Detection information
#    n.peaks n.detections min.peak.score max.peak.score
# w1       6            6    0.297370075      0.4615955
# w2      38            3    0.009121254      0.7291718
# o1       5            4    0.154263730      0.6322787
# o2       5            5    0.210719516      0.6566550
#    min.detection.score max.detection.score
# w1           0.2973701           0.4615955
# w2           0.4057619           0.7291718
# o1           0.2261725           0.6322787
# o2           0.2107195           0.6566550
```

Notice how the number of detections has changed for `o1` and `o2`[13].

```
plot(cdetects)

# 0 to 23.5 seconds
```

Since the template `w1` is nearly useless, and template `o1` and `o2` nearly identical, we might want to drop `w1` and either `o1` or `o2` from our results. This can be done using indexing[14].

```
cdetects <- cdetects[c("w2", "o2")]
cdetects

#
# A "detectionList" object
#
# Based on survey file:  /tmp/RtmpUZest1/survey2010-12-31_120000_EST.wav
#
# and  2  templates
#
# Detection information
#    n.peaks n.detections min.peak.score max.peak.score
# w2      38            3    0.009121254      0.7291718
# o2       5            5    0.210719516      0.6566550
#    min.detection.score max.detection.score
```

[12]Of course, you may want to reuse or save the template list with optimized cutoffs. The `getTemplates` function can be helpful here–it can be used to extract the templates from a `detectionList` object.

[13]The plot produced by the call below is omitted, but it would now show a different number of detections.

[14]Here, the omitted plot would only show results for w2 and o2.

```
# w2           0.4057619           0.7291718
# o2           0.2107195           0.6566550
```

```
plot(cdetects)

# 0 to 23.5 seconds
```

In this short example, it is easy to verify results in the results using the above plot. For longer surveys, for which these methods make more sense, the `showPeaks` function is more efficient. It can be used to view all detections or even all peaks, one by one, and by setting the `verify` argument to `TRUE`, it can be used to save results of the verification process.

This simple example doesn't show all the available options for creating templates. In particular, with the `select` argument set to `"rectangle"`, or `"cell"`, it is possible to select areas or even individual cells to be included in a template.

To summarize, for template matching using spectrogram cross-correlation one would typically use the following functions in the order given:

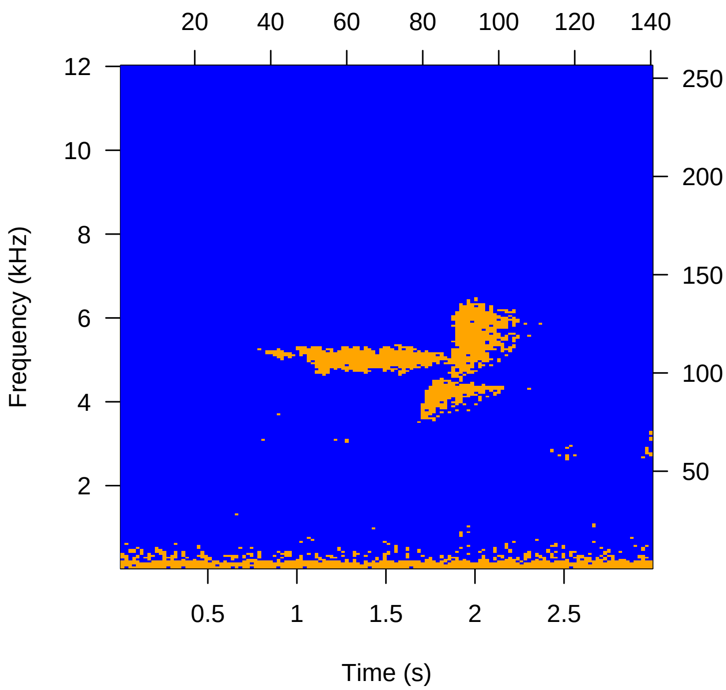
1. `makeCorTemplate` to make the template(s)
2. `combineCorTemplates` to combine templates together in a single template list
3. `corMatch` to calculate scores
4. `findPeaks` to find peaks and identify detections
5. `plot` to see the scores and detections
6. `getDetections` to get the (numeric) detection results
7. `templateCutoff` to change template cutoffs in the detection list object (iteratively with `plot` and `getDetections`)

3 Binary point matching

We can use binary point matching to carry out a procedure similar to the one described above in Section 2. Because the type of data needed to calculate scores differ, binary templates differ from correlation templates, and the corresponding functions used for creating templates also differ in some ways. The `makeBinTemplate` function converts the time-domain data into a binary format by default– i.e., cells are identified as either "high" or "low" (we'll refer to these as "on" and "off") depending on whether they are greater than or less than the user-set `amp.cutoff`, respectively. The value of `amp.cutoff` is set interactively by default [15]. With the default option, the spectrogram is updated each time the value of `amp.cutoff` changes. Here, we'll set the cutoff directly in the function call, based on earlier trials.

```
wbt1 <- makeBinTemplate(btnw.fp, amp.cutoff = -40, name = "w1")
```

[15] This can be changed with the `amp.cutoff` argument.




```
#
# Automatic point selection.
#
# Done.
```

Time and frequency limits can be set as with `makeCorTemplate`. We'll also include a buffer around the "on" cells–cells in the buffer are excluded from the template[16].

```
wbt2 <- makeBinTemplate(btnw.fp, amp.cutoff = -30, t.lim = c(1.5, 2.1),
                        frq.lim = c(4.2, 5.6), buffer = 2, name = "w2")

#
# Automatic point selection.
#
# Done.
```

The `amp.cutoff` and `buffer` values used above are based on trial-and-error not shown here.

Let's also make two ovenbird templates, with and without a buffer.

```
obt1 <- makeBinTemplate(oven.fp, amp.cutoff = -20, t.lim = c(1, 4),
                        frq.lim = c(1, 11), name = "o1")

#
```

[16]Although the resulting plot isn't shown here, again, to keep this file size down.

```
# Automatic point selection.
#
# Done.

obt2 <- makeBinTemplate(oven.fp, amp.cutoff = -17, t.lim = c(1, 4),
                        frq.lim = c(1, 11), buffer = 2, name = "o2")

#
# Automatic point selection.
#
# Done.
```

Binary templates are combined with `combineBinTemplates`.

```
btemps <- combineBinTemplates(wbt1, wbt2, obt1, obt2)
btemps

#
# Object of class "binTemplateList"
#
#  containing  4  templates
#          original.recording sample.rate lower.frequency
# w1 /tmp/RtmpUZest1/btnw.wav       24000        0.046875
# w2 /tmp/RtmpUZest1/btnw.wav       24000        4.218750
# o1 /tmp/RtmpUZest1/oven.wav       24000        1.031250
# o2 /tmp/RtmpUZest1/oven.wav       24000        1.031250
#    upper.frequency duration on.points off.points score.cutoff
# w1       12.000000     2.97      1845      33995           12
# w2        5.578125     0.58       223        310           12
# o1       10.968750     2.97      2676      27144           12
# o2       10.968750     2.97      1607      23833           12
```

Notice how binary templates have "on" points and "off" points, while correlation templates only have one type of point. Scores are calculated using `binMatch`, which is analogous to `corMatch`. As with `corMatch`, the output from this function is a `templateScores` object.

```
bscores <- binMatch(survey.fp, btemps)

#
# Starting  w1 . . .
#
#  Fourier transform on survey . . .
#  Continuing. . .
#
#  Done.
#
# Starting  w2 . . .
#
#  Done.
#
```

```
# Starting  o1 . . .
#
#  Done.
#
# Starting  o2 . . .
#
#  Done.
```

From this point on, objects created by binary point matching are identical to those made with spectrogram cross correlation. To find detections, use the same `findPeaks` function that we used above.

```
bdetects <- findPeaks(bscores)

#
# Done with  w1
# Done with  w2
# Done with  o1
# Done with  o2
# Done
```

And, as above, we can use the `plot` method for `detectionList` objects to view detections.

```
plot(bdetects)

# 0 to 23.5 seconds
```

Based on these results, it looks like all templates except w1 could be useful with the right score cutoff. We might want to use a cutoff of about 7 for `w2`, and perhaps 4 for `o1` and `o2` if the song around 15 seconds should be detected. First, let's drop `w1`[17].

```
bdetects <- bdetects[-1]
```

```
templateCutoff(bdetects) <- c(w2 = 7, default = 4)
```

```
plot(bdetects)

# 0 to 23.5 seconds
```

Finally, the detections are:

```
getDetections(bdetects)

#    template           date.time      time     score
# 1        w2 2010-12-31 11:59:38  2.133333 17.583758
# 2        w2 2010-12-31 11:59:47 10.730667  8.669633
# 3        w2 2010-12-31 11:59:58 21.717333 10.490479
```

[17] The plot that is omitted below would no longer show any results for w1.

```
# 4        o1 2010-12-31 11:59:42  6.208000 18.757236
# 5        o1 2010-12-31 11:59:45  9.450667 11.775177
# 6        o1 2010-12-31 11:59:51 14.698667  4.199115
# 7        o1 2010-12-31 11:59:58 21.952000 10.884734
# 8        o2 2010-12-31 11:59:38  1.536000  4.019856
# 9        o2 2010-12-31 11:59:42  6.208000 22.261260
# 10       o2 2010-12-31 11:59:45  9.472000 14.209354
# 11       o2 2010-12-31 11:59:51 14.720000  5.351632
# 12       o2 2010-12-31 11:59:58 21.952000 13.189866
```

As with correlation templates, there is much more flexibility in making binary templates than this example suggests. In particular, options for the `select` should be explored. The most flexible is `"cell"`, but it is certainly not the quickest and because individual cells are selected manually, may not be readily repeatable.

To summarize, for template matching using binary point matching one would typically use the following functions in the order given:

1. `makeBinTemplate` to make the template(s)
2. `combineBinTemplates` to combine templates together in a single template list
3. `binMatch` to calculate scores
4. `findPeaks` to find peaks and identify detections
5. `plot` to see the scores and detections
6. `getDetections` to get the (numeric) detection results
7. `templateCutoff` to change template cutoffs in the detection list object (iteratively with `plot` and `getDetections`)

4 Other functions

The `monitoR` package includes several functions not described in this short introduction.

- For manipulating recordings directly, there are the `cutWave`[18], `changeSampRate`, and `mp3Subsamp` functions.
- With the `plot` methods for template list and detection list objects (used in sections 2 and 3 above), color palettes and other characteristics of the plots can be modified.
- The `viewSpec` function is capable of much more than the above examples suggest. It can be used interactively to view different parts of a spectrogram, or even annotate it.
- Comments can be added to templates within a template list using the `templateComment` functions.
- Binary and correlation templates were designed to be portable, and can be saved to text files using `writeCorTemplates` or `writeBinTemplates`. Existing template files can be read in with `readCorTemplates` and `writeCorTemplates`.

[18] Originally named `cutw`, but recently changed.

- For verification of detections, there is the `showPeaks` function, which, shows (or plays) individual detections (or peaks), and, in the interactive mode, allows the user to add verification data to a detection list object.
- The functions `collapseClips` and `bindEvents` can be used to extract and bind together detections from a longer recording, and could also be useful for verification.
- The function `eventEval` provides a way to compare results from template detection to a manually annotated spectrogram, and the function `timeAlign` removes redundant detections from a survey (e.g. those from multiple templates).
- The `compareTemplates` function can be used to compare the performance of multiple templates and evaluate similarity between templates.
- An alternative to writing `templateLists` and `detectionLists` locally is to store templates, survey metadata, and detection results in a MySQL database. MonitoR has a variety of MySQL queries using package `RODBC` to push and pull data to an acoustics database: uploads are accomplished with `dbUploadSurvey`, `dbUploadResult`, and `dbUploadTemplate`. Templates can be downloaded to `templateList` objects with `dbDownloadTemplate`, and results can be downloaded to `detectionList` objects with `dbDownloadResult`. Survey metadata and media card/recorder metadata are downloaded with `dbDownloadSurvey` and `dbDownloadCardRecorderID`. An acoustics database schema is available for download at the monitoR website, which can be loaded on an active database instance using `dbSchema` or incorporated into an existing MySQL database.

Acknowledgements

Generous funding for this work was provided by the National Park Service, the U.S. Geological Survey, and the National Phenology Network.