RTISI-LA Spectrogram inversion

J.P. Meagher

16 November 2017

Outlined here is the R implementation of the RTISI-LA (Zhu, Beauregard, and Wyse 2007) method of time domain signal estimation from spectrograms.

Packages

I think that I may try to spin this out into a standalone package and so I will make every effort to implement this algorithm in base R. I will use some tidyverse packages for data exploration and smooth functional programming.

```
library(tidyverse)
library(magrittr)
library(audio)
```

Testing Data

This algorithim is being written for the inversion of ancestrally reconstructed spectrograms, so it will be developed using data of the same structure.

The end goal is to be able to take the preprocessed spectrogram and transform it into a time domain signal for which the sound is similar to that of the original call recording (see Figure 1).

Signal to Spectrogram

The `signal` package provides some tools for the time frequency analysis of acoustic signals however some of its functions are very unsatisfactory. There are also a number of intermediate aspects of the spectrogram inversion process that are required. Thus an effort will be made to build all the functions required to both produce spectrograms and invert them.

Hamming Window

The Hamming window is a window function commonly applied to the segments of signals in a Short Time Fourier Transform. The function is reproduced here for this analysis. Other windowing functions are available in the `signal` package.

```
hamming <- function(n){
    if (!(n == round(n) && n > 0))
        stop("hamming: n has to be an integer > 0")
    if (n == 1)
        w = 1
    else {
        n = n - 1
        w = 0.54 - 0.46 * cos(2 * pi * (0:n)/n)
```

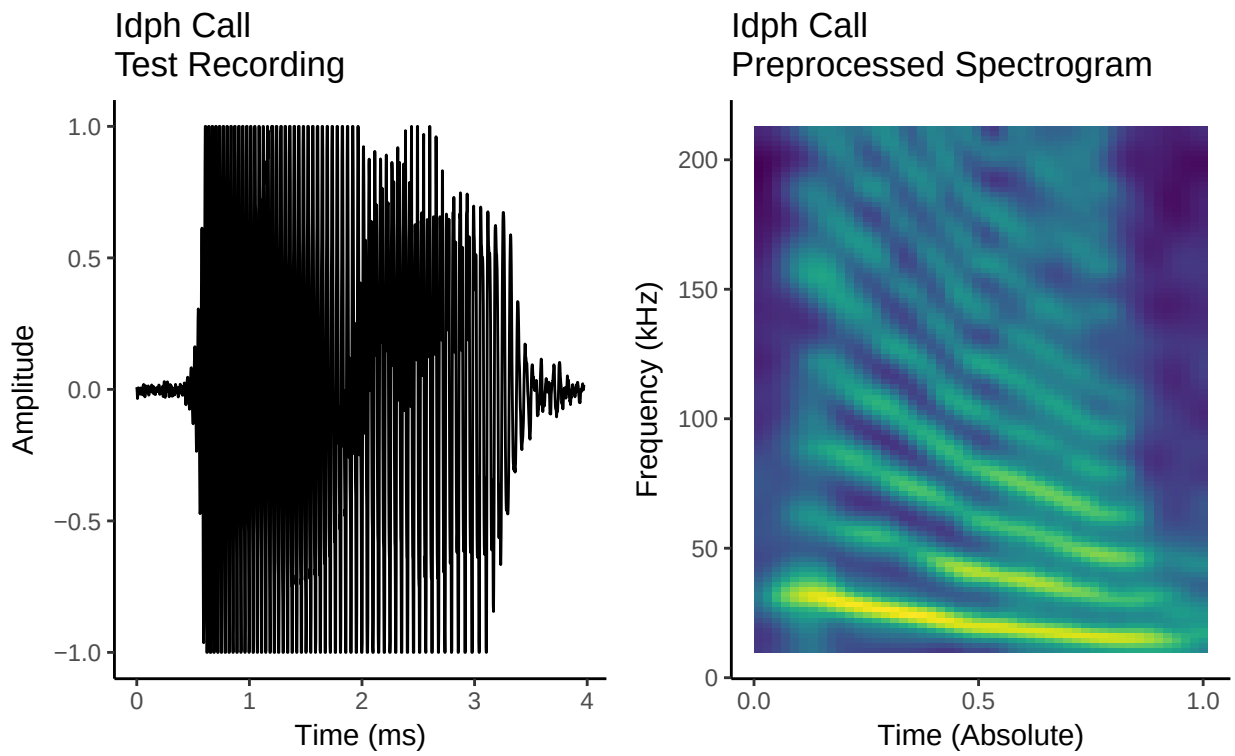

Figure 1: A visualisation of the data randomly selected for development of the algorithm

```
    }
    w
}
```

Complex STFT

The first step in obtaining a spectrogram for inversion is to calculate the short time fourier transfor of a signal. This produces a complex matrix.

```
complex_stft <- function(x, window_size = min(256, length(x)),
  window_function = hamming(window_size), step_size = ceiling(window_size/2)){
  if (!is.numeric(x)) stop("'x' has to be a numeric.")

  if (length(window_size) > 1) stop("Cannot comput for multiple window sizes")

  if (length(x) > window_size) {
    window_offset <- seq(1, length(x) - window_size, by = step_size)
  }else{
    window_offset <- 1
    }

  sections <- matrix(nrow = window_size, ncol = length(window_offset))
  for(i in seq_along(window_offset)){
    sections[, i] <-
      x[window_offset[i]:(window_offset[i] + window_size - 1)]*window_function
```

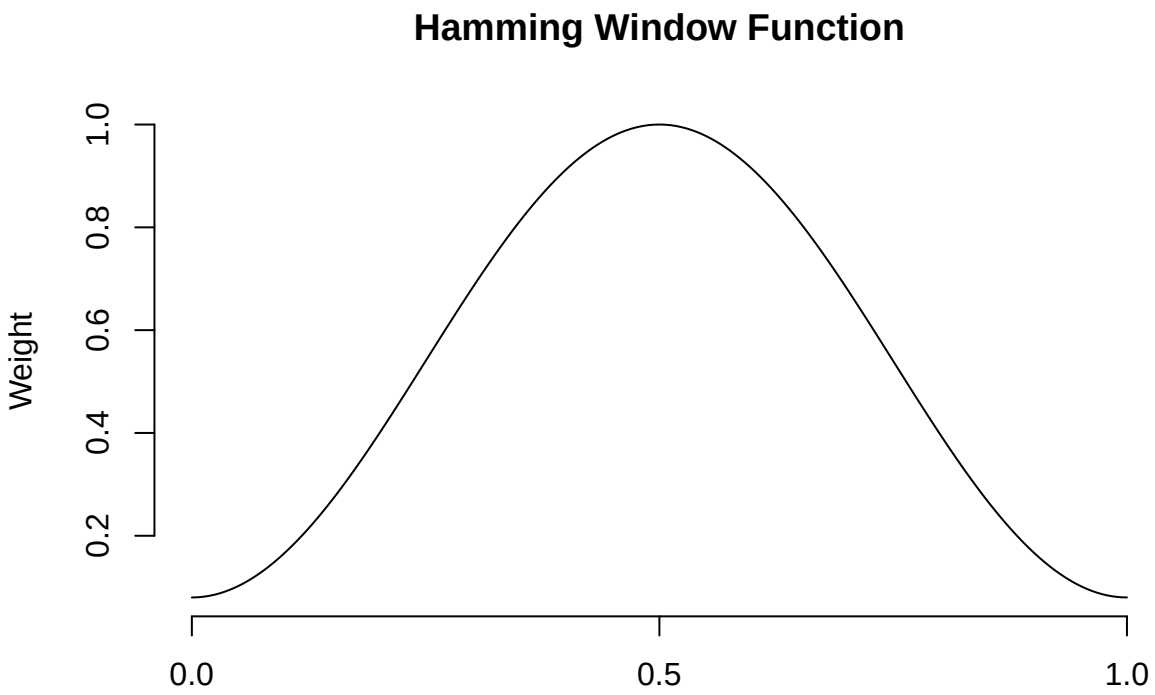

Figure 2: Plotted is the weight placed on each observation in a window by the Hamming function

```
  }

  stft <- mvfft(sections)

  return(stft)
}
```

Observation Positions: Frequency Domain

Given observations over a spectrum such as that produced by the `complex_stft` function, we want to be able to index the frequencies at which the intensity has been calculated. The order in which frequencies are calculated by the discrete Fourier transform is $\{0, 1, \ldots, \frac{n}{2} - 1, \pm\frac{n}{2}, -(\frac{n}{2} - 1), \ldots, -2, -1\}$ for series with an even number of observations and $\{0, 1, \ldots, \frac{n-1}{2}, -\frac{n-1}{2}, -(\frac{n}{2} - 1), \ldots, -2, -1\}$ for those with an odd number of observations. The `full_spectrum_frequencies` combines this information with the sampling rate to find the frequency in Hz to wich our spectrogram frequencies correspond.

```
full_spectrum_frequencies <- function(full_spectrum, fs = 500){
  if(is.vector(full_spectrum)){

    n_freq <- length(full_spectrum)

  }else if(is.matrix(full_spectrum)){

    n_freq <- nrow(full_spectrum)

  }else{

      stop('full_spectrum is in incorrect format')
  }


  if(n_freq %% 2 == 1){
    frequencies <- seq(from = 0, to = floor(n_freq/2), length.out = (n_freq + 1)/2)
    frequencies <- c( frequencies, -rev( frequencies[ 2:((n_freq + 1)/2) ] ) )
  }else{
    frequencies <- seq(from = 0, to = n_freq/2, length.out = (n_freq/2) + 1)
    frequencies <- c( frequencies, -rev( frequencies[ 2:(n_freq/2) ] ) )
  }

  frequencies <- frequencies * (fs/ n_freq)

  return(frequencies)
}
```

Observation Positions: Time Domain

In the STFT, windows are centred at a particular point in the time of the original signal. These centre points of each window are calculated by the `stft_time` function.

```
stft_time <- function(stft, fs, step_size){

  if(!is.matrix(stft)){
```

```
    stop('spectrum must be a matrix')
  }

  window_offset <- seq(1, by = step_size, length.out = ncol(stft))
  times <- window_offset / fs

  return(times)
}
```

Magnitude STFT

An intermediate step between a complex STFT and a spectrogram is the magnitude STFT. This is simply the absolute value of each point on the complex STFT and can be interpreted as the energy intensity at each point (f, t). Transforming a complex STFT to a magnitude STFT throws away phase information about the original signal and so the resulting representation is no longer uniquely reversible as there are potentially infinite signals that will all produce the same magnitude spectrum at any given point $(\cdot, t)$ by simly having different phases associated with them. This becomes a problem for inverting a STFT as each $(\cdot, t)$ overlaps with adjacent time points.

Simply taking the absolute value of the complex STFT will produce a magnitude STFT.

Spectrogram Inversion

The DFT and inverse DFT define a fully invertible transform of a signal from the time to the frequency domain and vice versa. In our case the signal is real valued in the time domain but complex valued over the spectrum in the frequency domain. The spectrogram is based on the magnitude of the signal over the frequency domain and so the phase information for the signal is lost. This is why spectrogram inversion is challenging, as the full time domain signal must fit tgether in a cohesive way. The steps required to perform a spectrogram inversion will be built up iteratively.

Complex STFT

The first step is to ensure that a complex STFT can be successfully inverted. The `inv_complex_stft` accomplishes this. This function will reproduce at original signal of a comples STFT. The only difference is that the reconstruction is a slighly shorter sequence due to the stft algorithm only considering full windows. Plotting and playing back the reconstructed acoustic signal makes the effectiveness of the reconstruction clear.

```
inv_complex_stft <- function(stft, window_size = nrow(stft),
  window_function = hamming(window_size),
  step_size = ceiling(window_size/2), stft_scale = nrow(stft)){

  if (!is.complex(stft) & !is.matrix(stft)) stop("'stft' has to be a complex matrix.")

  if (length(window_size) > 1) stop("Cannot compute for multiple window sizes")

  window_offset <- seq(1, by = step_size, length.out = ncol(stft))

  inv_stft <- (Re(mvfft(stft, inverse = T))) / stft_scale

  signal <- matrix(0, nrow = max(window_offset) + window_size - 1,
```

```
    ncol = length(window_offset))
  weight <-  matrix(0, nrow = max(window_offset) + window_size - 1,
    ncol = length(window_offset))

  temp_w <- apply(array(1, dim = dim(inv_stft)),
    2, `*`, window_function)

  for(i in seq_along(window_offset)){
    signal[window_offset[i]:(window_offset[i] + window_size - 1), i] <- inv_stft[, i]
    weight[window_offset[i]:(window_offset[i] + window_size - 1), i] <- temp_w[, i]
  }

  weight <- rowSums(weight)
  signal <- rowSums(signal) / weight

  return(signal)
}
```

Magnitude STFT

Complications in inverting a Magnitude STFT arise as phase information about the signal has been discarded. This is where RTISI-LA algorithm will become useful. Firstly however, simply consider the structure of phase information.

Given the phase associated with a magnitude stft, the inversion is straightforward.

The process becomes more complicated when there is no available phase information. In this case we generate a random phase for the first window of the signal and then try to use this to infer the phase of the next frames.

```
random_phase <- function(window_size){
  if(window_size %% 2 == 1){
    positive_frequencies <- runif((window_size + 1)/2, min = -pi, max = pi)
    phase <- c( positive_frequencies, -rev( positive_frequencies[-1] ) )
  }else{
    positive_frequencies <- runif((window_size/2) + 1, min = -pi, max = pi)
    phase <- c( positive_frequencies, -rev( positive_frequencies[ 2:(window_size/2) ] ) )
  }

  return(phase)
}

phase_from_partial <- function(partial_signal, partial_scaling,
  window_function = hamming(length(partial_signal))){

  if(!is.vector(partial_signal)) stop('partial_signal must be a vector')
  if(!is.vector(partial_scaling)) stop('partial_scaling must be a vector')
  if(length(partial_signal) != length(partial_scaling)){
    stop('partial_signal and partial_scaling must be the same length')
  }

  scaled_signal <- partial_signal / partial_scaling
  scaled_signal[is.nan(scaled_signal)] <-  0
```

```
  partial_fft <- fft(window_function*scaled_signal)

  phase <- atan2(Im(partial_fft), Re(partial_fft))
}

rtisi_la <- function(mag_stft, step_size, window_function = hamming(nrow(mag_stft)),
  iterations = 10, fft_normaliser = nrow(mag_stft)){
  if(!is.matrix(mag_stft)) stop('mag_stft must be a matrix')
  if(is.complex(mag_stft)) stop('mag_stft must be a real valued matrix')

  window_size <- nrow(mag_stft)
  total_frames <- ncol(mag_stft)

  committed_signal <- rep(0,  step_size*(total_frames - 1) + window_size)
  committed_weight <- rep(0,  step_size*(total_frames - 1) + window_size)

  for(i in seq.int(total_frames)){
    look_ahead <- min((window_size / step_size) - 1, total_frames - i)
    in_frame_signal <-
      committed_signal[(i-1)*step_size + (1:(window_size + look_ahead*step_size))]
    in_frame_weight <-
      committed_weight[(i-1)*step_size + (1:(window_size + look_ahead*step_size))]

    message('Initial Iteration')

    for(j in 0:look_ahead){
      if(j == 0 & i == 1){

        message(paste('Generate a random initial phase for frame',
          i, 'of magnitude spectrum'))
        initial_phase <- random_phase(window_size)

      }else
        if(j == 0 & i != 1){

        message(
          paste(
          'Committed signal used to generate inital phase from partial signal for frame',
            i
            )
          )
        initial_phase <- phase_from_partial(
          partial_signal = committed_signal[j*step_size + 1:window_size],
          partial_scaling = committed_weight[j*step_size + 1:window_size],
          window_function = window_function
          )

      }else{

        initial_phase <- phase_from_partial(
          partial_signal = in_frame_signal[j*step_size + 1:window_size],
          partial_scaling =in_frame_weight[j*step_size + 1:window_size],
          window_function = window_function
```

```
      )

  }

  in_frame_signal[j*step_size + 1:window_size] <-
    in_frame_signal[j*step_size + 1:window_size] +
    (Re(fft(mag_stft[, i + j]*exp((0+1i)*initial_phase), inverse = T)) /
        fft_normaliser)

  in_frame_weight[j*step_size + 1:window_size] <-
    in_frame_weight[j*step_size + 1:window_size] +
    window_function
}

frame_i_estimate <- (in_frame_signal / in_frame_weight)[1:window_size]

temp_stft <- complex_stft(frame_i_estimate, window_size = window_size,
  step_size = step_size)

estimated_phase <- atan2(Im(temp_stft), Re(temp_stft))

message(paste('Iteratively improving phase estimates for frame', i))
for(k in seq.int(iterations)){
  in_frame_signal <-
    committed_signal[(i-1)*step_size + (1:(window_size + look_ahead*step_size))]
  in_frame_weight <-
    committed_weight[(i-1)*step_size + (1:(window_size + look_ahead*step_size))]

  for(j in 0:look_ahead){
    if(j == 0){

      initial_phase <- estimated_phase

    }else if(j > 0){

      initial_phase <- phase_from_partial(
        partial_signal = in_frame_signal[j*step_size + 1:window_size],
        partial_scaling =in_frame_weight[j*step_size + 1:window_size],
        window_function = window_function)

    }

    in_frame_signal[j*step_size + 1:window_size] <-
      in_frame_signal[j*step_size + 1:window_size] +
      (Re(fft(mag_stft[, i + j]*exp((0+1i)*initial_phase), inverse = T)) /
          fft_normaliser)

    in_frame_weight[j*step_size + 1:window_size] <-
      in_frame_weight[j*step_size + 1:window_size] +
      window_function
  }

  frame_i_estimate <- (in_frame_signal / in_frame_weight)[1:window_size]
```

```
      temp_stft <- complex_stft(frame_i_estimate, window_size = window_size,
        step_size = step_size)

      estimated_phase <- atan2(Im(temp_stft), Re(temp_stft))
    }

    message(paste('Committing frame', i))

    committed_signal[(i-1)*step_size + 1:window_size] <-
      committed_signal[(i-1)*step_size + 1:window_size] +
      (Re(fft(mag_stft[, i]*exp((0+1i)*estimated_phase), inverse = T)) /
          fft_normaliser)*
      window_function

    committed_weight[(i-1)*step_size + 1:window_size] <-
      committed_weight[(i-1)*step_size + 1:window_size] +
      window_function
  }

  reconstructed_signal <- committed_signal / committed_weight
  return(reconstructed_signal)
}
```

The RTISI-LA algorithm estimates the time domain signal that produced a given magnitude stft spectrum. Thus in order to invert a preprocessed spectrogram it must first be converted into the format of a magnitude STFT spectrum.

```
surface_to_mag_stft <- function(surface, surface_freq, pos_mag_freq){
  mag_surface <- sqrt(10^(surface / 10))

  mag_stft <- matrix(nrow = length(pos_mag_freq), ncol = ncol(surface))
  mag_stft[
    pos_mag_freq < (min(surface_freq) - 0.1) |
      pos_mag_freq > (max(surface_freq) + 0.1)
    ] <- 0
  for(i in seq.int(ncol(x))){

    mag_stft[is.na(mag_stft[,i]),i] <-
      sqrt(10^((
        spline(surface_freq, surface[,i], xout = pos_mag_freq[is.na(mag_stft[,i])])$y
        )/10))

  }

  mag_stft <- rbind(mag_stft, mag_stft[(length(pos_mag_freq) - 1):2,])

  return(mag_stft)
}
```

Manipulating the preprocessed spectrograms appropriately means that acoustic signals which produce approximations for the preprocessed spectrograms can be obtained and listened back to.

References

Zhu, Xinglei, Gerald T Beauregard, and Lonce L Wyse. 2007. "Real-Time Signal Estimation from Modified Short-Time Fourier Transform Magnitude Spectra." *IEEE Transactions on Audio, Speech, and Language Processing* 15 (5). IEEE: 1645–53.