Online Resource 1
Rhythm Reproduction Processing and Evaluation Scripts

Supplementary information for: Rhythmic Skills and Reading Performance in Brazilian Primary School Children: Evidence from a Low-SES Context

Description. Python scripts for audio preprocessing, rhythmic tap detection, temporal correspondence evaluation, score aggregation, and the auxiliary rhythmEval implementation. Manual quality-control exclusions were performed separately.

The following scripts document the workflow used for processing rhythm-reproduction recordings. The same signal-processing procedure was applied across the three data collection phases. The auxiliary rhythmEval.py implementation is reproduced in Section 2.

Script for the extraction of the rhythm reproduction analysis

```
import os
import glob
import numpy as np
import pandas as pd
import scipy.io.wavfile as wav
import matplotlib.pyplot as plt

from mne.filter import filter_data
from scipy.signal import hilbert, find_peaks
from rhythmEval import rhythmEval


# ============================================================
# CONFIGURATION
# ============================================================

# Default sampling frequency
SFREQ = 44100

# ------------------------------------------------------------
# Stimulus-specific settings
# ------------------------------------------------------------
#
# The first two repetitions of each rhythmic stimulus were
# treated as a learning/practice phase and excluded from the
# analysis.
#
# The values below indicate the duration, in seconds, of the
# first two repetitions for each stimulus.
#
# Initial cut applied equally to all four rhythmic stimuli.
# Initial cut duration: 6.542 seconds.
# The recorded S4 run contains this value explicitly.
# ------------------------------------------------------------
# ------------------------------------------------------------

STIMULUS_CONFIG = {
    "S1": {
        "analysis_start_sec": 6.542,
        "stimulus_taps_path": "<cropped_S1_stimulus_file.npy>"
    },
    "S2": {
        "analysis_start_sec": 6.542,
        "stimulus_taps_path": "<cropped_S2_stimulus_file.npy>"
    },
    "S3": {
        "analysis_start_sec": 6.542,
        "stimulus_taps_path": "<cropped_S3_stimulus_file.npy>"
    },
    "S4": {
        "analysis_start_sec": 6.542,
        "stimulus_taps_path": "<cropped_S4_stimulus_file.npy>"
    }
}

# ------------------------------------------------------------
# Local directories
```

```
# ------------------------------------------------------------

# Directory containing the original .wav response recordings
responses_wav_dir = "<wav_files_directory>"

# Directory in which detected taps will be saved
taps_output_dir = "<tap_output_directory>"

# Directory in which tap-detection plots will be saved
plots_output_dir = "<plot_output_directory>"

# Final output files
stimulus_scores_csv = "<stimulus_level_scores.csv>"
participant_scores_csv = "<participant_level_rhythm_reproduction_scores.csv>"

os.makedirs(taps_output_dir, exist_ok=True)
os.makedirs(plots_output_dir, exist_ok=True)


# ============================================================
# 1. EXTRACTION OF RHYTHMIC EVENTS (TAPS)
# ============================================================

def extract_taps_response(
    signal,
    sfreq=44100.,
    fmin=100.,
    fmax=200.,
    thresh_factor=2.5
):
    """
    Extract rhythmic events (taps) from an audio recording.

    Parameters
    ----------
    signal : np.ndarray
        Audio signal.
    sfreq : float
        Sampling frequency in Hz.
    fmin : float
        Lower bound of the band-pass filter in Hz.
    fmax : float
        Upper bound of the band-pass filter in Hz.
    thresh_factor : float
        Multiplier applied to the standard deviation of the
        amplitude envelope to define peak prominence.

    Returns
    -------
    taps : np.ndarray
        Binary time series in which detected tap events are
        coded as 1 and all other samples as 0.
    """

    # Band-pass filter
    filtered = filter_data(
        signal,
        sfreq,
        fmin,
        fmax,
        verbose=False
    )

    # Compute amplitude envelope using the Hilbert transform
    n = int(np.log2(signal.size)) + 1

    envelope = np.abs(
        hilbert(
            filtered,
            N=2**n
        )
    )

    # Detection threshold
    threshold = (
```

```
        thresh_factor *
        np.std(envelope)
    )

    # Detect peaks
    peaks_idx, _ = find_peaks(
        envelope,
        distance=5000,
        prominence=(threshold, None)
    )

    # Exclude peaks very close to signal boundaries
    peaks_idx = peaks_idx[
        (peaks_idx > 1000) &
        (peaks_idx < len(envelope) - 1000)
    ]

    # Create binary tap series
    taps = np.zeros_like(envelope)
    taps[peaks_idx] = 1

    return taps


# ============================================================
# 2. IDENTIFICATION OF PARTICIPANT AND STIMULUS
# ============================================================

def parse_filename(filename):
    """
    Extract participant ID and stimulus ID from a file name.

    The function assumes that filenames contain a participant
    identifier and a stimulus identifier.

    Examples
    --------
    001_S1.wav
    001S1.wav

    Returns
    -------
    participant_id : str
    stimulus_id : str
    """

    name = os.path.basename(filename)

    name = (
        name
        .replace(".wav", "")
        .replace(".npy", "")
    )

    if "_" in name:

        participant_id, stimulus_id = (
            name.split("_", 1)
        )

    else:

        participant_id = name[:3]
        stimulus_id = name[3:]

    return (
        participant_id,
        stimulus_id.upper()
    )


# ============================================================
# 3. PROCESS ORIGINAL AUDIO RECORDINGS
# ============================================================
```

```
audio_files = glob.glob(
    os.path.join(
        responses_wav_dir,
        "*.wav"
    )
)


for file in audio_files:

    participant_id, stimulus_id = (
        parse_filename(file)
    )

    # Skip files that do not correspond to one of the
    # four rhythmic stimuli
    if stimulus_id not in STIMULUS_CONFIG:

        print(
            f"Skipping file with unknown stimulus: {file}"
        )

        continue


    # Retrieve stimulus-specific cut duration
    analysis_start_sec = (
        STIMULUS_CONFIG[
            stimulus_id
        ]["analysis_start_sec"]
    )


    if analysis_start_sec is None:

        raise ValueError(
            f"No analysis_start_sec value was specified "
            f"for {stimulus_id}. Enter the duration of the "
            f"first two practice repetitions before running "
            f"the complete analysis."
        )


    # Read audio file
    sr, raw = wav.read(file)


    # If stereo, retain the second channel, following
    # the original analysis procedure
    if raw.ndim > 1:

        raw = raw[:, 1]


    raw = raw.astype(
        np.float64
    )


    # ----------------------------------------------------------
    # Exclude the first two repetitions
    # ----------------------------------------------------------

    cut_samples = int(
        analysis_start_sec *
        sr
    )

    cropped_raw = raw[
        cut_samples:
    ]


    # ----------------------------------------------------------
    # Extract taps
```

```
# ----------------------------------------------------------

taps = extract_taps_response(
    cropped_raw,
    sfreq=sr
)


# ----------------------------------------------------------
# Save extracted tap series
# ----------------------------------------------------------

output_filename = (
    f"{participant_id}_{stimulus_id}.npy"
)

np.save(
    os.path.join(
        taps_output_dir,
        output_filename
    ),
    taps
)


# ----------------------------------------------------------
# Generate plot for visual verification
# ----------------------------------------------------------

times = np.arange(
    0,
    cropped_raw.shape[0] / sr,
    1. / sr
)


fig, ax = plt.subplots(
    figsize=(10, 3)
)


ax.plot(
    times,
    cropped_raw,
    color="gray",
    label="Cropped Audio"
)


tap_idx = np.where(
    taps == 1
)[0]


ax.plot(
    times[tap_idx],
    np.zeros(tap_idx.size),
    "ro",
    markersize=2,
    label="Detected Taps"
)


ax.set_title(
    f"Participant {participant_id} - {stimulus_id}"
)

ax.set_xlim(
    [
        times.min(),
        times.max()
    ]
)

ax.set_xlabel(
```

```
        "Time (s)"
    )

    ax.legend()


    fig.savefig(
        os.path.join(
            plots_output_dir,
            f"{participant_id}_{stimulus_id}.png"
        ),
        bbox_inches="tight"
    )


    plt.close(fig)


print(
    "Audio recordings processed and tap series saved."
)


# ============================================================
# 4. CALCULATE STIMULUS-RESPONSE TEMPORAL CORRESPONDENCE
# ============================================================

results = []


for stimulus_id, config in STIMULUS_CONFIG.items():

    stimulus_path = (
        config[
            "stimulus_taps_path"
        ]
    )


    # Load stimulus tap series
    stimulus = np.load(
        stimulus_path
    )


    # Select participant responses corresponding to
    # the current stimulus
    response_files = glob.glob(
        os.path.join(
            taps_output_dir,
            f"*_{stimulus_id}.npy"
        )
    )


    for response_path in response_files:

        participant_id, detected_stimulus = (
            parse_filename(
                response_path
            )
        )


        response = np.load(
            response_path
        )


        # ----------------------------------------------------
        # Match time-series lengths
        # ----------------------------------------------------

        min_len = min(
            len(stimulus),
```

```
            len(response)
        )


        cropped_stimulus = (
            stimulus[:min_len]
        )

        cropped_response = (
            response[:min_len]
        )


        # ---------------------------------------------------
        # Calculate temporal correspondence
        # ---------------------------------------------------
        #
        # Parameters are retained from the original analysis.
        # ---------------------------------------------------

        score, _ = rhythmEval(
            cropped_stimulus,
            cropped_response,
            SFREQ,
            100,
            120,
            0.01,
            30
        )


        results.append(
            {
                "Participant_ID":
                    participant_id,

                "Stimulus":
                    detected_stimulus,

                "Phase":
                    1,

                "Correlation":
                    score
            }
        )


# ============================================================
# 5. CREATE STIMULUS-LEVEL DATASET
# ============================================================

stimulus_df = pd.DataFrame(
    results
)


# Sort results
stimulus_df = (
    stimulus_df
    .sort_values(
        [
            "Participant_ID",
            "Stimulus"
        ]
    )
    .reset_index(drop=True)
)


# Save numerical values using the conventional decimal point
stimulus_df.to_csv(
    stimulus_scores_csv,
    index=False
)
```

```
print(
    f"Stimulus-level correlation scores saved to: "
    f"{stimulus_scores_csv}"
)


# ============================================================
# 6. FISHER Z TRANSFORMATION
# ============================================================

# Cross-correlation values are bounded correlation
# coefficients. Before averaging across stimuli, scores are
# transformed using Fisher's r-to-z transformation.

#
# Values equal to exactly -1 or 1 cannot be transformed using
# arctanh. Clipping is therefore applied only to avoid
# numerical infinities.
#

epsilon = 1e-7


stimulus_df[
    "Correlation_clipped"
] = stimulus_df[
    "Correlation"
].clip(
    lower=-1 + epsilon,
    upper=1 - epsilon
)


stimulus_df[
    "Fisher_z"
] = np.arctanh(
    stimulus_df[
        "Correlation_clipped"
    ]
)


# ============================================================
# 7. AGGREGATE RHYTHM REPRODUCTION SCORE
# ============================================================

# The aggregate rhythm reproduction score corresponds to
# the mean Fisher-z-transformed correlation across the four
# rhythmic stimuli.


participant_summary = (
    stimulus_df
    .groupby(
        "Participant_ID",
        as_index=False
    )
    .agg(
        Valid_Stimuli=(
            "Fisher_z",
            "count"
        ),

        Mean_Fisher_z=(
            "Fisher_z",
            "mean"
        )
    )
)


# ============================================================
# 8. APPLY VALIDITY CRITERION
```

```
# ============================================================

# According to the analysis procedure, participants with
# fewer than two valid stimulus responses receive a missing
# aggregate rhythm reproduction score.

participant_summary[
    "Rhythm_Reproduction_Score"
] = np.where(
    participant_summary[
        "Valid_Stimuli"
    ] >= 2,

    participant_summary[
        "Mean_Fisher_z"
    ],

    np.nan
)


# ============================================================
# 9. OPTIONAL: BACK-TRANSFORMED SCORE FOR DESCRIPTIVE PURPOSES
# ============================================================

# The inferential analyses use the Fisher-z-transformed
# aggregate score.
#
# For descriptive presentation, the mean Fisher z value can
# be transformed back to the correlation metric using tanh.

participant_summary[
    "Back_Transformed_Correlation"
] = np.tanh(
    participant_summary[
        "Mean_Fisher_z"
    ]
)


# Participants with fewer than two valid stimuli should also
# have missing descriptive aggregate scores.

participant_summary.loc[
    participant_summary[
        "Valid_Stimuli"
    ] < 2,
    "Back_Transformed_Correlation"
] = np.nan


# ============================================================
# 10. EXPORT PARTICIPANT-LEVEL SCORES
# ============================================================

participant_summary.to_csv(
    participant_scores_csv,
    index=False
)


print(
    f"Participant-level rhythm reproduction scores saved to: "
    f"{participant_scores_csv}"
)


# ============================================================
# 11. SUMMARY
# ============================================================

n_total = (
    participant_summary.shape[0]
)
```

```
n_valid = (
    participant_summary[
        "Rhythm_Reproduction_Score"
    ]
    .notna()
    .sum()
)

n_missing = (
    participant_summary[
        "Rhythm_Reproduction_Score"
    ]
    .isna()
    .sum()
)


print("\nAnalysis completed.")

print(
    f"Participants with processed recordings: {n_total}"
)

print(
    f"Participants with valid aggregate scores: {n_valid}"
)

print(
    f"Participants with missing aggregate scores: {n_missing}"
)
```

2. Auxiliary implementation: rhythmEval.py

This auxiliary module contains the xCorr, resample, smooth, and rhythmEval functions called by the rhythm reproduction analysis. The source below is reproduced verbatim from the supplied rhythmEval.py file; no calculations or parameters have been changed.

```
def xCorr(X, Y):
    """Perform normalized cross-correlation.
    Parameters
    ----------
    X : array
        Time domain signal
    Y : array
        Time domain signal
    Returns
    -------
    xcorr : array
        Normalized cross-correlation
    """
    import numpy as np

    # Remove mean of signals
    _X = X - np.mean(X)
    _Y = Y - np.mean(Y)

    # Correlate
    xcorr = np.correlate(_X, _Y, mode=2)

    # Normalize
    norm = np.sqrt(np.dot(_X, _X) * np.dot(_Y, _Y))
    xcorr /= norm
    return xcorr

def resample(X, sfreq, target_freq):
    """Resample signal with discrete events (Dirac).
    Parameters
    ----------
    X : array
        Time domain signal
        Must be all zeros and ones. One when an event is present
```

```
    sfreq : float
        Sampling rate
    target_freq : float
        Target sampling rate
    Returns
    -------
    X_resampled : array
        Resampled signal
    """
    import numpy as np

    # Sample time
    times = np.linspace(0, (1./sfreq)*X.size, X.size)
    idx = times[np.where(X == 1)[0]]

    # Resample
    times_resampled = np.arange(0, times.max() + 1, 1./target_freq)
    idx_resampled = [np.where(np.isclose(t, times_resampled, atol=0.5/target_freq))[0][0] for t in idx]

    # Put in another array
    X_resampled = np.zeros((times_resampled.size))
    X_resampled[idx_resampled] = 1.
    return X_resampled

def smooth(X, sfreq, width=75):
    """Convolve signal with gaussian window.
    Parameters
    ----------
    X : array
        Time domain signal
    sfreq : float
        Sampling rate
    width : float
        Standard deviation of window, in ms
    Returns
    -------
    X_smooth : array
        Smoothed signal
    """
    import scipy.signal
    import numpy as np

    # Create kernel
    from scipy.signal.windows import gaussian
    gaussian_kernel = gaussian(M=3*sfreq, std=1e-3*width*sfreq)

    # Convolve
    X_smooth = np.convolve(X, gaussian_kernel, mode='same')
    return X_smooth

def rhythmEval(X, Y, sfreq, target_freq=100, width=75, stretch=15, n_stretch=50):
    """Performs the complex rhythm reproduction evaluation.
    The 2 signals are first smooted in time with a gaussian window.
    They are then correlated both in time and stretches, i.e. correlation
    is computed for every time shift and every stretch.
    Parameters
    ----------
    X : array
        Time domain signal, typically the stimulus
        Must be all zeros and ones when an event is present
    Y : array
        Time domain signal, typically the participant response
        Must be all zeros and ones when an event is present
        X and Y must have the same size
    sfreq : float
        Sampling rate
    target_freq : float
        Target sampling rate
    width : float
        Standard deviation of smoothing window, in ms
    stretch : float
        Limit of stretching allowed (15 corresponds to -15%/+15% range)
    n_stretch : int
        Number of stretching performed
    Returns
    -------
```

```
score : float
    Score of rhythm reproduction
C : array
    All value of the correlation for all times and stretches
"""
import numpy as np
import mne

# Convert stretch into normalized value
stretch_n = float(stretch)/100.

# Resample arrays to accelerate computations
X_r = resample(X, sfreq, target_freq)
Y_r = resample(Y, sfreq, target_freq)

# Smooth signals
X_s = smooth(X_r, target_freq, width=width)
Y_s = smooth(Y_r, target_freq, width=width)

# Extend array size to store stretching values
X_e = np.r_[X_s, np.zeros(1 * X_s.size)]

# Declare generalized correlation matrix
n = X_e.size * 2 - 1 # Size of the matrix
C = np.zeros((n, n_stretch))

# Compute normalized cross-correlation for every stretching values
for i, s in enumerate(np.linspace(1-stretch_n, 1+stretch_n, n_stretch)):
    Y_stretched = np.zeros(X_e.size) # Declare array size
    _temp = mne.filter.resample(Y_s, down=s) # Stretch signal
    Y_stretched[0:_temp.size] = _temp # Put stretch signal in array
    C[:, i] = xCorr(X_e, Y_stretched) # Compute normalized cross-correlation

# Return
return np.nanmax(C), C
```