MUSICNTWRK: data tools for music theory, analysis and composition

Marco Buongiorno Nardelli

MUSICNTWRK: data tools for music theory, analysis and composition

Marco Buongiorno Nardelli[1,2,3,4,5][0000−0003−0793−5055]

[1] CEMI, Center for Experimental Music and Intermedia, University of North Texas, Denton, TX 76203, USA
[2] iARTA, Initiative for Advanced Research in Technology and the Arts, University of North Texas, Denton, TX 76203, USA
[3] Department of Physics, University of North Texas, Denton, TX 76203, USA
[4] IMéRA - Institut d'Études Avancée of Aix-Marseille Université, Marseille 13004, France
[5] Laboratoire CNRS-PRISM, 13402 Marseille, France
mbn@unt.edu
http://www.musicntwrk.com

Abstract. We present the API for MUSICNTWRK, a python library for pitch class set and rhythmic sequences classification and manipulation, the generation of networks in generalized music and sound spaces, deep learning algorithms for timbre recognition, and the sonification of arbitrary data. The software is freely available under GPL 3.0 and can be downloaded at www.musicntwrk.com.

Keywords: Computational Music Theory · Computer Aided Composition · Data Tools · Machine Learning.

1 Introduction

Big data tools have become pervasive in virtually every aspects of culture and society. In music, application of such techniques in Music Information Retrieval applications are common and well documented. However, a full approach to musical analysis and composition is not yet available for the music community and there is a need for providing a more general education on the potential, and the limitations, of such approaches. From a more fundamental point of view, the abstraction of musical structures (notes, melodies, chords, harmonic or rhythmic progressions, timbre, etc.) as mathematical objects in a geometrical space is one of the great accomplishments of contemporary music theory. Building on this foundation, we have generalized the concept of musical spaces as networks and derive functional principles of compositional design by the direct analysis of the network topology. This approach provides a novel framework for the analysis and quantification of similarity of musical objects and structures, and suggests a way to relate such measures to the human perception of different musical entities. The original contribution of this work is in the introduction of the representation of musical spaces as large-scale statistical mechanics networks:

uncovering their topological structure is a fundamental step to understand their underlying organizing principles, and to unveil how classifications or rule-based frameworks (such as common-practice harmony, for instance) can be interpreted as emerging phenomena in a complex network system. Results from this research and the theoretical and technical foundation for this paper can be found in Ref. [1].

This paper is intended to introduce the community of computer music practitioners, composers and theorists to the practical use of data science tools (network theory, machine learning etc.) through the `MUSICNTWRK` package (www.musicntwrk.com), a python library comprised of four modules:

1. `pcsPy` - pitch class set classification and manipulation; construction of generalized pitch class set networks using distances between common descriptors; the analysis of scores and the generation of compositional frameworks;
2. `rhythmPy` - rhythmic sequence classification and manipulation; and construction of rhythmic sequence networks using various definitions of rhythmic distance;
3. `timbrePy` - orchestration color networks; analysis and characterization of timbre from a (psycho-)acoustical point of view; and machine learning models for timbre recognition; and
4. `sonifiPy` - a module for the sonification of arbitrary data structures, including automatic score (musicxml) and MIDI generation.

In the following we will discuss the API of each module. All theoretical and technical analysis, including the definition of all the quantities that `MUSICNTWRK` is able to calculate, are not explicitly discussed here. The interested reader can find all this information in Ref. [1].

2 MUSICNTWRK

`MUSICNTWRK` is a python library written by the author and available at https://www.musicntwrk.com or on GitHub: https://github.com/marcobn/musicntwrk. `MUSICNTWRK` is written in python 3 and requires installation of the following modules via the "pip install" command:[§]

1. System modules: `sys, re, time, os`
2. Math modules: `numpy, itertools, fractions, gcd, functools`
3. Data modules: `pandas, sklearn, networkx, community, tensorflow`
4. Music and audio modules: `music21, librosa`
5. Parallel processing: `mpi4py`
6. Visualization modules: `matplotlib, vpython` (optional)

The reader is encouraged to consult the documentation of each package to get acquainted with its purposes and use. In what follows we provide the full API of `MUSICNTWRK` only.

[§] this step is unnecessary if running on a cloud service like Google Colaboratory.

2.1 pcsPy

pcsPy is a module for pitch class set classification and manipulation in any arbitrary temperament; the construction of generalized pitch class set networks using distances between common descriptors (interval vectors, voice leadings); the analysis of scores and the generation of compositional frameworks.

The PCSet class. pcsPy is is comprised of the PCSet class and its methods (listed below) and a series of functions for pcs network manipulations. The PCSet class deals with the classification and manipulation of pitch set classes generalized to arbitrary temperament systems (arbitrary number of pitches). The following methods are available:

```
def class PCSet
```

- `def __init__(self,pcs,TET=12,UNI=True,ORD=True)`
  - `pcs (int)` pitch class set as list or numpy array
  - `TET (int)` number of allowed pitches in the totality of the musical space (temperament). Default = 12 tones equal temperament
  - `UNI (logical)` if True, eliminate duplicate pitches (default)
  - `ORD (logical)` if True, sorts the pcs in ascending order (default)
- `def normalOrder(self)`
  Order the pcs according to the most compact ascending scale in pitch-class space that spans less than an octave by cycling permutations.
- `def normal0Order(self)`
  As normal order, transposed so that the first pitch is 0
- `def transpose(self,t=0)`
  Transposition by t (int) units (modulo TET)
- `def zeroOrder(self)`
  transposed so that the first pitch is 0
- `def inverse(self)`
  inverse operation: (-pcs modulo TET)
- `def primeForm(self)`
  most compact normal 0 order between pcs and its inverse
- `def intervalVector(self)`
  total interval content of the pcs
- `def LISVector(self)`
  Linear Interval Sequence Vector: sequence of intervals in an ordered pcs
- `def operator(self,name)`
  operate on the pcs with a distance operator
  - `name (str)` name of the operator O(ni)
- `def forteClass(self)`
  Name of pcs according to the Forte classification scheme (only for TET=12)
- `def jazzChord(self)`
  Name of pcs as chord in a jazz chart (only for TET=12 and cardinalities 7)
- `def commonName(self)`
  Display common name of pcs (music21 function - only for TET=12)

- `def commonNamePrime(self)`
  As above, for prime forms
- `def nameWithPitchOrd(self)`
  Name of chord with first pitch of pcs in normal order
- `def nameWithPitch(self)`
  Name of chord with first pitch of pcs
- `def displayNotes(self,xml=False,prime=False)`
  Display pcs in score in musicxml format. If prime is True, display the prime form.
  - `xml (logical)` write notes on file in musicxml format
  - `prime (logical)` write pcs in prime form

Network functions. `pcsPy` contains specific functions for network generation and analysis. Network functions include:

- `def pcsDictionary(Nc,order=0,TET=12,row=False,a=np.array(None))`
  Generate the dictionary of all possible pcs of a given cardinality in a generalized musical space of TET pitches. Returns the dictionary as pandas DataFrame and the list of all Z-related pcs
  - `Nc (int)` cardinality
  - `order (logical)` if 0 returns pcs in prime form, if 1 retrns pcs in normal order, if 2, returns pcs in normal 0 order
  - `row (logical)` if True build dictionary from tone row, if False, build dictionary from all combinatorial pcs of Nc cardinality given the totality of TET.
  - `a (int)` if row = True, a is the list of pitches in the tone row
- `def pcsNetwork(input_csv, thup=1.5, thdw=0.0,TET=12, distance= 'euclidean', col=2,prob=1)`
  generate the network of pcs based on distances between interval vectors
  In output it writes the nodes.csv and edges.csv as separate files in csv format
  - `input_csv (str)` file containing the dictionary generated by pcsNetwork
  - `thup, thdw (float)` upper and lower thresholds for edge creation
  - `distance (str)` choice of norm in the musical space, default is 'euclidean'
  - `col (int)` metric based on interval vector, col = 1 can be used for voice leading networks in spaces of fixed cardinality NOT RECOMMENDED
  - `prob (float)` if not 1, defines the probability of acceptance of any given edge
- `def pcsEgoNetwork(label,input_csv,thup_e=5.0,thdw_e=0.1,thup=1.5,thdw=0.1, TET=12,distance='euclidean')`
  Generates the network for a focal node (ego) and the nodes to whom ego is directly connected to (alters). In output it writes the nodes_ego.csv, edges_ego.csv and edges_alters.csv as separate files in csv format
  - `label (str)` label of the ego node
  - `thup_e, thdw_e (float)` upper and lower thresholds for edge creation from ego node

  - `thup, thdw (float)` upper and lower thresholds for edge creation among alters
  - `distance (str)` choice of norm in the musical space, default is 'euclidean'

– `def vLeadNetwork(input_csv,thup=1.5,thdw=0.1,TET=12,w=True, distance='euclidean',prob=1)`
Generation of the network of all minimal voice leadings in a generalized musical space of TET pitches based on the minimal distance operators select by distance. In output returns nodes and edges tables as pandas DataFrames.
  - `input_csv (str)` file containing the dictionary generated by pcsNetwork
  - `thup, thdw (float)` upper and lower thresholds for edge creation
  - `distance (str)` choice of norm in the musical space, default is 'euclidean'
  - `w (logical)` if True it writes the nodes.csv and edges.csv files in csv format

– `def vLeadNetworkByName(input_csv,thup=1.5,thdw=0.1,TET=12,w=True, distance='euclidean',prob=1)`
Generation of the network of all minimal voice leadings in a generalized musical space of TET pitches based on the minimal distance operators select by name. In output returns nodes and edges tables as pandas DataFrames. Available also in vector form for computational efficiency as `vLeadNetworkByNameVec`
  - `input_csv (str)` file containing the dictionary generated by pcsNetwork
  - `name (str)` name of operator for edge creation
  - `distance (str)` choice of norm in the musical space, default is 'euclidean'
  - `w (logical)` if True it writes the nodes.csv and edges.csv files in csv format

– `def scoreNetwork(seq,TET=12)`
Generates the directional network of chord progressions from any score in musicxml format
  - `seq (int)` list of pcs for each chords extracted from the score

– `def scoreDictionary(seq,TET=12)`
Builds the dictionary of pcs in any score in musicxml format

– `def readScore(inputxml,TET=12,music21=False)`
Reads musicxml score and returns chord sequence
  - `inputxml (str)` score file
  - `music21 (logical)` if True search the music21 corpus

2.2 `rhythmPy`.

`rhythmPy` is a module for rhythmic sequence classification and manipulation; and the construction of rhythmic sequence networks using various definitions of rhythmic distance.

The `RHYTHMSeq` class `rhythmPy` is comprised of the `RHYTHMSeq` class and its methods (listed below) and a series of functions for rhythmic network manipulations. The `RHYTHMSeq` class deals with the classification and manipulation of rhythmic sequences. The following methods are available:

`def class RHYTHMSeq`

- `__init__( self,rseq,REF='e',ORD=False)`
  - `rseq (str/fractions/floats)` rhythm sequence as list of strings/fractions/floats name
  - `REF (str)` reference duration for prime form the RHYTHMSeq class contains a dictionary of common duration notes that uses the fraction module for the definitions (implies import fraction as fr): {`'w':fr.Fraction(1,1), 'h':fr.Fraction(1,2),'q':fr.Fraction(1,4), 'e':fr.Fraction(1,8), 's':fr.Fraction(1/16),'t':fr.Fraction(1,32), 'wd':fr.Fraction(3,2), 'hd':fr.Fraction(3,4),'qd':fr.Fraction(3,8), 'ed':fr.Fraction(3,16), 'sd':fr.Fraction(3,32),'qt':fr.Fraction(1,6), 'et':fr.Fraction(1,12), 'st':fr.Fraction(1,24), 'qq':fr.Fraction(1,5), 'eq':fr.Fraction(1,10), 'sq':fr.Fraction(1,20)`}. This dictionary can be extended by the user on a case by case need.
  - `ORD (logical)` if `True` sort durations in ascending order
- `def normalOrder(self)`
  Order the rhythmic sequence according to the most compact ascending form.
- `def augment(self,t='e')`
  Augmentation by t units
  - `t (str` duration of augmentation
- `def diminish(self,t='e')`
  Diminution by t units
  - `t (str)` duration of diminution
- `def retrograde(self)`
  Retrograde operation
- `def isNonRetro(self)`
  Check if the sequence is not retrogradable
- `def primeForm(self)`
  reduce the series of fractions to prime form
- `def durationVector(self,lseq=None)`
  total relative duration ratios content of the sequence
  - `lseq (list of fractions)` reference list of duration for evaluating interval content the default list is: [fr.Fraction(1/8),fr.Fraction(2/8), fr.Fraction(3/8), fr.Fraction(4/8),fr.Fraction(5/8), fr.Fraction(6/8), fr.Fraction(7/8), fr.Fraction(8/8), fr.Fraction(9/8)]
- `def durationVector(self,lseq=None)`
  inter-onset duration interval content of the sequence
  - `lseq (list of fractions)` reference list of duration for evaluating interval content the default list is the same as above.

Network functions. `rhythmPy` contains specific functions for network generation and analysis. Network functions include:

- `def rhythmDictionary(Nc,a=None,REF='e')`
  Generates the dictionary of all possible rhythmic sequences of Nc length in a generalized meter space of N durations. Returns the dictionary as pandas DataFrame and indicates all non retrogradable and Z-related cells
  - `Nc (int)` cell length
  - `a (str)` list of durations in the rhythm sequence
- `def rhythmPDictionary(N,Nc,REF='e')`
  Generate the dictionary of all possible rhythmic sequences from all possible groupings of N REF durations. Returns the dictionary as pandas DataFrame and indicates all non retrogradable and Z-related cells
  - `Nc (int)` cell length
  - `N (int)` number of REF units
- `def rhythmNetwork(input_csv,thup=1.5,thdw=0.0,distance='euclidean',prob=1,w=False)`
  Generates the network of rhythmic cells based on distances between duration vectors. In output it writes the nodes.csv and edges.csv as separate files in csv format
  - `input_csv (str)` file containing the dictionary generated by rhythmNetwork
  - `thup, thdw (float)` upper and lower thresholds for edge creation
  - `distance (str)` choice of norm in the musical space, default is 'euclidean'
  - `prob (float)` if not 1, defines the probability of acceptance of any given edge
  - `w (logical)` if True it writes the nodes.csv and edges.csv files in csv format
- `def rLeadNetwork(input_csv,thup=1.5,thdw=0.1,w=True,distance='euclidean',prob=1)`
  Generation of the network of all minimal rhythm leadings in a generalized musical space of Nc-dim rhythmic cells based on the rhythm distance operator. Returns nodes and edges tables as pandas DataFrames
  - `input_csv (str)` file containing the dictionary generated by rhythmNetwork
  - `thup, thdw (float)` upper and lower thresholds for edge creation
  - `distance (str)` choice of norm in the musical space, default is 'euclidean'
  - `prob (float)` if not 1, defines the probability of acceptance of any given edge
  - `w (logical)` if True it writes the nodes.csv and edges.csv files in csv format

2.3 `timbrePy`

`timbrePy` comprises of two sections: the first deals with orchestration color and it is the natural extension of the score analyzer in `pscPy`; the second deals with analysis and characterization of timbre from a (psycho-)acoustical point of view. In particular, it provides: the characterization of sound using, among others, Mel Frequency or Power Spectrum Cepstrum Coefficients (MFCC or PSCC); the construction of timbral networks using descriptors based on MF- or PS-CCs; and machine learning models for timbre recognition through the TensorFlow Keras framework.

Orchestration analysis. The orchestration analysis section of `timbrePy` comprises of the following modules:

- `def orchestralVector(inputfile,barplot=True)`
  Builds the orchestral vector sequence from score in `musicxml` format. Returns the score sliced by beat; orchestration vector.
  - `barplot=True` plot the orchestral vector sequence as a matrix
- `def orchestralNetwork(seq)`
  Generates the directional network of orchestration vectors from any score in musicxml format. Use orchestralScore() to import the score data as sequence. Returns nodes and edges as Pandas DataFrames; average degree, modularity and partitioning of the network.
  - `seq (int)` list of orchestration vectors extracted from the score
- `def orchestralVectorColor(orch,dnodes,part,color=plt.cm.binary)`
  Plots the sequence of the orchestration vectors color-coded according to the modularity class they belong. Requires the output of orchestralNetwork()
  - `seq (int)` list of orchestration vectors extracted from the score

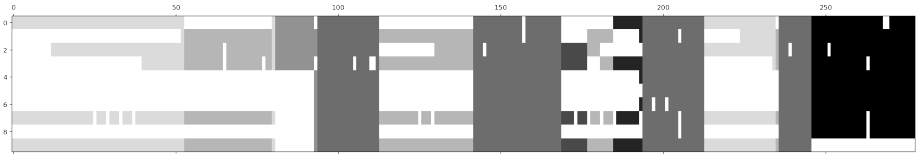

Fig. 1. Orchestration map of the first movement (Allegro) of J.S. Bach's Brandenburg Concerto n. 2, BWV 1047, as produced by the `orchestralVectorColor` function. Different shades of gray represent the sections of similar orchestral color as measured by their modularity class in the network.

Sound classification. The sound classification section of `timbrePy` comprises of modules for specific sound analysis that are based on the `librosa` python library for audio signal processing. We refer the interested reader to the `librosa` documentation at `https://librosa.github.io/librosa/index.html`. Fore a more complete discussion on the descriptors defined in `MUSICNTWRK` please refer to the work in Ref. [2]. Specific audio signal processing functions are:

- `def computeMFCC(input_path,input_file,barplot=True,zero=True)`
  read audio files in repository and compute a normalized MEL Frequency Cepstrum Coefficients and single vector map of the full temporal evolution of the sound as the convolution of the timeresolved MFCCs convoluted with the normalized first MFCC component (power distribution). Returns the list of files in repository, MFCC0, MFCC coefficients.
  - `input_path (str)` path to repository
  - `input_file (str)` filenames (accepts "*")
  - `barplot (logical)` plot the MFCC0 vectors for every sound in the repository
  - `zero (logical)` If False, disregard the power distribution component.
- `def computePSCC(input_path,input_file,barplot=True,zero=True)`
  Reads audio files in repository and compute a normalized Power Spectrum Frequency Cepstrum Coefficients and single vector map of the full temporal evolution of the sound as the convolution of the time-resolved PSCCs convoluted with the normalized first PSCC component (power distribution). Returns the list of files in repository, PSCC0, PSCC coefficients. Other variables as above.
- `def computeStandardizedMFCC(input_path,input_file,nmel=16, nmfcc=13,lmax=None,nbins=None)`
  read audio files in repository and compute the standardized (equal number of samples per file) and normalized MEL Frequency Cepstrum Coefficient. Returns the list of files in repository, MFCC coefficients, standardized sample length.
  - `nmel (int)` number of Mel bands to use in filtering
  - `nmfcc (int)` number of MFCCs to return
  - `lmax (int)` max number of samples per file
  - `nbins (int)` number of FFT bins
- `def computeStandardizedPSCC(input_path,input_file,nmel=16, psfcc=13,lmax=None,nbins=None)`
  read audio files in repository and compute the standardized (equal number of samples per file) and normalized Power Spectrum Frequency Cepstrum Coefficients. Returns the list of files in repository, PSCC coefficients, standardized sample length.
  Variables defined as for MFCCs.
- `def timbralNetwork(waves,vector,thup=10,thdw=0.1)`
  generates the network of MFCC vectors from sound recordings. Returns the nodes and edges tables as pandas DataFrames
  - `seq (float)` list of MFCC0 vectors
  - `waves (str)` names of sound files

Machine Learning Models. The definition of machine learning models for sound recognition requires standard techniques of data science (like the separation of data entries in training and testing sets, definition of neural network architectures, etc.) that will not be discussed here. Basic knowledge of Keras is also

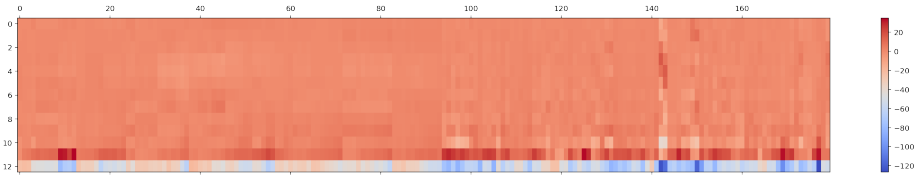

Fig. 2. Map of the MFCC0 for a repository of 180 impact sounds.

assumed. `MUSICNTWRK` module `timbrePy` contains many auxiliary functions to deal with such tasks. Here we limit to report the API for the main machine learning functions:

- `def trainNNmodel(mfcc,label,gpu=0,cpu=4,niter=100,nstep=10, neur=16,test=0.08,num_classes=2,epoch=30,verb=0,thr=0.85,w=False)` train a 2 layer neural network model on the ful MFCC spectrum of sounds. Returns: model,training and testing sets,data for re-scaling and normalization,data to asses the accuracy of the training session.
  - `mfcc (float)` list of all the MFCCs (or PSCCs) in the repository
  - `gpu, cpu (int)` number of GPUs or CPSs used for the run
  - `niter (int)` max number of model fit sessions
  - `nstep (int)` how often the training and testing sets are redefined
  - `neur (int)` number of neurons in first layer (it is doubled on the second layer
  - `test (float)` defines the relative size of training and testing sets
  - `num_classes=2 (int)` dimension of the last layer
  - `epoch (int)` number of epochs in the training of the neural network
  - `verb (int)` verbose - print information during the training run
  - `thr (float)` keep the model if accuracy is ¿ test
  - `w (logical)` write model on file if accuracy is above `thr`
- `def trainCNNmodel(mfcc,label,gpu=0,cpu=4,niter=100,nstep=10, neur=16,test=0.08,num_classes=2,epoch=30,verb=0,thr=0.85,w=False)` train a convolutional neural network (CNN) model on the full MFCC/PSCC spectrum of sounds. Returns: model,training and testing sets,data for re-scaling and normalization,data to asses the accuracy of the training session. Parameters are defined as above.

2.4 `sonifiPy`

`sonifiPy` contains functions for the sonification of data in multi-column or csv format and produces output as WAV (it requires an installation of csound and direct reference to the `ctcsound` module -), or musicxml or MIDI. Two sonification protocols are available: spectral - data are mapped to a single sound using subtractive synthesis (FIR filter); and linear - individual data points are mapped to pitches in a time-series structure. See Ref. [3, 4] for a complete description of this protocol. `sonifiPy` contains:

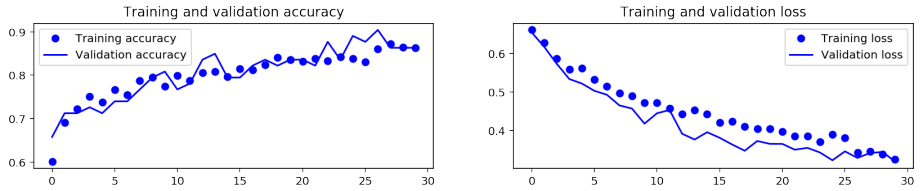

Fig. 3. Training and validation accuracy and loss in a typical Neural Network model learning run

- `def r_1Ddata(path,fileread)`
  Read data file in a multicolumn format (csv files can be easily put in this format using Pandas). Returns the data values as (x,y).
  - `path (str)` path to data file
  - `fileread (str)` data file
- `def i_spectral2(xv,yv,itime,path='./',instr='noise')`
  Use subtractive synthesis to sonify data structure. Returns the sound file.
  - `xv,yv (float)` data structure to sonify
  - `path (str)` path to data file
  - `fileread (str)` data file
- `def i_time_series(xv,yv,path='./',instr='csb701')`
  Use csound instruments to sonify data structures as time-series. Returns the sound file.
  - `xv,yv (float)` data structure to sonify
  - `path (str)` path to data file
  - `fileread (str)` data file
  - `instr (str)` csound instrument (it can be modified by user)
- `def MIDImap(pdt,scale,nnote)`
  Data to MIDI conversion on a given scale defined in scaleMapping (see below). Returns the MIDI data structure.
  - `pdt (float)` data structure mapped to MIDI numbers
  - `scale (float)` scale mapping (from scaleMapping)
  - `nnote (int)` number of notes in the scale (from scaleMapping)
- `def scaleMapping(scale)`
  Scale definitions for MIDI mapping. Returns: scale, nnote (see above).
- `def MIDIscore(yvf,dur=2,w=None,outxml='./music',outmidi='./music')`
  Display score or writes to file
  - `yvf (float)` data structure mapped to MIDI numbers (from MIDImap)
  - `dur (int)` reference duration
  - `w (logical)` if True writes either musicxml or MIDI file)
- `def MIDImidi(yvf,vnorm=80,dur=4,outmidi='./music')`
  Display score or writes to file
  - `yvf (float)` data structure mapped to MIDI numbers (from MIDImap)
  - `vnorm (int)` reference velocity
  - `outmidi (str)` MIDI file

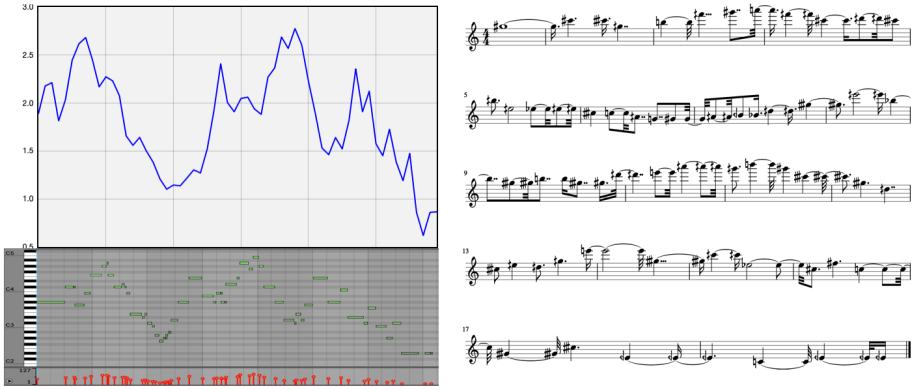

Fig. 4. Data, MIDI map and score from the sonification protocol in `MIDIscore`

The most computationally intensive parts of the modules can be run on parallel processors using the MPI (Message Passing Interface) protocol. Communications are handled by two additional modules: `communications` and `load_balancing`. Since the user will never have to interact with these modules, we omit here a detailed description of their functions.

Finally, a full set of examples is provided as Jupyter notebooks with the distribution.

3 Conclusions and acknowledgments

We have presented the API for the `MUSICNTWRK` software package. The software is freely available under GPL 3.0 and can be downloaded at www.musicntwrk.com. We acknowledge the support of Aix-Marseille University, IMéRA, and of Labex RFIEA+. Finally, we thank Richard Kronland-Martinet, Sølvi Ystad, Mitsuko Aramaki, Jon Nelson, Joseph Klein, Scot Gresham-Lancaster, David Bard-Schwarz, Roger Malina and Alexander Veremyer for useful discussions.

References

1. Buongiorno Nardelli, M.: Topology of Networks in Generalized Musical Spaces. https://arxiv.org/abs/1905.01842, 2019.
2. Buongiorno Nardelli, M., Aramaki, M., Ystad, S., Kronland-Martinet, R.: *in preparation*, 2019
3. Buongiorno Nardelli, M.: materialssoundmusic: a computer-aided data-driven composition environment for the sonification and dramatization of scientific data streams. International Computer Music Conference Proceedings, **356** (2015).
4. Buongiorno Nardelli, M.: Beautiful Data: Reflections for a Sonification and Post-Sonification Aesthetics, in Leonardo Gallery: Scientific Delirium Madness 4.0, Leonardo **51**(3), 227–238 (2018).