You can't un-neighbor your neighbor

For player piano and electronics (~15min)

Introductory notes

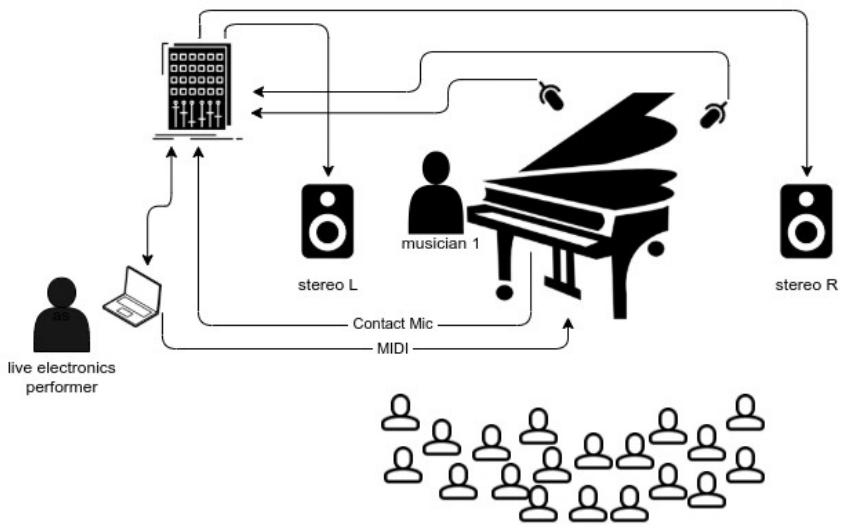
> *"Most personalized filters are based on a three-step model. First, you figure out who people are and what they like. Then, you provide them with content and services that best fit them. Finally, you tune to get the fit just right. Your identity shapes your media. There's just one flaw in this logic: Media also shape identity. And as a result, these services may end up creating a good fit between you and your media by changing. . . you."*
>
> *"The Filter Buble", Eli Parisier, 2011*

You can't un-neighbor your neighbor is a 15-minute piece composed in 3 scenes. The piece requires a prepared piano player (i. e. A *Disklavier)*, a computer, a pair of speakers, a pianist, and a live electronics performer. The player piano is simultaneously controlled by a live-generated *performance-rnn* and by a live electronics performer manipulating a set of algorithms. The *performance-rnn* is generated through a provided *Python* script which uses a pre-trained *Magenta PerformanceRNN* model to produce the MIDI files.

The pianist follows a minimal set of instructions for preparing and playing the piano together with the algorithms, all throughout the performance.

Scene

The figure below illustrates how the stage should be organized for the execution of this piece. Although not explicitly indicated in the diagram, the live electronics performer is encouraged to use an external audio device and a MIDI controller to manipulate both the player piano and the effects, as described in the next section.



Setup requirements

Hardware

- A computer with an audio/MIDI interface
- A MIDI controller
- A pair of speakers
- A digital player piano (i. e. a *Disklavier*)
- Two microphones to amplify the player piano (on the soundboard)
- Two contact microphones to amplify the mechanical structure under the player piano

Software

- GNU/Linux distribution (might also run on other systems)
- Python version 3 and libraries installed as documented in the git repository located at https://github.com/tiagovaz/detaching
- Carla or similar audio plugins host with a similar effects chain as illustrated in the table bellow

General instructions

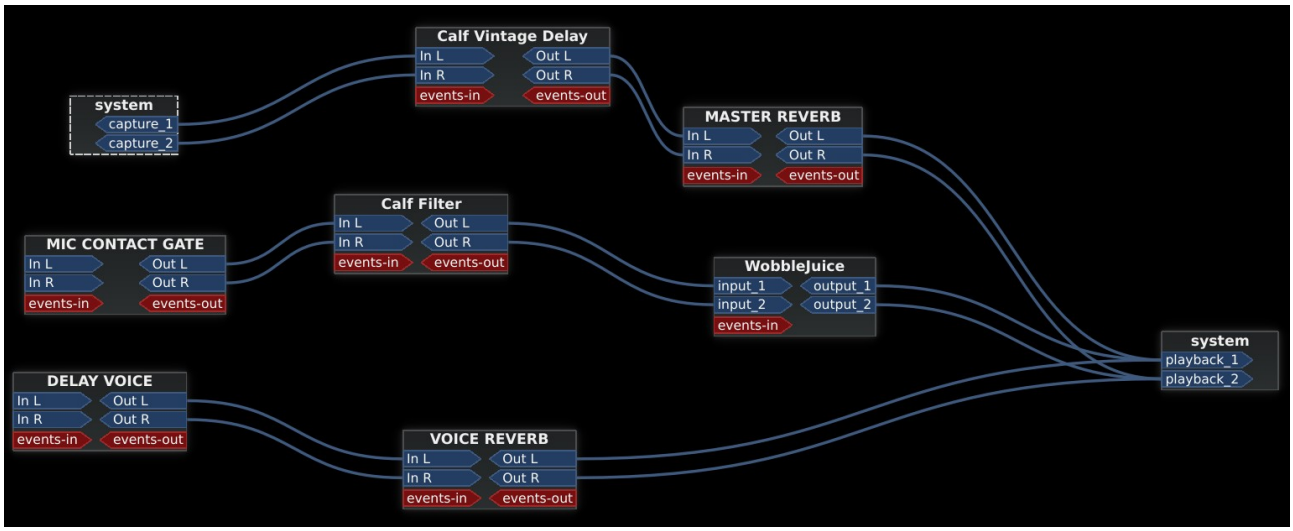
Once all hardware and software are set up by the instructions provided in the repository, two steps should be followed for the performance to take place:

1. **Generate the *performance-rnn* MIDI sequences to be played on the piano.** In this step, all the needed MIDI files are generated through the pre-trained models available in a bundle format ("`bach.mag`" and "`debussy.mag`"). The *Python* script which generates these files is the "`performance_generator.py`" and should be called without parameters. It will generate 10 *performances-rnn* for each scene. The characteristics of each model used by this script (and how they are used to generate sequences for each scene) is illustrated in the table bellow:

	Act 1	Act 2	Act 3
Magenta model	PerformanceRNN	PerformanceRNN	PerformanceRNN
Model type	MPD	MPD	MPD
Input composer(s)	Claude Debussy	J. S. Bach	Claude Debussy
N input performances	29	147	29
Input prime	–	BWV802	–
Training steps	23785	303140	23785
Accuracy	0.7370561	0.71957767	0.7370561
Loss	0.78055185	0.790744	0.78055185
Training time	8.3h	72.4h	8.3h
Output pitch distribution	[0, 1, 0, 0, 2, 1, 0, 1, 1, 0, 1, 1]	[10, 1, 3, 10, 15, 1, 10, 10, 1, 10, 2, 10]	[0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0]
Output temperature	1.4	1.3	0.7
Output notes per second	5	5	20
MIDI duration (num_steps)	120min	300min	120min

2. **Run the performance code.** A parameter indicating the number of the scene (1, 2, or 3) should be provided to the script "performance.py". For instance, for scene 1: "`python3 performance.py 1`". Once the code starts running, the live electronics performer controls the effects and, to some extent, the MIDI data being sent to the piano, according to the attached score. A *XML* file containing the configuration of audio objects and MIDI

assignments is provided in the repository ("`carla_patch.carxp`"). This file is to be used by *Carla audio plugin host* software. Any other similar software can be used as long as the following chain of effects is respected:



The audio effects in *Carla* can be assigned to a MIDI controller through its graphical interface. The MIDI manipulation can be assigned by editing the "`performance.py`" script, as documented in the code, by attributing the CC values to the variables listed in the above table.

The following picture and table might be useful as a guidance for this configuration, which supports a *nanoKONTROL2* controller by *Korg* and was used in the video-recorded performance available in https://archive.org/details/you-cant-uneighbor-your-neighbor.

Label	Source	Var name / plugin name	Action
P ON	Python code	PEDAL_ON	*Disklavier* sustain pedal ON
P OFF	Python code	PEDAL_OFF	*Disklavier* sustain pedal OFF
WOBBLE	Carla	*Wobble Juice Plugin*	Piano mechanics *Wobblejuice* effect gain
DL WET	Carla	*Calf Vintage Delay*	Piano soundboard delay dry/wet
REV	Carla	*Calf Reverb*	Piano soundboard reverb gain
DL FEED	Carla	*Calf Vintage Delay*	Piano soundboard delay feed

FL RES	Carla	*Calf Filter*	Piano mechanics filter resonance
FL FREQ	Carla	*Calf Filter*	Piano mechanics filter frequence
DL SPEECH	Carla	*Calf Vintage Delay*	Speech (scene 3 only) delay dry/wet
SPEECH VOL	Carla	*Carla Input Gain*	Speech volume (scene 3 only)
PAT VOL	Python code	PATTERN_VOL	Controls the MIDI velocity average of algorithmic musical patterns sent to the *Disklavier*
PAT SPEED	Python code	PATTERN_SPEED	Controls the density (or *tempo*) of MIDI data of algorithmic musical patterns sent to the *Disklavier*
CHAOS	Python code	CHAOS	Adds chaos (random notes) to the *performance-rnn* sequence being played on the *Disklavier*
PIANO GEN	Python code	PIANO_GEN	Skips notes (replaces by silence) from the *performance-rnn* sequence being played on the *Disklavier*

Per scene instructions

This is a highly improvised piece, where both the pianist and the live electronics performer follow very little instructions in order to accompany the AI-driven part being played. A sub-theme taken from the quotation in the beginning of this score reflects the composer's intentions and should be kept in mind as inspiration.

The following instructions are nothing more than an open guide for the execution of the piece. This comes from a collaborative work between composer Tiago Vaz and pianist Daniel Áñez, during rehearsals and the performance recorded in December 2021 (https://archive.org/details/you-cant-uneighbor-your-neighbor).

Scene 1 (3-4 min)

> *"First, you figure out who people are and what they like."*

This scene starts with a performance-rnn generated from a model trained against a dataset of 29 pieces from Claude Debussy. It plays on average 5 notes per second, has a temperature of 1.4 – being the most "creative" among the three scenes – and follows a pitch distribution represented as [0, 1, 0, 0, 2, 1, 0, 1, 1, 0, 1, 1]. This distribution is structured as a *Python* list, in wich numbers represent the probability of a note being played, from C to B. In this scene C = 0, C# = 1, D = 0, D# = 2 and so forth. Given the hight level of "creativity", these numbers are approximative. So, C = 0 doesn't necessary mean "C is not to be played". Instead, it can be read as "C is much less likely to be played than D# - but still possible to be played".

This scene is resonant, slow *tempo* and rich in wide-range chords. The musicians are encouraged to attentively observe the sounds and try to *extend* them rather than adding new sounds. A natural direction to long/continuous sounds should be followed. In the recorded video, this was accomplished with the aid of *e-bows* resonating piano strings on the soundboard. The whole scene shouldn't go over a *mezzo-forte* dynamics.

During Scene 1, the live electronics performer mostly improvises with "PIANO GEN" and "CHAOS" generators as described in the table above.

Scene 2 (6-7 min)

"Then, you provide them with content and services that best fit them."

The *performance-rnn* which guides this scene was generated by a model trained against 147 pieces of J. S. Bach. Like Scene 1, it plays on average 5 notes per second, and has a little less "creativity", with temperature set to 1.3. Its pitch distribution is represented as [10, 1, 3, 10, 15, 1, 10, 10, 1, 10, 2, 10]. Unlike the other scenes, the *performance-rnn* for Scene 2 is additionally fed with with a *primer*, which is the theme of *Duetto in E minor* (BWV 802). A music sheet with the first page of this piece should be on the music desk.

Once the code starts running, the theme of BWV 802 is automatically played. The pianist just observes, getting ready to react only a few bars later. This reaction might come from an attempt to create a dialogue with the machine. A *flawed* counterpoint results from this attempt. Little by little, the gestures develop toward another musical universe. This evolution reveals itself initially through modulations, then moving toward new sonorities on the prepared piano's soundboard.

Except for the very beginning of this scene, most of MIDI data sent to the *Disklavier* is manipulated by the live electronics performer, who triggers and controls algorithms that generate pseudo-random motives. This is achieved using "PAT VOL" and "PAT SPEED" parameters. Compared to Scene 1, here the musicians are encouraged to increase the dynamics to *forte* or even *fortíssimo*, yet avoiding aggressive gestures and, like in Scene 1, ending the piece exploring resonance and continuous sounds. Also, in this scene, the live electronics performer is encouraged to make further use of "WOBBLE" and "DL WET" effects.

Scene 3 (5-6 min)

"Finally, you tune to get the fit just right."

This scene, certainly the most "aggressive" among the three, is based on *performances-rnn* generated *in extremis*, comprising an unusual pitch distribution of [0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0] at 20 notes per second on average. The level of creativity is lower than seen in previous scenes (0.7). This means that notes marked with low probability are more unlikely to be played. Also, for Scene 3 the musicians should use a higher amount of effects, as well as an extra incitement material for the performance, in the form of audible excerpts extracted from a conversation which inspired the piece.

This conversation comes from a discussion between historian Yuval Harari and Facebook CEO Mark Zuckerberg, and is part of a series of "public discussions about the future of technology in society", featured by Zuckerberg himself. The video is publicly available on Harari's Youtube channel.[1]

The scene starts with Zuckerberg's proclaiming *"Hey everyone, this year I'm doing a series of public discussions on uh... the future of the Internet and Society and some of the big issues around that..."*, followed by the generated *performance-rnn.* From its very beginning, this scene reveals a certain *anxiety*, *insistence*, and *lack of direction*. Musicians are encouraged to musically represent these intentions through *repetition*, *contrast* and *mechanical/powerful gestures.* Audio excerpts from the same conversation are played all over the piece, motivating immediate reactions from the musicians.

The only direction this scene takes is toward a dramatic *crescendo* in the end, when it goes back down to a non-dramatic repetitive gesture, which should somewhat represent that we ultimately have been tuned to *fit just right*.

1 *Mark Zuckerberg & Yuval Noah Harari in Conversation*: https://www.youtube.com/watch?v=Boj9eD0Wug8

main.py

```
from pyo import *
import mido
import random
import time

####### MIDI CC variables ##########

PEDAL_ON = 43
PEDAL_OFF = 44
PATTERN_ON = 37
PATTERN_OFF = 53
PATTERN_VOL = 21
PATTERN_SPEED = 5
CHAOS = 6
RNN_ON = 39
RNN_OFF = 55
PIANO_GEN = 7

####################################

# DEV means using internal midi ports
DEV = True

s = Server(audio='jack', duplex=0)

# Randomly set the midi file to be picked from a subset
midi_pick = str(random.randint(1, 5))

# Set the movement to be played:
mov = 1

# 1)
mov_01_midi_file = 'MIDI/01/' + random.choice(os.listdir("MIDI/01/"))

# 2)
mov_02_midi_file = 'MIDI/02/' + random.choice(os.listdir("MIDI/02/"))

# 3)
mov_02_midi_file = 'MIDI/03/' + random.choice(os.listdir("MIDI/03/"))

s.setMidiOutputDevice(99)
s.setMidiInputDevice(99)
s.boot().start()

if DEV is True:
    port = mido.open_output('Midi Through:Midi Through Port-0 14:0')
else:
    port = mido.open_output('io|2:io|2 MIDI 1 28:0')

```

```
conversation = "conversation.flac"
sf = SfPlayer(conversation, speed=1, loop=True, mul=0).stop()

midi_play = False
pattern_play = False

def midi_scanner(ctlnum, midichnl):
    global midi_play
    global pattern_play
    if ctlnum == RNN_ON:
        midi_play = True
    elif ctlnum == RNN_OFF:
        midi_play = False
    elif ctlnum == PATTERN_ON:
        pattern_play = True
    elif ctlnum == PATTERN_OFF:
        pattern_play = False
    elif ctlnum == PEDAL_ON:
        s.ctlout(64, 127)
    elif ctlnum == PEDAL_OFF:
        s.ctlout(64, 0)
    elif ctlnum == 42:
        sf.setMul(0)
    elif ctlnum == 41:
        sf.out()
        sf.setMul(0.02)


# Listen to controller input.
midi_in = CtlScan2(midi_scanner, toprint=True)

counter = 0
counter_tick = 0

velocity = Randi(min=50, max=100, freq=1)

ctl_skip_notes = Midictl(ctlnumber=[PIANO_GEN], minscale=0, maxscale=10, init=0)
ctl_chaos = Midictl(ctlnumber=[CHAOS], minscale=0, maxscale=10, init=0)

########################## PATTERN ###################################

ctl_pattern_time = Midictl(ctlnumber=[PATTERN_SPEED], minscale=0.05, maxscale=1, init
    =0.125)
ctl_pattern_velocity = Midictl(ctlnumber=[PATTERN_VOL], minscale=0, maxscale=70, init=21)

l = Phasor(.4)
pitch = XnoiseMidi(12, freq=8, x1=1, x2=l, scale=0, mrange=(21, 108))

# Global variable to count the down and up beats.
count = 0

```

```
100 def midi_event():
101     if pattern_play == True:
102         global count
103         pit = int(pitch.get())
104         if count == 0:
105             vel = random.randint(int(ctl_pattern_velocity.get()), int(
    ctl_pattern_velocity.get()) + 30)
106             dur = 500
107         else:
108             vel = random.randint(int(ctl_pattern_velocity.get()), int(
    ctl_pattern_velocity.get()) + 10)
109             dur = 125
110         count = (count + 1) % random.randint(2,5)
111         pat.setTime(float(ctl_pattern_time.get()))
112         s.makenote(pitch=pit, velocity=vel, duration=dur)
113 
114 pat = Pattern(midi_event, time=0.125).play()
115 
116 ###################### MAIN LOOP ######################################
117 
118 while True:
119     if mov == 1:
120         mid = mido.MidiFile(mov_01_midi_file)
121         for message in mid.play():
122             if (counter_tick % 100) == 0:
123                 tick = mido.Message('note_on', channel=0, note=random.choice([10, 99,
    101]), velocity=80, time=6.2)
124                 tick_off = mido.Message('note_off', channel=0, note=99)
125                 counter_tick = 0
126 
127             if not message.is_meta:
128                 # PAUSE
129                 if midi_play is True:
130                     # SKIP NOTES
131                     if random.randint(0, int(ctl_skip_notes.get())) == 0:
132                         print('CONTROLLER IN %d' % int(ctl_skip_notes.get()))
133                         if random.randint(0, int(ctl_chaos.get())) == 0:
134                             print('CONTROLLER IN %d' % int(ctl_skip_notes.get()))
135                             port.send(message)
136                         else:
137                             # CHAOS
138                             s.makenote(pitch=random.randint(48,82), velocity=random.
    randint(20,50), duration=random.randint(1,1000))
139                 else:
140                     while midi_play is False:
141                         print('wating for resume...')
142                         time.sleep(1)
143             counter = counter + 1
144             counter_tick = counter_tick + 1
145             print(counter_tick)
146 
```

```
147     elif mov == 2:
148         mid = mido.MidiFile(mov_03_midi_file)
149         for message in mid.play():
150             if counter == 500:
151                 delay = random.choice([10,12,15])
152                 time.sleep(delay)
153             if (counter_tick % 100) == 0:
154                 tick = mido.Message('note_on', channel=0, note=random.choice([10, 99,
    101]), velocity=80, time=6.2)
155                 tick_off = mido.Message('note_off', channel=0, note=99)
156                 port.send(tick)
157                 port.send(tick_off)
158                 counter_tick = 0
159 
160             if not message.is_meta:
161                 if message.type == 'note_on':
162                     message.velocity = max(0, message.velocity + int(velocity.get()))
163                 if counter > 50: # modulate
164                     if message.note == 69:
165                         message.note = 70
166                 if counter > 100: # modulate
167                     if message.note == 74:
168                         message.note = 75
169                 if counter > 150: # do not play
170                     if message.note == 76:
171                         continue
172                 port.send(message)
173                 # print(message)
174             counter = counter + 1
175             counter_tick = counter_tick + 1
176             print(counter_tick)
177 
178     elif mov == 3:
179         mid = mido.MidiFile(mov_03_midi_file)
180         for message in mid.play():
181             if midi_play is True:
182                 if message.type == 'note_on' or message.type == 'note_off':
183                     port.send(message)
184             else:
185                 while midi_play is False:
186                     print('wating for resume...')
187                     time.sleep(1)
```