lattice chains

for two performers and 9 channel modular wall/sound system

John Eagle
2018

The piece consists of five possible sections (a minimum of three should be done). In each section the 8 walls and one stationary speaker will produce a chain of tones in intervals matching each harmonic plane. The primes of each axis are condensed as to limit the intervals to a maximum of one octave (The prime 3 is reduced to 3/2, 5 to 5/4, 7 to 7/4, 11 to 11/8, and 13 to 13/8). A program randomly generates each tone chain. The performers may amplify the tone of each wall by standing in front of the sensor on the front face to allow easier identification of each interval. They arrange the walls according to each interval, beginning with the interval between the stationary speaker and wall 1 (in the first section, if the wall were one octave higher it would point in the designating ascending direction on the y-axis. If it were an octave lower, it would point in the descending direction. A fifth higher it would point in the ascending direction on the x-axis. A fifth lower it would point in the descending direction). They then work until all 8 walls have been arranged properly and then trigger a new chain to be generated and begin the process again. Performers may hum/sing pitches quietly to help themselves pick out the correct intervals and take as much time as necessary to arrange the walls accurately.

Sections:

I. 2, 3 plane
II. 3, 5 plane
III. 5, 7 plane
IV. 7, 11 plane
V. 11, 13 plane

20’ / 5 sections = 4’ each
20’ / 4 sections = 5’ each
20’ / 3 sections = 7’ each

2 performers (A + B).

Proceed in this fashion: (in duets A always takes walls 1-4, B takes 5-9)

||: A solo chain | A + B duet chain | B solo chain | A + B duet chain :||

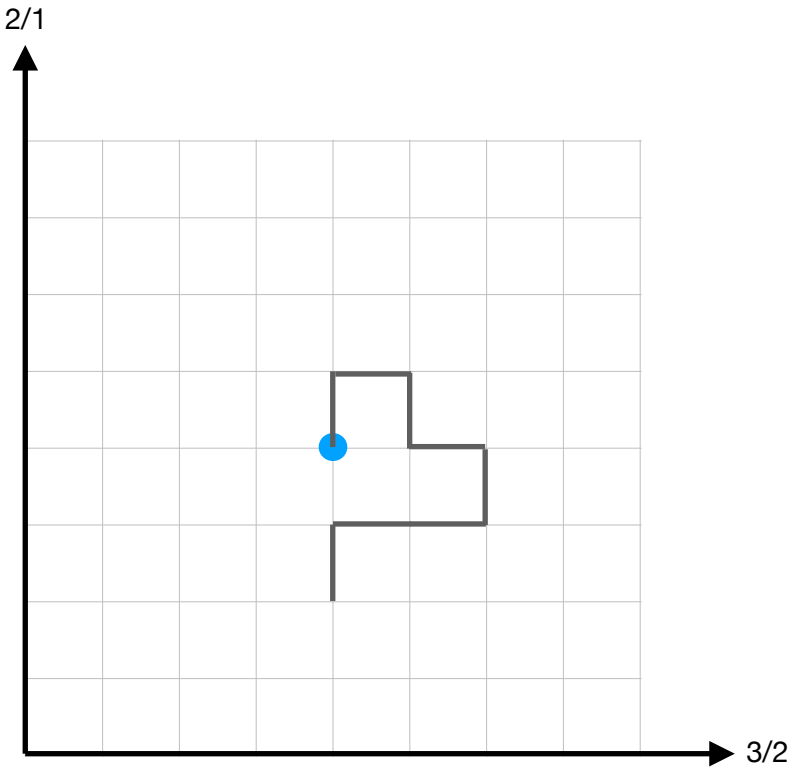
At designated timings, finish the current chain and then trigger switch to new prime pair. At end, finish current chain and then terminate sound

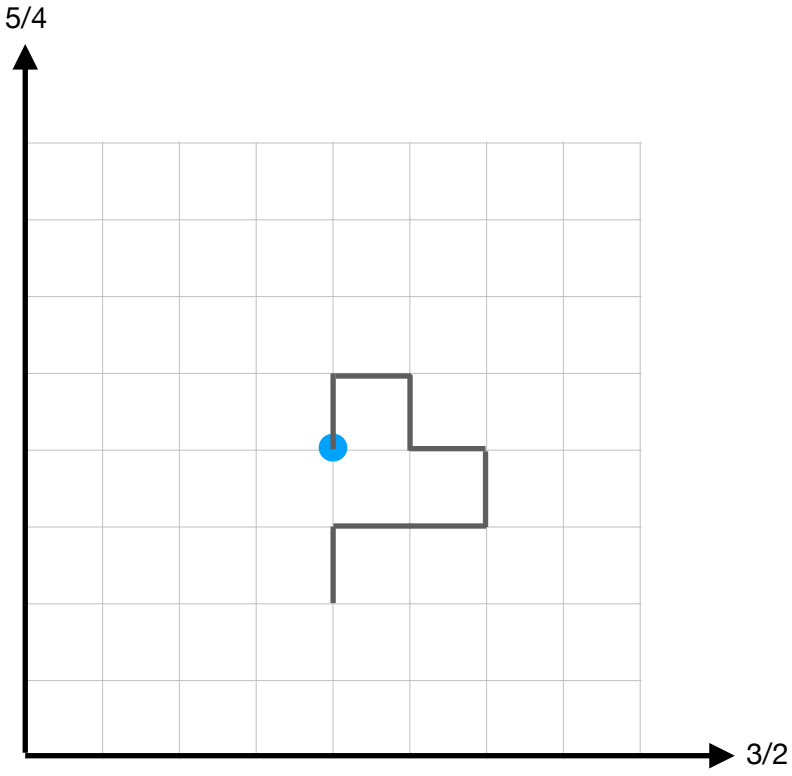
5 sections:
0’ 4’ 8’ 12’ 16’ 20’

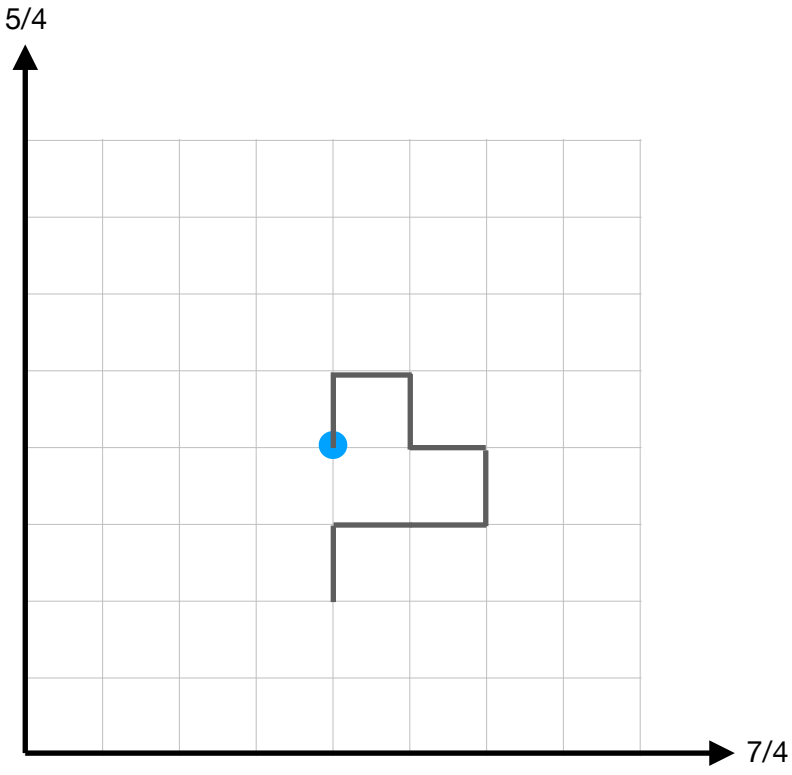
4 sections:
0’ 5’ 10’ 15’ 20’

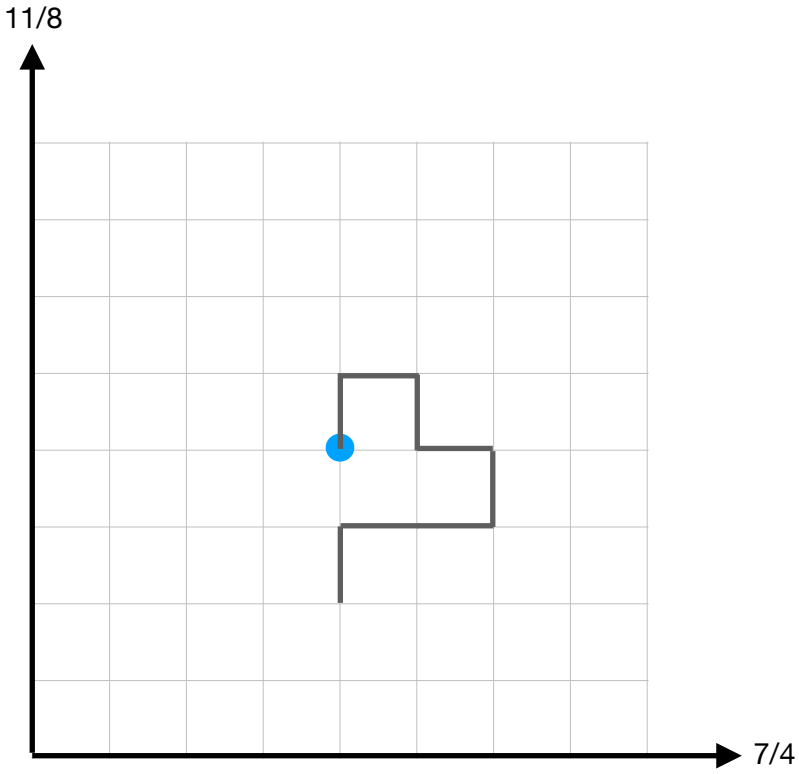
3 sections:
0’ 7’ 14’ 21’

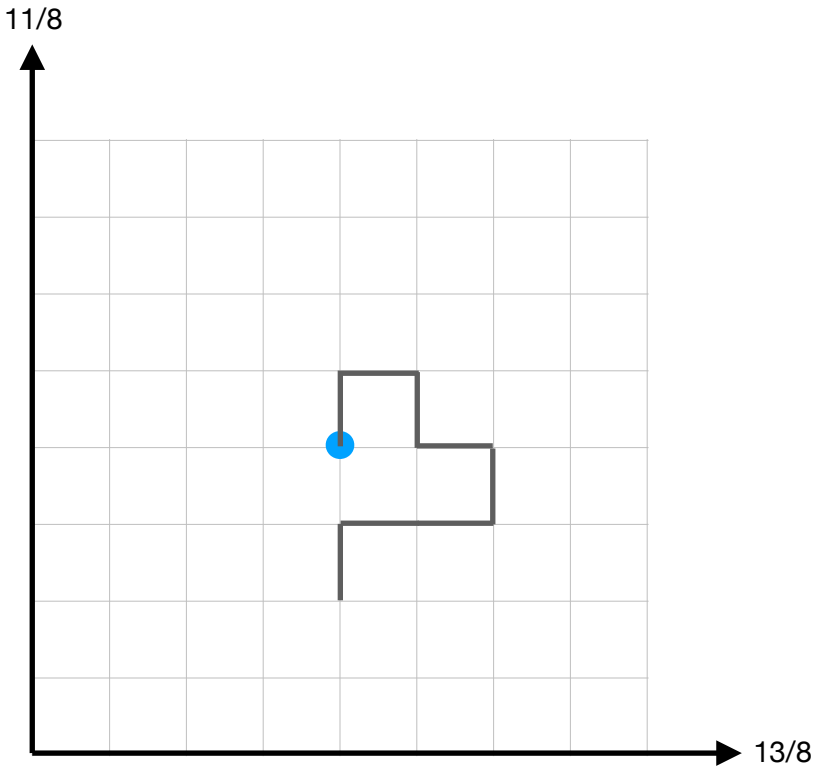
Here is an example chain illustrated on the 5 different harmonic planes (2, 3; 3, 5; 5, 7; 7, 11; 11, 13) so that one can see how the different harmonic intervals in each section might generate each shape. Remember that the speaker and its tone represent the origin of the chain. Thus each wall's tone indicates the endpoint of that wall (not the beginning). This is why one should start with wall 1 and work in ascending order to be most efficient. In the duets, person B will be able to assemble walls 5-8 in the right arrangement, but will need to move the position to connect with wall 4 after person A finishes.




5/4
3/2


5/4
7/4


11/8
7/4


11/8
13/8

Source code for performance:

8 walls (about 4'x4') on wheels are equipped with Raspberry Pi's connected to ultrasonic sensors and speakers. One stationary speaker, placed at the center of the space, is connected to control computer (9 total speakers). They are all on one wifi network. The control computer runs 'lattice_dev.py', which is the main program that generates a new chain of tones upon UI and changes primes when prompted. 'chains.py' contains the math functions for generating each chain and 'network_functions.py' contains the functions for sending control data to each Raspberry Pi/speaker.

'local_receiver.ck' is the ChucK script on the control computer which generates sine tone for stationary speaker.

'oscDistance.py' is run on each Raspberry Pi which checks to see if a person has stepped in front of wall. 'receiver.ck' (also on each Pi) will generate sine tone when triggered by sensor reading.

Several of these scripts contain additional functionality for other pieces created for the same system.

```
"""
lattice_dev.py
John Eagle
"""

import argparse
from pythonosc import udp_client, osc_message_builder, dispatcher,
osc_server
import threading
import pygame, sys
import network_functions as nf
import chains
from time import sleep


# TO DO: add ability to change to next chain by standing in front
of 1 * 8 for x seconds
# is there another method that could work for changing interval? or
just have x number of times for each prime?

###
def parse_sensors(address, hostname, val):
    #print("pi: ", hostname, "val: ", val)
    if hostname in wall_names:
        wall_index = wall_names.index(hostname)
        wall_vals[wall_index] = val
        #print(wall_vals)

def make_server(IP, port, address):
    # creates server on thread for receiving osc messages
    # set IP and port to listen on
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", default=IP, help="The ip of the OSC
    server")
    parser.add_argument("--port", type=int, default=port,
                        help="The port the OSC server is listening
                        on")
    args = parser.parse_args()

    # the thread that listens for the OSC messages
    dispatcherX = dispatcher.Dispatcher()
    dispatcherX.map(address, parse_sensors)

    server = osc_server.ThreadingOSCUDPServer((args.ip, args.port),
                                              dispatcherX)
    server_thread = threading.Thread(target=server.serve_forever)
```

```
    server_thread.start()

    print("Serving on {}".format(server.server_address))

    return server

def send_OscControl_data(freq_list, amp_list):
    # send sine freqs and amplitudes
    freq_message = "/sineFreq"
    amp_message = "/sineGain"

    for index, client in enumerate(wall_oscs):
        amp = amp_list[index]
        freq = freq_list[index]

        # amplitude envelope
        if freq < 200:
            amp *= 2.5
            #print('200', freq)
        elif freq < 300:
            amp *= 2.1
            #print('300', freq)
        elif freq < 400:
            amp *= 2
            #print('400', freq)
        elif freq < 700:
            amp *= 1.5
            #print('700', freq)
        elif freq < 1000:
            amp *= 1.25
            #print('1k', freq)
        elif freq < 2000:
            amp *= 0.5
            #print('2k', freq)
        elif freq < 3000:
            amp *= 0.5
            #print('3k', freq)
        elif freq < 4000:
            amp *= 0.6
            #print('4k', freq)
        else:
            pass
            #print('4k+', freq)

        # cap max amp
        if amp > 1.0:
```

```
            amp = 1.0

        # now send everything
        client.send_message(freq_message, freq)
        client.send_message(amp_message, amp)

        # uncomment these two lines to arpeggiate chord
        #sleep(1)
        #client.send_message(amp_message, 0)

def send_local_osc(client, freq, amp):
    freq_message = "/sineFreq"
    amp_message = "/sineGain"
    client.send_message(freq_message, freq)
    client.send_message(amp_message, amp)

def check_for_threshold(wall_vals, wall_amps):
    # checks sensor data to see if distance threshold has been
    passed
    threshold = 50 # cm
    min_amp = 0.05
    max_amp = 0.7

    for index, val in enumerate(wall_vals):
        if val < threshold:
            wall_amps[index] = max_amp
            send_OscControl_data(wall_freqs, wall_amps)
            print(index + 1, ' on')
        else:
            wall_amps[index] = min_amp
    return wall_amps

def send_test_values():
    client = int(input("wall num: ")) - 1
    freq = float(input("frequency: "))
    amp = float(input("amp: "))
    wall_oscs[client].send_message("/sineFreq", freq)
    wall_oscs[client].send_message("/sineGain", amp)

def get_pair(user_list, index_1):
    first_pair = user_list[index_1:index_1 + 2]
    return first_pair

def check_events(local_freq, wall_freqs, prime_pair,
prime_progression):

    for event in pygame.event.get():
```

```
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            shutdown()
        elif event.type == pygame.KEYDOWN:
            local_freq, wall_freqs, prime_pair, prime_progression =
            check_keydown_events(event, local_freq, wall_freqs,
            prime_pair, prime_progression)

    return local_freq, wall_freqs, prime_pair, prime_progression

def check_keydown_events(event, local_freq, wall_freqs, prime_pair,
prime_progression):
    if event.key == pygame.K_q:
        shutdown()
    # key triggers
    if event.key == pygame.K_RETURN:
        # get new chain
        print('return')
        local_freq, wall_freqs = make_chain()
    if event.key == pygame.K_LEFT:
        prime_progression -= 1
        if prime_progression < 0:
            prime_progression = 0
        prime_pair = get_pair(primes, prime_progression)
        print(prime_pair)
    elif event.key == pygame.K_RIGHT:
        prime_progression += 1
        if prime_progression > (len(primes) - 2):
            prime_progression = len(primes) - 2
        prime_pair = get_pair(primes, prime_progression)
        print(prime_pair)

    # add events for changing primes

    return local_freq, wall_freqs, prime_pair, prime_progression

def shutdown():
    # shutdown procedures
    print('Goodbye')
    local_server.shutdown()
    sys.exit()


def update_screen(screen):
    # draw screen
    screen.fill((0, 0, 0))
    pygame.display.flip()
```

```
    pygame.display.flip()

###

local_IP = "127.0.0.1"
john_MBP = "192.168.1.163" #"192.168.0.7"

# select runtime mode or dev mode
ui = input("1. Run Program (networked) 2. Dev. Mode (local): ")

if ui == "1":
    IP = john_MBP
    sending = "pis"
elif ui == "2":
    IP = local_IP
    sending = "local"

#prime_1 = float(input('prime 1: '))
#prime_2 = float(input('prime 2: '))
#primes = [prime_1, prime_2]
primes = [2, 1.5, 1.25, 1.75, 1.375, 1.625] # len = 6, 2nd to last
index = 4
prime_progression = 0
prime_pair = get_pair(primes, prime_progression)


# pi hostnames
wall_IPs = ['pione.local', 'pitwo.local', 'pithree.local',
'pifour.local',
            'pifive.local', 'pisix.local', 'piseven.local',
            'pieight.local']

# ports for sending and receiving
ping_port = 5000
rec_port = 12345
osc_port = 10001
local_sine = 10002
local_osc = nf.make_client(local_IP, local_sine)


# create one local client or 8 pi clients
wall_clients = []
wall_oscs = []
if sending == "pis":
    # for pinging sensors
    for wall_IP in wall_IPs:
        client = nf.make_client(wall_IP, ping_port)
```

```
        wall_clients.append(client)
    # for sending sine tone data
    for wall_IP in wall_IPs:
        client = nf.make_client(wall_IP, osc_port)
        wall_oscs.append(client)

elif sending == "local":
    client = nf.make_client(IP, rec_port)
    wall_clients.append(client)

local_server = make_server(IP, rec_port, "/w")

wall_names = ["pione", "pitwo", "pithree", "pifour", "pifive",
"pisix",
              "piseven", "pieight"]
wall_vals = [1000, 1000, 1000, 1000, 1000, 1000, 1000, 1000]
wall_amps = [0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2]

#chain = chains.make_quaternary_chain(8)
#freqs = chains.convert_to_freqs(chain, primes)
#local_freq = freqs[0]
#wall_freqs = freqs[1:]

pygame.init()
screen = pygame.display.set_mode((400, 400))
pygame.display.set_caption("Lattice Chain Pi Controller")

def make_chain():
    searching = True
    while searching:
        chain = chains.make_quaternary_chain(8)
        freqs = chains.convert_to_freqs(chain, prime_pair)
        if min(freqs) < 100.0:
            # too low start again
            continue
        elif min(freqs) > 400:
            # too high start again
            continue
        else:
            searching = False
            local_freq = freqs[0]
            # print(local_freq)
            wall_freqs = freqs[1:]
            print("primes:", prime_pair, "sines: ", wall_freqs)

    return local_freq, wall_freqs
```

```
264
265 local_freq, wall_freqs = make_chain()
266
267
268 #searching = True
269
270 # main program loop
271 while True:
272     # ping
273     for client in wall_clients:
274         if sending == "pis":
275             nf.send(client, "/w")
276         elif sending == "local":
277             # generate random reading
278             nf.send_reading(client, "/w", sending) #simulation use
  •             nf.send for real
279     # check for trigger
280     wall_amps = check_for_threshold(wall_vals, wall_amps)
281
282     # send local freq and amp
283     send_local_osc(local_osc, local_freq, 0.9)
284     # send frequency and amp to walls
285     print(wall_freqs)
286     send_OscControl_data(wall_freqs, wall_amps)
287
288     local_freq, wall_freqs, prime_pair, prime_progression = \
289         check_events(local_freq, wall_freqs, prime_pair,
  •         prime_progression)
290     update_screen(screen)
291
292
293     # testing
294     #send_test_values()
295
296     # how many chains for each prime set?
297
298     sleep(0.1)
299
```

```
"""
network_functions.py
john eagle
"""

import argparse
from pythonosc import udp_client, osc_message_builder, dispatcher,
osc_server
import threading
#import other_functions as of
from time import sleep
from random import randrange, choice

import pygame

import math


#### new functions from may 2018

def make_client(IP, port):
    # creates osc client
    # set IP and port
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", default=IP, help="The ip of the OSC
    server")
    parser.add_argument("--port", type=int, default=port,
                        help="The port the OSC server is listening
                        on")
    args = parser.parse_args()
    # make client
    client = udp_client.SimpleUDPClient(args.ip, args.port)
    return client

def send(client, msg1, msg2=0):
    # sends message to osc client
    # msg2 is a nul value just to make send work
    client.send_message(msg1, msg2)
    #print("sending...")

def send_reading(client, adr, mode):#self, junk):
    # we send the Pi's IP address as the OSC address
    # so the host computer knows which Pi sent a message
    packet = osc_message_builder.OscMessageBuilder(address=adr)

    # adds whichPi to the OSC message
```

```
    #hostname = socket.gethostname()

    # print(hostname)
    # for dev mode only
    if mode == 'local':
        pi = choice(["pione", "pitwo", "pithree", "pifour",
        "pifive", "pisix",
                     "piseven", "pieight"])
        val = randrange(0, 100)
        hostname = pi

    packet.add_arg(hostname, arg_type='s')

    # adds distance reading to the OSC message
    packet.add_arg(val, arg_type='f')

    # completes the OSC message
    packet = packet.build()

    # sends distance back to the host
    client.send(packet)

    #print("sending:", adr, hostname, val)

def parse_sensors(address, hostname, val):
    print("pi: ", hostname, "val: ", val)

def make_server(IP, port, address):
    # creates server on thread for receiving osc messages
    # set IP and port to listen on
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", default=IP, help="The ip of the OSC
    server")
    parser.add_argument("--port", type=int, default=port,
                        help="The port the OSC server is listening
                        on")
    args = parser.parse_args()

    # the thread that listens for the OSC messages
    dispatcherX = dispatcher.Dispatcher()
    dispatcherX.map(address, parse_sensors)

    server = osc_server.ThreadingOSCUDPServer((args.ip, args.port),
                                              dispatcherX)
    server_thread = threading.Thread(target=server.serve_forever)
    server_thread.start()
```

```
87
88      print("Serving on {}".format(server.server_address))
89
90      return server
91
92
93
94  #### old functions
95
96
97
98
99   # need to test this with walls running
100
101 def initialize_sensorReceiver_port(ctl_settings):
102
103     wallIPs = ctl_settings.wallIPs
104     port = ctl_settings.portGetSensorData
105     client_list = []
106     for IP in wallIPs:
107         wallIP = IP
108         parser = argparse.ArgumentParser()
109         parser.add_argument("--ip", default=wallIP, help="The ip of
    •    the OSC"
110                                                          "server")
111         parser.add_argument("--port", type=int, default=port,
112                             help="The port the OSC server is
    •                        listening on")
113         args = parser.parse_args()
114         client = udp_client.SimpleUDPClient(args.ip, args.port)
115         client_list.append(client)
116     ctl_settings.wallSensor_clients = client_list
117
118     #ip = ctl_settings.localIP
119     #port = ctl_settings.portGetSensorData
120     #parser = argparse.ArgumentParser()
121     #parser.add_argument("--ip", default=ip, help="The ip to listen
    •    on")
122     #parser.add_argument("--port", type=int, default=port,
123     #                    help="The port to listen on")
124     #args = parser.parse_args()
125
126     #receives two messages and a value (/w pione 216.3)
127     address = "/w"
128
129     # the thread that listens for the OSC messages
130     dispatcherX = dispatcher.Dispatcher()
```

```
    dispatcherX = dispatcher.Dispatcher()
    dispatcherX.map(address,
    print)#ctl_settings.update_wall_sensors)

    ctl_settings.server =
    osc_server.ThreadingOSCUDPServer((args.ip, args.port),
                                                        dispatcherX
                                                        )
    ctl_settings.server_thread = threading.Thread(
        target=ctl_settings.server.serve_forever)
    ctl_settings.server_thread.start()

    print("Serving on
    {}".format(ctl_settings.server.server_address))

def stopServer(ctl_settings):
    if ctl_settings.server:
        ctl_settings.server.shutdown()

def initialize_audioControl_port(ctl_settings): # sends to sender.ck
    IP = ctl_settings.localIP
    port = ctl_settings.portFeedbackControl # 7400
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", default=IP, help="The ip of the OSC"
                                                 "server")
    parser.add_argument("--port", type=int, default=port,
                        help="The port the OSC server is listening
                        on")
    args = parser.parse_args()
    ctl_settings.senderCK_client =
    udp_client.SimpleUDPClient(args.ip, args.port)

def initialize_OscControl_ports(ctl_settings): # need to test this
    wallIPs = ctl_settings.wallIPs
    port = ctl_settings.portOscControl
    client_list = []
    for IP in wallIPs:
        #wallIP = '--' + IP # if this doesn't work, use
        'default=wallIP' # will host names work?
        wallIP = IP
        parser = argparse.ArgumentParser()
        parser.add_argument("--ip", default=wallIP, help="The ip of
        the OSC"
                                                         "server")
        parser.add_argument("--port", type=int, default=port,
                            help="The port the OSC server is
                            listening on")
```

```
                                listening on")
        args = parser.parse_args()
        client = udp_client.SimpleUDPClient(args.ip, args.port)
        client_list.append(client)
    ctl_settings.wallOsc_clients = client_list

def initialize_sensorPing_ports(ctl_settings):
    wallIPs = ctl_settings.wallIPs
    port = ctl_settings.portPingSensors
    client_list = []
    for IP in wallIPs:
        wallIP = IP
        parser = argparse.ArgumentParser()
        parser.add_argument("--ip", default=wallIP, help="The ip of
        the OSC"
                                                          "server")
        parser.add_argument("--port", type=int, default=port,
                            help="The port the OSC server is
                            listening on")
        args = parser.parse_args()
        client = udp_client.SimpleUDPClient(args.ip, args.port)
        client_list.append(client)
    ctl_settings.wallSensor_clients = client_list

def initialize_video_port(ctl_settings):
    IP = ctl_settings.videoIP
    port = ctl_settings.portVideo
    parser = argparse.ArgumentParser()
    parser.add_argument("--ip", default=IP, help="The ip of the OSC"
                                                 "server")
    parser.add_argument("--port", type=int, default=port,
                        help="The port the OSC server is listening
                        on")
    args = parser.parse_args()
    ctl_settings.video_client = udp_client.SimpleUDPClient(args.ip,
    args.port)

def send_audioControl_data(ctl_settings, msg, val):
    # sends to sender.ck
    if ctl_settings.networkOn:
        ctl_settings.senderCK_client.send_message(msg, val)
    else:
        print(msg, val)

def send_OscControl_data(ctl_settings, freq_list=None): # need to
test this
    # add amplitude scaling?
```

```
    freq_message = "/sineFreq"
    amp_message = "/sineGain"

    for index, client in enumerate(ctl_settings.wallOsc_clients):
        amp = ctl_settings.wall_amps[index]
        # add coeffecient or function here to EQ amp val
        if switch == 'on':
            freq = freq_list[index]
            amp = ctl_settings.wall_amps[index]
            client.send_message(freq_message, freq)
            client.send_message(amp_message, amp)
        else:
            client.send_message(amp_message, 0)

def send_ternary_chain(ctl_settings, ternary_chain):
    """
    Sends freqs out to walls from ternary chain
    """
    freqs = of.convert_chain_to_freqs(ternary_chain, ctl_settings)
    send_OscControl_data(ctl_settings, freqs)
    print(freqs)

def send_OscControl_off(ctl_settings):
    freqs = [0, 0, 0, 0, 0, 0, 0, 0]        # debug this!
    send_OscControl_data(ctl_settings, 'off')

def ping_sensors(ctl_settings):
    msg = "/w"
    if ctl_settings.networkOn:
        for index, client in
        enumerate(ctl_settings.wallSensor_clients):
            client.send_message(msg, '') # is a val needed here?
    else:
        print(msg)

def send_brickplay(ctl_settings):
    wall_index = ctl_settings.count
    sample = str(randrange(1, 17))
    msg = '/brickPlay'
    if ctl_settings.networkOn:
        ctl_settings.wallOscClients[wall_index].send_message(msg,
        sample) # need to test this
    else:
        print(wall_index, msg, sample)

def sendVideoTrigger(ctl_settings, channel, val): # placeholder
```

```
    function for testing video OSC messages
253     msg = '/video/' + str(channel) # placeholders
254     if ctl_settings.networkOn:
255         ctl_settings.video_client.send_message(msg, val)
256     else:
257         print(msg, val)
258
```

```python
"""
chains.py
john eagle
"""

from random import randint, choice
from fractions import Fraction

# 0=left, 1=straight, 2=right
#   2       3       5       7       11      13
#   2       1.5     1.25    1.75    1.375   1.625
primes = [2, 1.5, 1.25, 1.75, 1.375, 1.625]


# initialize
freqs = [0, 0, 0, 0, 0, 0, 0, 0]
fundamental = 100

# quaternary chain functions
def is_even(integer):
    if integer % 2 == 0:
        is_even = True
    else:
        is_even = False
    return is_even

def make_quaternary_chain(chain_length): # input 8
    # need a chain of 8 for 9 pitches
    chain = [randint(0, 3)] # start with intial number 0 - 3
    for index, number in enumerate(range(chain_length - 1)):
        last_num = chain[index]
        options = [last_num]
        # if last_num is even, next_num must be odd or the same
        # if last_num is odd, next_num must be even or the same
        if is_even(last_num):
            options.append(1)
            options.append(3)
        else:
            options.append(0)
            options.append(2)
        next_num = choice(options)
        chain.append(next_num)
    return chain


# converting to freqency functions
```

```
47
48  def convert_to_freqs(chain, two_primes): # two primes is a list
    with 2 values
49      # chain is 8 integers, converting to 9 frequencies
50      initial = 100
51      freqs = [initial]
52      up_y = two_primes[0] # 1
53      down_y = 1 / two_primes[0] # 3
54      up_x = two_primes[1] # 0
55      down_x = 1 / two_primes[1] # 2
56
57      # get values in Hz
58      for index, number in enumerate(chain):
59          if number == 0:
60              # up_x
61              new_freq = freqs[index] * up_x
62              freqs.append(new_freq)
63
64          if number == 1:
65              # up_y
66              new_freq = freqs[index] * up_y
67              freqs.append(new_freq)
68
69          if number == 2:
70              # down_x
71              new_freq = freqs[index] * down_x
72              freqs.append(new_freq)
73
74          if number == 3:
75              # down_y
76              new_freq = freqs[index] * down_y
77              freqs.append(new_freq)
78
79      #print(freqs)
80      # now normalize
81      normalized = normalize_data(freqs)
82      #print(normalized)
83      # now put in right range
84      scaled = scale_freqs(normalized)
85      #print(scaled)
86      return scaled
87
88  def normalize_data(data_list):
89      #minimum = min(data_list)
90      maximum = max(data_list)
91      #print(minimum, maximum)
```

```
    new_list = []
    for i in data_list:
        scaled_datum = i / maximum
        new_list.append(scaled_datum)
    return new_list

def scale_freqs(normalized_data):
    # data all between 0 and 1
    # returns freqs in Hz between 100 and ___
    scale_factor = 4096 # power of 2
    new_list = []
    for i in normalized_data:
        scaled_datum = i * scale_factor
        new_list.append(scaled_datum)
    return new_list

#######
"""
1 <-- 2 <-- 3 <-- 4 --> 5 --> 6 --> 7 --> 8
"""

CENTER_FREQ = 440

interval = Fraction(7/4)    # 3/2 5/4 7/4
u_interval = 1 / interval

def get_user_chain():
    user_entry = input('Enter 7 digit ternary sequence (no spaces):
    ')
    chain = []
    for i in user_entry:
        try:
            i = int(i)
        except ValueError:
            print("Please enter only 0's, 1's, or 2's.")
            break
            get_user_chain()
        if i < 0 or i > 2:
            print("Please enter only 0's, 1's, or 2's.")
            break
            get_user_chain()
        else:
            chain.append(i)
    return chain

def freqs_going_down(chain, interval, u_interval, center_freq):
    # Walls 3, 2, 1 (from Wall 4) returns freqs [wall1, wall2
```

```
    # Walls 3, 2, 1 (from Wall 4) returns freqs [wall1, wall2,
    wall3, wall4]
    walls_1234 = []
    current_freq = center_freq
    walls_1234.insert(0, current_freq)
    for direction in chain[2::-1]: # get first three values
        if direction == 2:
            current_freq = float(current_freq * u_interval)
        elif direction == 1:
            pass
        elif direction == 0:
            current_freq = float(current_freq * interval)
        walls_1234.insert(0, current_freq)
    return walls_1234

def freqs_going_up(chain, interval, u_interval, center_freq):
    # Walls 5, 6, 7, 8 (from Wall 4) returns freqs [wall5, wall6,
    wall7, wall8]
    walls_5678 = []
    current_freq = center_freq
    for direction in chain[3:]:
        if direction == 2:
            current_freq = float(current_freq * interval)
        elif direction == 1:
            pass
        elif direction == 0:
            current_freq = float(current_freq * u_interval)
        walls_5678.append(current_freq)
    return walls_5678

"""

def make_ternary_chain(chain_length):
    chain = []
    for number in range(chain_length):
        chain.append(randint(0, 2))
    return chain


while True:
    get_interval = input("Interval ratio (x/y, 'q' to quit): ")
    if get_interval == 'q':
        break
    elif not get_interval:
        pass
    else:
        try:
```

```
            try:
                interval = Fraction(get_interval)
                u_interval = 1 / interval
            except ValueError:
                print("Invalid entry.")
                continue
        chain = get_user_chain()
        walls_1234 = freqs_going_down(chain, interval, u_interval,
        CENTER_FREQ)
        walls_5678 = freqs_going_up(chain, interval, u_interval,
        CENTER_FREQ)
        walls_12345678 = walls_1234 + walls_5678
        print(walls_12345678)
        client.send_message("/chain", walls_12345678)
    """

```

```
"""
oscDistance.py
Eric Heep, John Eagle

Waits for an OSC message from a host, then sends back a distance
measurement to the host.
"""

# standard imports
import socket, argparse, random, time
from datetime import datetime, timedelta

# pi import
import RPi.GPIO as GPIO

# osc imports
from pythonosc import dispatcher, osc_server, osc_message_builder,
udp_client

TRIG = 23
ECHO = 24


def ultrasonic_init():
    GPIO.setmode(GPIO.BCM)

    GPIO.setup(TRIG, GPIO.OUT)
    GPIO.setup(ECHO, GPIO.IN)

    GPIO.output(TRIG, False)
    time.sleep(2)


def get_ip():
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        # doesn't even have to be reachable
        s.connect(('192.255.255.255', 1))
        IP = s.getsockname()[0]
    except:
        IP = '127.0.0.1'
    finally:
        s.close()
        return IP


```

```
def get_reading():
    # does a ping or something
    GPIO.output(TRIG, True)
    time.sleep(0.00001)
    GPIO.output(TRIG, False)

    break_loop = False
    timeout = datetime.now() + timedelta(seconds=1)

    pulse_end = 0
    pulse_start = 0

    # finds the time measurements?
    while GPIO.input(ECHO)==0:
        pulse_start = time.time()
        # if timeout > datetime.now():
        #    break;

    timeout = datetime.now() + timedelta(seconds=1)
    while GPIO.input(ECHO)==1:
        pulse_end = time.time()
        # if timeout > datetime.now():
        #     break;

    # some calculations to convert to centimeters
    pulse_duration = pulse_end - pulse_start
    distance = pulse_duration * 17150
    return round(distance, 2)


def send(self, junk):
    packet = osc_message_builder.OscMessageBuilder(address=piWall)

    # adds whichPi to the OSC message
    hostname = socket.gethostname()

    # we send the Pi's IP address as the OSC address
    # so the host computer knows which Pi sent a message
    packet.add_arg(hostname, arg_type='s')

    # adds distance reading to the OSC message
    packet.add_arg(get_reading(), arg_type='f')

    # completes the OSC message
    packet = packet.build()
```

```
    # sends distance back to the host
    client.send(packet)


if __name__ == "__main__":
    # osc vars
    piWall = "/w"
    hostIp = "192.168.0.7"
    piPort = 5000
    hostPort = 12345

    piIp = get_ip()
    ultrasonic_init()

    # sets up arguments for the dispatcher
    parser = argparse.ArgumentParser()
    parser.add_argument("--hostIp",
                        type=str, default=hostIp, help="The IP
                        address to send back to")
    parser.add_argument("--hostPort",
                        type=int, default=hostPort, help="The port
                        to send back to")
    args = parser.parse_args()

    # this IP is set to send out
    client = udp_client.UDPClient(args.hostIp, args.hostPort)

    # the thread that listens for the OSC messages
    dispatcher = dispatcher.Dispatcher()
    dispatcher.map(piWall, send)

    # the server we're listening on
    server = osc_server.ThreadingOSCUDPServer(
        (piIp, piPort), dispatcher)

    print("Serving on " + piIp + " (" + socket.gethostname() + ")"
    +  " on port " + str(piPort))
    print("Sending back to " + args.hostIp + " to port " +
    str(args.hostPort))

    # here we go!
    server.serve_forever()

```

```
// local_receiver.ck
// John Eagle

// osc stuff
OscIn in;
OscMsg msg;

10002 => in.port;
in.listenAll();

// sine tones
SinOsc sin => dac;
sin.gain(0.0);
sin.freq(1.0);

Step st => Gain stGain => dac;

// because of distortion
dac.gain(0.5);

0.001 => float gainInc;
0.0 => float targetGain;

fun void easeGain() {
    while (true) {
        if (sin.gain() < targetGain - gainInc) {
            sin.gain() + gainInc => sin.gain;
        }
        else if (sin.gain() > targetGain + gainInc) {
            sin.gain() - gainInc => sin.gain;
        }
        1::ms => now;
    }
}

spork ~ easeGain();

// constant
512 => int bufferSize;

// loop it
while (true) {
    in => now;
    while (in.recv(msg)) {
        // frequency of the sine tone
        if (msg.address == "/sineFreq") {
```

```
47              msg.getFloat(0) => sin.freq;
48              <<< "/sineFreq", sin.freq() >>>;
49          }
50          // gain of the sine tone
51          if (msg.address == "/sineGain") {
52              msg.getFloat(0) => targetGain;
53              <<< "/sineGain", targetGain >>>;
54          }
55              stGain.gain(0.0);
56
57          }
58      1::samp => now;
59  }
60
```

```
// receiver.ck
// Eric Heep, John Eagle

// osc stuff
OscIn in;
OscMsg msg;

[
 me.dir(-1) + "samples/fake-bricks/fake-brick-1.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-2.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-3.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-4.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-5.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-6.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-7.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-8.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-9.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-10.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-11.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-12.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-13.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-14.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-15.wav",
 me.dir(-1) + "samples/fake-bricks/fake-brick-16.wav"
] @=> string brickSamplePaths[];

[
 me.dir(-1) + "samples/fake-field/fake-field-1.wav",
 me.dir(-1) + "samples/fake-field/fake-field-2.wav"
] @=> string fieldRecordingPaths[];

brickSamplePaths.size() => int numBrickSamples;
fieldRecordingPaths.size() => int numFieldRecordings;

SndBuf brickSamples[numBrickSamples];
SndBuf fieldRecordings[numFieldRecordings];

for (0 => int i; i < numBrickSamples; i++) {
    brickSamples[i] => dac;
    brickSamples[i].read(brickSamplePaths[i]);
    brickSamples[i].pos(brickSamples[i].samples());
}

for (0 => int i; i < numFieldRecordings; i++) {
    fieldRecordings[i] => dac;
    fieldRecordings[i].read(fieldRecordingPaths[i]);
```

```
    fieldRecordings[i].pos(fieldRecordings[i].samples());
}

10001 => in.port;
in.listenAll();

// sine tones
SinOsc sin => dac;
sin.gain(0.0);
sin.freq(1.0);

Step st => Gain stGain => dac;

// because of distortion
dac.gain(0.5);

0.001 => float gainInc;
0.0 => float targetGain;

fun void easeGain() {
    while (true) {
        if (sin.gain() < targetGain - gainInc) {
      sin.gain() + gainInc => sin.gain;
        }
        else if (sin.gain() > targetGain + gainInc) {
            sin.gain() - gainInc => sin.gain;
        }
        1::ms => now;
    }
}

spork ~ easeGain();

// constant
512 => int bufferSize;

// loop it
while (true) {
    in => now;
    while (in.recv(msg)) {
        // frequency of the sine tone
        if (msg.address == "/sineFreq") {
            msg.getFloat(0) => sin.freq;
            <<< "/sineFreq", sin.freq() >>>;
        }
        // gain of the sine tone
```

```
93          if (msg.address == "/sineGain") {
94              msg.getFloat(0) => targetGain;
95              <<< "/sineGain", targetGain >>>;
96          }
97          // receive packet of audio samples
98          if (msg.address == "/m") {
99              <<< "received sound", "" >>>;
100             stGain.gain(1.0);
101             // start the sample playback
102             for (0 => int i; i < bufferSize; i++) {
103                 msg.getFloat(i) => st.next;
104                 1::samp => now;
105             }
106
107             stGain.gain(0.0);
108
109         }
110
111         if (msg.address == "/bufferSize") {
112             msg.getInt(0) => bufferSize;
113             <<< "Buffer size set to", bufferSize, "" >>>;
114         }
115
116         if (msg.address == "/brickPlay") {
117             msg.getInt(0) => int idx;
118             if (idx > 0 && idx <= numBrickSamples) {
119                 brickSamples[idx - 1].pos(0);
120             }
121         }
122
123         if (msg.address == "/fieldPlay") {
124             msg.getInt(0) => int idx;
125             if (idx > 0 && idx <= numFieldRecordings) {
126                 fieldRecordings[idx - 1].pos(0);
127             }
128         }
129
130         if (msg.address == "/fieldStop") {
131             msg.getInt(0) => int idx;
132             if (idx > 0 && idx <= numFieldRecordings) {
133                 fieldRecordings[idx - 1].pos(fieldRecordings[idx -
  •              1].samples());
134             }
135         }
136     }
137     1::samp => now;
138 }
```

```
138   }
139
```