Final Report

AI Bandmate

신재민

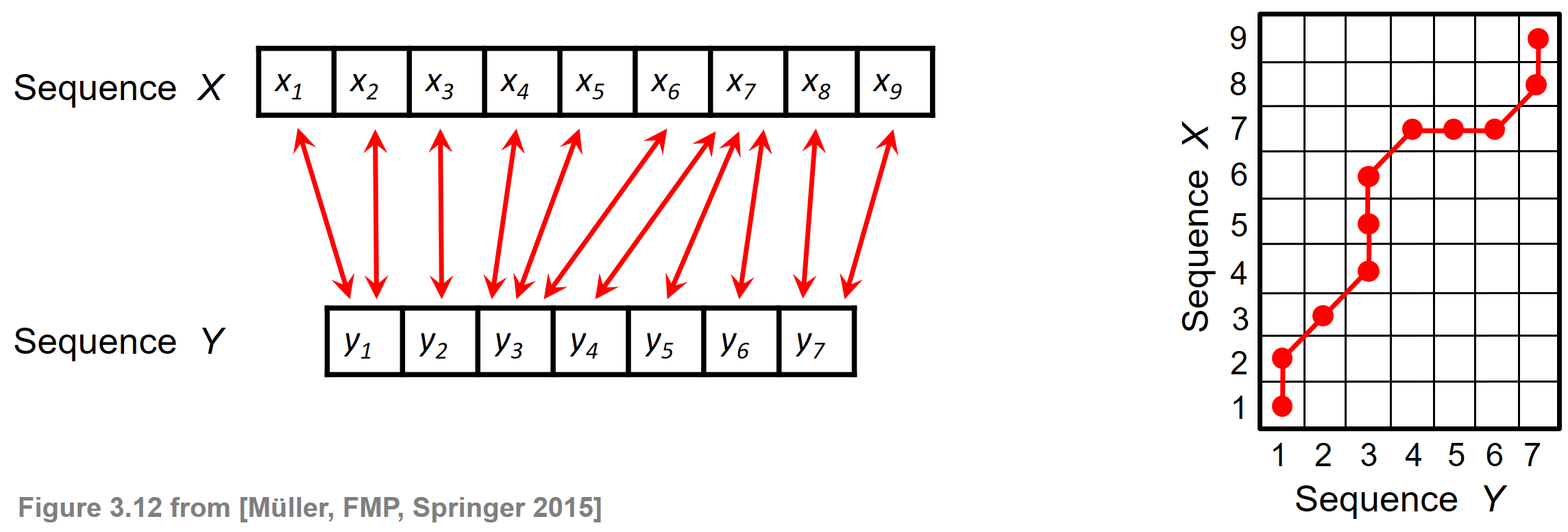
1. 서론

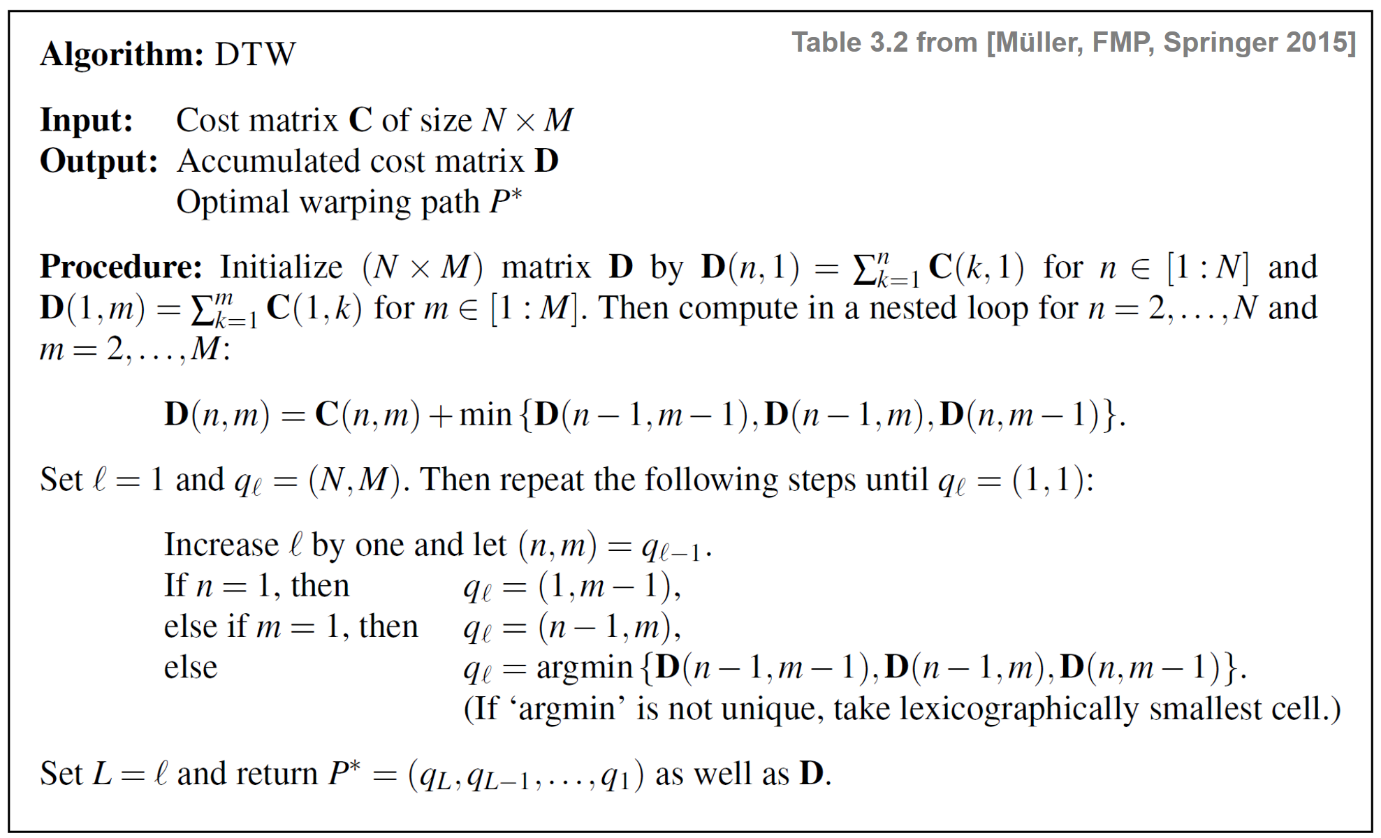
코로나 판데믹으로 인해 서로간의 소통이 단절된 시간이 지속되는 중이다. 대부분이 비대면으로 진행되는 가운데 많은 음악인들은 합주 및 연습에 큰 불편함을 느끼고 있다. 이에 맞추어 YAMAHA 사에서 나온 싱크룸(シンクルーム / SYNCROOM)[[1]](#endnote-1)과 같은 프로그램이 있지만 별도의 오디오 인터페이스가 없다면 레이턴시로 인해 정상적인 작동을 하지 못하는 등 제약사항이 많다. 이러한 상황에 한계를 느꼈기에 우리는 합주에서 쓸 수 있게끔 실시간으로 연주하는 박자에 맞추어 다른 세션을 연주하는 프로그램을 제작하기로 하였다.

2. 배경지식

우리가 제작할 프로그램에 가장 중요한 기준점은 실시간 처리(real time)가 가능하고, 박자를 정확하게 처리해야 한다는 것이었다. 여러가지 방법론이 있었지만 우리가 최종적으로 선택한 방법은 On-Line Time Warping(on-line dynamic time warping, OLTW 이하 oDTW)이라는 알고리즘이었다. 이를 이해하기 위해서는 원형인 Dynamic Time Warping(이하 DTW) 알고리즘의 이해가 필요하다.

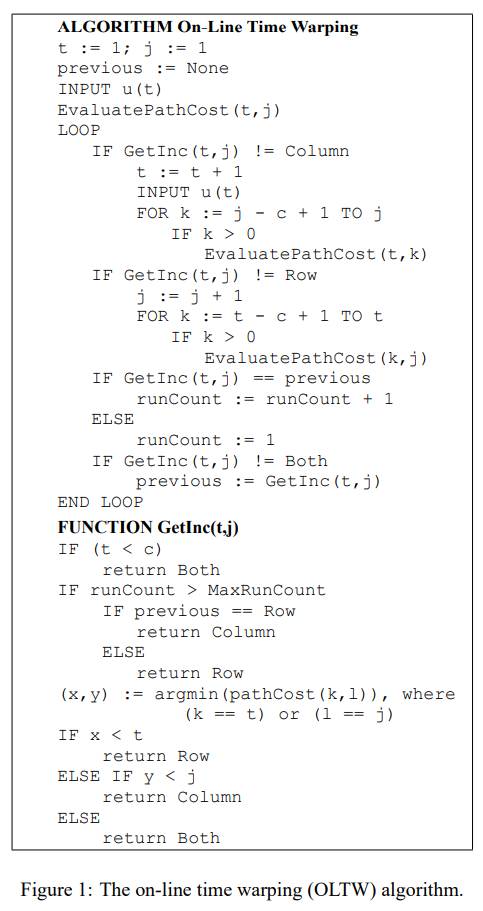
DTW는 주어진 두개의 sequence 사이에서 최적의 warping path를 찾아내는 알고리즘이다.[[2]](#endnote-2) 이를 음악 데이터에 적용하게 되면 서로의 piecewise한 빠르기가 다르더라도 서로 동일한 내용을 담고 있는 부분을 matching할 수 있다. 아래 그림들은 해당 알고리즘의 예시와 pseudocode를 보여준다.



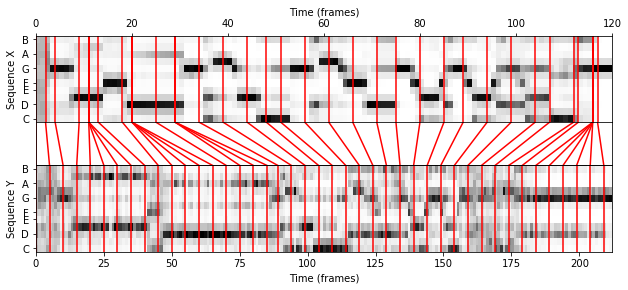


DTW 알고리즘은 실제 여러 상용 음악 편집 프로그램에서 사용되고 있는 성능이 검증된 알고리즘이지만 치명적인 제약조건이 있다. 바로 서로 비교할 두 sequence는 finite하고 그 시작과 끝이 미리 알고 있어야 된다는 점이다. 이는 실시간으로 데이터가 들어오는 우리의 조건에는 쓸 수 없는 알고리즘이다. 그래서 이를 해결하기 위한 개선 알고리즘이 oDTW이다.

oDTW는 Simon Dixon에 의해 소개되어진 알고리즘으로 DTW를 기반으로 새로운 데이터가 계속 들어오면서 mapping을 진행할 수 있게 되었다.[[3]](#endnote-3) 가장 큰 차이점은 바로 직전의 데이터를 기반으로 다음 path를 결정하기에 마지막 지점에서 path 계산을 시작하는 DTW와 달리 첫 지점부터 순서대로 path 계산이 시작된다는 것이다. 또한 끝을 정확하게 알지 못한다는 조건으로 인해 정확도가 떨어진다는 단점이 있다. 최악의 경우는 warping이 실패하여 column 또는 row 한 방향만으로만 계속해서 path를 이어나갈 수 도 있다는 것이다. 이를 보완하기 위해 미리 한계치를 정해놓고 한 방향으로만 계속 진행하면 강제로 방향을 바꾸어주는 방식을 사용한다. 해당 알고리즘의 pseudocode는 다음과 같다.

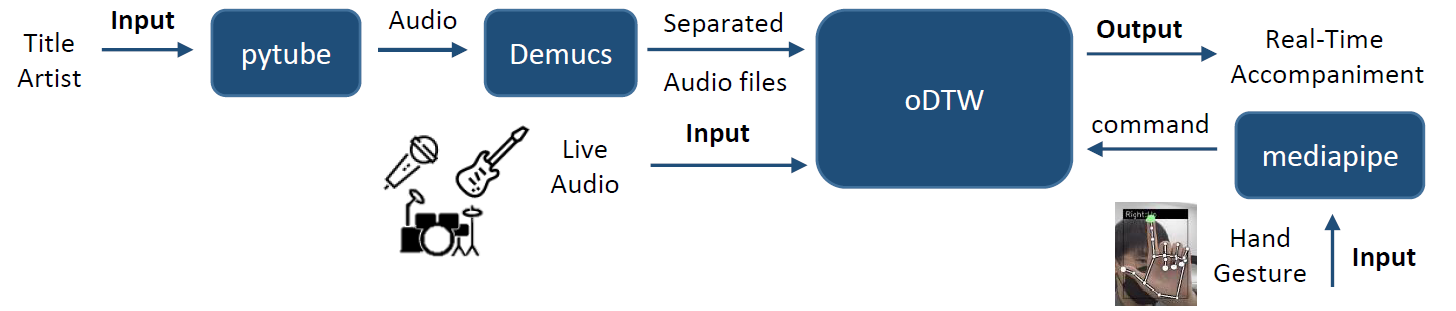


앞서 소개한 DTW, oDTW 모두 sequence 형태의 데이터가 필요한데 우리는 chroma-based feature를 사용하여 비교하기로 하였다. Chroma-based feature란 특정 구간에 각 주파수가 어느 정도의 비율로 분포하고 있는지를 나타내는 데이터로써, 일반적으로는 음악에서 사용하는 12 음계를 기반으로 분류한다. 이를 얻기 위해서는 음악 데이터를 일정 구간으로 쪼개어 이산 푸리에 변환을 통해 주파수 성분을 얻어 낸 뒤, 이를 12개의 음계를 나타내는 bin에 정리를 하게 된다. 해당 feature의 이점은 빠르기를 다르게 연주하더라도 주파수는 변하지 않기에 정확한 기준점이 될 수 있다는 것이다. 이를 이용하여 DTW를 시행한 예시는 다음과 같다.



3. 개발 과정

우리가 만든 시스템의 개요는 다음과 같다. 전체 코드는 github을 통해 확인 할 수 있다.[[4]](#endnote-4)



우선 연주할 노래를 무엇인지 입력하게 되면 해당 음원을 찾아서 각 파트 별로 음원을 분리하게 된다. 해당 작업이 완료되면 live로 음악 데이터를 입력하게 되고 기존 음원과 oDTW를 진행하여 현재 곡의 위치를 알아낸 후 선택한 반주의 해당 부분을 출력하게 된다. 또한 연주 도중 필요할 수 있는 입력을 주기 위하여 손 동작을 인식하여 이에 해당하는 command를 수행할 수 있게끔 하였다.

전체적인 구현은 windows 환경에서 python을 이용하여 구현하였으며 사용한 외부 라이브러리들은 다음과 같다.

|  |  |
| --- | --- |
| pytube | 음원을 받아오는데 사용 |
| Demucs | 음원을 분리하는데 사용 |
| pyaudio | 사운드 입출력에 사용 |
| librosa | 사운드 프로세싱에 사용 |
| mediapipe | 손동작 인식에 사용 |

4. oDTW 코드 정리

가장 핵심이 되는 oDTW 알고리즘 구현 코드는 다음과 같다.

|  |
| --- |
| import numpy as np  import librosa  import time  import pyaudio  import math  import keyboard  import motion  from multiprocessing import Process, Queue  def distance(x, y):  total = 0.0  for i in range(12):  total = total + (x[i] - y[i]) \* (x[i] - y[i])  return math.sqrt(total)  if \_\_name\_\_ == '\_\_main\_\_':  q = Queue()  process = Process(target=motion.motion\_main.main,args=(q,))  process.start()  inst, inst\_sr = librosa.load("wav/inst.wav", dtype="float32", sr=None)  # inst\_sr = 48000  INST\_RATE = inst\_sr  INST\_FRAME = int(INST\_RATE / 10)  vocal, vocal\_sr = librosa.load("wav/vocal.wav", dtype="float32", sr=None)  # vocal\_sr = 48000  VOCAL\_RATE = vocal\_sr  VOCAL\_FRAME = int(VOCAL\_RATE / 10)  chroma\_ref = librosa.feature.chroma\_stft(  y=vocal, sr=vocal\_sr, tuning=0, norm=2, hop\_length=VOCAL\_FRAME, center=False  )  INPUT\_RATE = VOCAL\_RATE  INPUT\_FRAME = int(INPUT\_RATE / 10)  input\_half, dummy = librosa.load("wav/vocal\_05x.wav", dtype="float32", sr=None)  input\_one = vocal  input\_one\_half, dummy = librosa.load("wav/vocal\_15x.wav", dtype="float32", sr=None)  input\_twice, dummy = librosa.load("wav/vocal\_2x.wav", dtype="float32", sr=None)  input = input\_twice  print("loaded")  is\_out\_ended = False  VERY\_BIG = 999.0  MAX = 5  WIN\_SIZE = 50  D = np.full((chroma\_ref.shape[1], 3 \* chroma\_ref.shape[1]), VERY\_BIG)  i\_frame = 0  t = h = 0  path\_t = []  path\_h = []  STEP = 8  is\_mic = False  def callback(in\_data, frame\_count, time\_info, flag):  # in\_data : 4 bytes  global i\_frame, t, h, D, path\_t, path\_h, is\_out\_ended  print("call back: ", i\_frame)  # input part  if i\_frame == 0:  if is\_mic == False:  tmp = input[0:frame\_count]  else:  tmp = np.frombuffer(in\_data, np.float32)  chroma = librosa.feature.chroma\_stft(  y=tmp,  sr=INPUT\_RATE,  norm=2,  tuning=0,  hop\_length=frame\_count,  win\_length=frame\_count,  n\_fft=frame\_count,  center=False,  )  D[0][0] = distance(chroma\_ref[:, [0]], chroma)  else:  if is\_mic == False:  tmp = input[i\_frame \* frame\_count : (i\_frame + 1) \* frame\_count]  else:  tmp = np.frombuffer(in\_data, np.float32)  chroma = librosa.feature.chroma\_stft(  y=tmp,  sr=INPUT\_RATE,  norm=2,  tuning=0,  hop\_length=frame\_count,  win\_length=frame\_count,  n\_fft=frame\_count,  center=False,  )  for i in range(max(1, h - WIN\_SIZE), min(D.shape[0], h + WIN\_SIZE)):  dist = distance(chroma\_ref[:, [i]], chroma)  top = D[i][i\_frame - 1]  mid = D[i - 1][i\_frame - 1]  bot = D[i - 1][i\_frame]  if (top < mid) and (top < bot):  D[i][i\_frame] = top + dist \* 0.1  elif mid < bot:  D[i][i\_frame] = mid + dist \* 0.11  else:  D[i][i\_frame] = bot + dist \* 0.1  cnt = 0  while (cnt < MAX) and (t < i\_frame) and (h < D.shape[0] - 1):  up = D[h + 1][t]  diag = D[h + 1][t + 1]  side = D[h][t + 1]  if (up < diag) and (up < side):  h = h + 1  cnt = cnt + 1  elif diag < side:  h = h + 1  t = t + 1  else:  t = t + 1  if cnt == MAX:  t = t + 1  if not q.empty():  if q.get() == 'Up':  print('up')  h += 1  path\_t = np.append(path\_t, t)  path\_h = np.append(path\_h, h)  print(t, h)  # output part  if i\_frame == 0:  if is\_mic == True:  out\_data = inst[:frame\_count]  else:  out\_data = inst[:frame\_count] + input[:frame\_count]  else:  if i\_frame < STEP:  pre = int(path\_h[i\_frame - 1])  now = int(path\_h[i\_frame])  slope = now - pre  else:  pre = int(path\_h[i\_frame - STEP])  now = int(path\_h[i\_frame])  slope = int((now - pre) / STEP)  if slope >= 1:  if (now + 1) \* frame\_count >= inst.shape[0]:  if is\_mic == False:  out\_data = input[i\_frame \* frame\_count : -1]  out\_data = np.pad(  out\_data, frame\_count - len(out\_data), constant\_values=0  )  else:  out\_data = np.zeros(frame\_count, dtype=np.float32)  tmp = inst[now \* frame\_count : -1]  tmp = np.pad(tmp, slope \* frame\_count - len(tmp), constant\_values=0)  is\_out\_ended = True  next\_status = pyaudio.paComplete  else:  if is\_mic == False:  out\_data = input[  i\_frame \* frame\_count : (i\_frame + 1) \* frame\_count  ]  else:  out\_data = np.zeros(frame\_count, dtype=np.float32)  tmp = inst[now \* frame\_count : (now + slope) \* frame\_count]  out\_data = out\_data + librosa.effects.time\_stretch(tmp, rate=slope)  elif slope == 0:  if is\_mic == False:  out\_data = input[i\_frame \* frame\_count : (i\_frame + 1) \* frame\_count]  else:  out\_data = np.zeros(frame\_count, dtype=np.float32)  i\_frame = i\_frame + 1  if is\_out\_ended:  next\_status = pyaudio.paComplete  else:  next\_status = pyaudio.paContinue  return (out\_data, next\_status)  p = pyaudio.PyAudio()  FORMAT = pyaudio.paFloat32  CHANNELS = 1  stream = p.open(  format=FORMAT,  channels=CHANNELS,  rate=INPUT\_RATE,  input=True,  output=True,  frames\_per\_buffer=INPUT\_FRAME,  stream\_callback=callback,  )  stream.stop\_stream()  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  q.get()  stream.start\_stream()  while True:  if keyboard.is\_pressed('q'):  stream.stop\_stream()  break  elif keyboard.is\_pressed('r'):  print("reset")  stream.stop\_stream()  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('1'):  print("0.5x")  stream.stop\_stream()  is\_mic = False  input = input\_half  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('2'):  print("1x")  stream.stop\_stream()  is\_mic = False  input = input\_one  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('3'):  print("1.5x")  stream.stop\_stream()  is\_mic = False  input = input\_one\_half  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('4'):  print("2x")  stream.stop\_stream()  is\_mic = False  input = input\_twice  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('5'):  print("mic")  stream.stop\_stream()  is\_mic = True  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  if not q.empty():  print(q.get())  print("quit")  stream.close()  p.terminate()  process.terminate() |

이하 내용은 각 부분의 자세한 설명이다.

|  |
| --- |
| def distance(x, y):  total = 0.0  for i in range(12):  total = total + (x[i] - y[i]) \* (x[i] - y[i])  return math.sqrt(total) |

두 개의 chroma feature가 있을 때, 그 둘의 유사도를 Euclidian distance로 계산하는 함수이다. 하나의 chroma feature는 12개의 bin이 있기에 12번 loop를 돌리게 된다.

|  |
| --- |
| if \_\_name\_\_ == '\_\_main\_\_':  q = Queue()  process = Process(target=motion.motion\_main.main,args=(q,))  process.start() |

사운드 프로세스와 동시에 손 동작 인식 프로세스를 돌려야 하기에 multiprocessing 모듈을 이용하여 두 개의 프로세스가 돌아가게 해준다.

|  |
| --- |
| inst, inst\_sr = librosa.load("wav/inst.wav", dtype="float32", sr=None)  # inst\_sr = 48000  INST\_RATE = inst\_sr  INST\_FRAME = int(INST\_RATE / 10)  vocal, vocal\_sr = librosa.load("wav/vocal.wav", dtype="float32", sr=None)  # vocal\_sr = 48000  VOCAL\_RATE = vocal\_sr  VOCAL\_FRAME = int(VOCAL\_RATE / 10)  chroma\_ref = librosa.feature.chroma\_stft(  y=vocal, sr=vocal\_sr, tuning=0, norm=2, hop\_length=VOCAL\_FRAME, center=False  )  INPUT\_RATE = VOCAL\_RATE  INPUT\_FRAME = int(INPUT\_RATE / 10)  input\_half, dummy = librosa.load("wav/vocal\_05x.wav", dtype="float32", sr=None)  input\_one = vocal  input\_one\_half, dummy = librosa.load("wav/vocal\_15x.wav", dtype="float32", sr=None)  input\_twice, dummy = librosa.load("wav/vocal\_2x.wav", dtype="float32", sr=None)  input = input\_twice |

반주로 재생하게 될 음원을 inst로, live input과 비교를 하게 될 음원을 vocal로 로드를 하고 chroma feature로 변경을 하였다. live input으로 사용할 수 있게끔 0.5배속, 1배속, 1.5배속, 2배속으로 미리 가공한 음원을 로드 하였다.

|  |
| --- |
| is\_out\_ended = False  VERY\_BIG = 999.0  MAX = 5  WIN\_SIZE = 50  D = np.full((chroma\_ref.shape[1], 3 \* chroma\_ref.shape[1]), VERY\_BIG)  i\_frame = 0  t = h = 0  path\_t = []  path\_h = []  STEP = 8  is\_mic = False |

oDTW 계산에 사용할 변수들과 matrix를 초기화 해주었다.

|  |
| --- |
| def callback(in\_data, frame\_count, time\_info, flag):  # in\_data : 4 bytes  global i\_frame, t, h, D, path\_t, path\_h, is\_out\_ended  print("call back: ", i\_frame)  # input part  if i\_frame == 0:  if is\_mic == False:  tmp = input[0:frame\_count]  else:  tmp = np.frombuffer(in\_data, np.float32)  chroma = librosa.feature.chroma\_stft(  y=tmp,  sr=INPUT\_RATE,  norm=2,  tuning=0,  hop\_length=frame\_count,  win\_length=frame\_count,  n\_fft=frame\_count,  center=False,  )  D[0][0] = distance(chroma\_ref[:, [0]], chroma) |

실시간으로 사운드를 계속해서 받아오기 위해선 callback 방식으로 한 프레임에 4바이트의 데이터를 받아와 처리하였다. 마이크 입력, 선 녹음 파일 두가지 입력 방법에 따라 데이터를 고를 수 있게 하였고, 제일 첫 프레임인 경우를 계산하였다.

|  |
| --- |
| else:  if is\_mic == False:  tmp = input[i\_frame \* frame\_count : (i\_frame + 1) \* frame\_count]  else:  tmp = np.frombuffer(in\_data, np.float32)  chroma = librosa.feature.chroma\_stft(  y=tmp,  sr=INPUT\_RATE,  norm=2,  tuning=0,  hop\_length=frame\_count,  win\_length=frame\_count,  n\_fft=frame\_count,  center=False,  )  for i in range(max(1, h - WIN\_SIZE), min(D.shape[0], h + WIN\_SIZE)):  dist = distance(chroma\_ref[:, [i]], chroma)  top = D[i][i\_frame - 1]  mid = D[i - 1][i\_frame - 1]  bot = D[i - 1][i\_frame]  if (top < mid) and (top < bot):  D[i][i\_frame] = top + dist \* 0.1  elif mid < bot:  D[i][i\_frame] = mid + dist \* 0.11  else:  D[i][i\_frame] = bot + dist \* 0.1  cnt = 0  while (cnt < MAX) and (t < i\_frame) and (h < D.shape[0] - 1):  up = D[h + 1][t]  diag = D[h + 1][t + 1]  side = D[h][t + 1]  if (up < diag) and (up < side):  h = h + 1  cnt = cnt + 1  elif diag < side:  h = h + 1  t = t + 1  else:  t = t + 1  if cnt == MAX:  t = t + 1  if not q.empty():  if q.get() == 'Up':  print('up')  h += 1  path\_t = np.append(path\_t, t)  path\_h = np.append(path\_h, h) |

t는 live input의 위치를 h는 비교 음원의 위치를 나타내며, 첫 프레임 이후 oDTW 알고리즘 대로 최적의 path를 계산하는 과정을 거치게 된다. 여기서 path를 계산할 때 시간 복잡도를 줄여주기 위하여 미리 정한 window size만큼 계산을 하게 하였다. 또한, 만약 싱크가 맞지 않을 경우를 대비해 강제로 빠르게 재생시키는 수신호를 지정하고 이 때에는 h의 위치를 더 증가시키게 하였다.

|  |
| --- |
| # output part  if i\_frame == 0:  if is\_mic == True:  out\_data = inst[:frame\_count]  else:  out\_data = inst[:frame\_count] + input[:frame\_count]  else:  if i\_frame < STEP:  pre = int(path\_h[i\_frame - 1])  now = int(path\_h[i\_frame])  slope = now - pre  else:  pre = int(path\_h[i\_frame - STEP])  now = int(path\_h[i\_frame])  slope = int((now - pre) / STEP)  if slope >= 1:  if (now + 1) \* frame\_count >= inst.shape[0]:  if is\_mic == False:  out\_data = input[i\_frame \* frame\_count : -1]  out\_data = np.pad(  out\_data, frame\_count - len(out\_data), constant\_values=0  )  else:  out\_data = np.zeros(frame\_count, dtype=np.float32)  tmp = inst[now \* frame\_count : -1]  tmp = np.pad(tmp, slope \* frame\_count - len(tmp), constant\_values=0)  is\_out\_ended = True  next\_status = pyaudio.paComplete  else:  if is\_mic == False:  out\_data = input[  i\_frame \* frame\_count : (i\_frame + 1) \* frame\_count  ]  else:  out\_data = np.zeros(frame\_count, dtype=np.float32)  tmp = inst[now \* frame\_count : (now + slope) \* frame\_count]  out\_data = out\_data + librosa.effects.time\_stretch(tmp, rate=slope)  elif slope == 0:  if is\_mic == False:  out\_data = input[i\_frame \* frame\_count : (i\_frame + 1) \* frame\_count]  else:  out\_data = np.zeros(frame\_count, dtype=np.float32) |

앞에서 계산한 path인 (t,h)쌍을 기반으로 현재 연주 속도를 구한 후 이 속도에 맞추어 inst 음원을 늘리고 줄여 재생하게 하였다. 이때, 출력 노이즈를 줄이기 위하여 바로 직전의 쌍이 아니라 특정 STEP 만큼의 데이터까지 참고하여 재생 속도를 보정해 주었다.

|  |
| --- |
| p = pyaudio.PyAudio()  FORMAT = pyaudio.paFloat32  CHANNELS = 1  stream = p.open(  format=FORMAT,  channels=CHANNELS,  rate=INPUT\_RATE,  input=True,  output=True,  frames\_per\_buffer=INPUT\_FRAME,  stream\_callback=callback,  )  stream.stop\_stream()  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  q.get()  stream.start\_stream() |

사운드 입출력을 담당하는 스트림을 열어 앞서 작성한 callback 함수가 계속 호출되게 하였다.

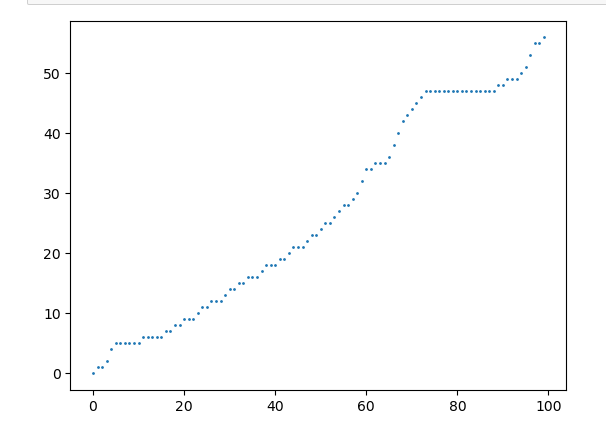
|  |
| --- |
| while True:  if keyboard.is\_pressed('q'):  stream.stop\_stream()  break  elif keyboard.is\_pressed('r'):  print("reset")  stream.stop\_stream()  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('1'):  print("0.5x")  stream.stop\_stream()  is\_mic = False  input = input\_half  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('2'):  print("1x")  stream.stop\_stream()  is\_mic = False  input = input\_one  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('3'):  print("1.5x")  stream.stop\_stream()  is\_mic = False  input = input\_one\_half  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('4'):  print("2x")  stream.stop\_stream()  is\_mic = False  input = input\_twice  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  elif keyboard.is\_pressed('5'):  print("mic")  stream.stop\_stream()  is\_mic = True  i\_frame = 0  t = 0  h = 0  path\_t = []  path\_h = []  stream.start\_stream()  if not q.empty():  print(q.get()) |

키보드를 통해 live input의 모드를 고를 수 있게끔 하였다.

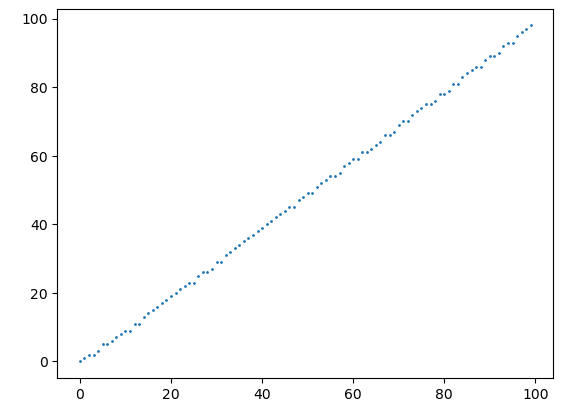
5. 실행 결과 및 분석

0.5배속, 1배속, 2배속의 음원으로 live input을 주었을 때의 path를 plot한 결과는 다음과 같다.

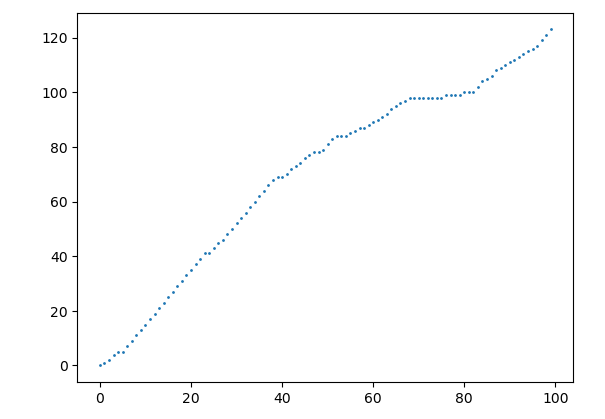
0.5배



1배



2배



1배속은 제일 정확하게 1의 기울기로 매핑되는 것에 비하여, 0.5배속, 2배속은 대략 50 프레임 이후부터 정확도가 급감하는 것을 확인 할 수 있다. 이는 우리가 사용한 oDTW 알고리즘 성능의 한계가 있기에 생긴 오차라 보여진다. 실제로 현재 여러 연구에서 알고리즘의 성능을 개선하기 위한 노력들이 계속되고 있기에 더 개선된 알고리즘을 이용하게 된다면 훨씬 더 깔끔하고 정확한 매핑을 기대할 수 있을 것이다.

알고리즘 외적으로 발견된 문제들은 다음과 같다.

가. 프레임 경계에서의 불연속

1배속을 제외한 모든 배속에서 프레임의 경계에서 노이즈가 생기는 것을 확인하였다. 그 원인을 분석해보니 pitch 변경없이 배속을 변경하게 되면 같은 원본이더라도 구간을 잘라서 배속을 변경하고 이어 붙인 것과 전체 구간을 한 번에 배속을 변경한 것의 결과물이 다르게 나오는 것을 확인하였다. 이는 배속을 변경할 때는 frequency domain에서 처리를 하게 되는데 푸리에 변환의 특성상 프레임 구간의 길이를 다르게 하면 결과물이 다르게 나오기 때문이다.

나. 1배속 미만 재생의 불확실성

우리가 사용하는 모든 데이터들은 digitize 되어 양자화 되어있기에 근본적으로 가질 수 있는 slope의 값이 한정되어 있다. 그로 인해 기울기가 계속 0인 구간이 진짜 연주가 되고 있지 않는 중인지, 매우 낮은 값의 기울기를 가진 것인지 구분이 불가능하다. chroma feature로 변환 할 때 window size를 키우고 hop size를 줄이면 기울기의 해상도가 올라가지만, 그만큼 더 많은 연산을 해야하기에 real time processing에서 overhead가 더 커지게 된다.

6. 결론

이번 창의설계3을 통해 live input을 매핑하는 알고리즘인 oDTW를 이용하여 real time sound processing을 구현해 보았다. 이를 통해 python을 이용한 사운드 프로세싱을 직접 다루어 볼 수 있었고, state-of-the-art algorithm을 이용한 처리를 사용해 볼 수 있었다. 다만 real time image, video processing과는 달리 사운드는 추후에 들어올 데이터의 불확실성으로 인해 다양한 문제점이 발생하는 것을 확인할 수 있었다. 특히 output에 노이즈, 결함이 조금이라도 있으면 부자연 스러움을 금방 느낄 수 있고, 미래 sequence의 예측을 해야한다는 근본적인 결함을 현재로써는 해결할 수 없었다.

처음에 예상했던 퀄리티를 내기 위해서는 근본적인 알고리즘의 개선부터 시작하여 통계적인 분석을 통해 미래 입력의 예측과 사운드 변환에서의 불연속 문제까지 매우 어려운 문제들을 해결해야만 한다. 다음 설계에서 이어 하더라도 이들을 모두 해결하기는 힘들 것으로 보이므로 특정 부분만을 개선하는 것으로 절충하거나 다른 주제를 고려해 봐야 할 것으로 생각된다.

1. https://syncroom.yamaha.com/ [↑](#endnote-ref-1)
2. Meinard Müller: Dynamic Time Warping (Chapter 4 of Information Retrieval for Music and Motion). Springer Verlag, ISBN: 3540740473, 2007. [↑](#endnote-ref-2)
3. Simon Dixon: An On-Line Time Warping Algorithm for Tracking Musical Performances. IJCAI 2005: 1727-1728 [↑](#endnote-ref-3)
4. https://github.com/studio-luke/citd3/tree/main/demo [↑](#endnote-ref-4)