Recep Oğuz Araz
30.07.2019

Introduction

A pitch track is a series of time-frequency pairs following a component of a musical recording. For this case, the pitch track is following the melody of a Turkish-Ottoman Makam recording. While extracting the melody of a recording, we use *PredominantMelodyMakam()* object, which can make 2 types of mistakes:

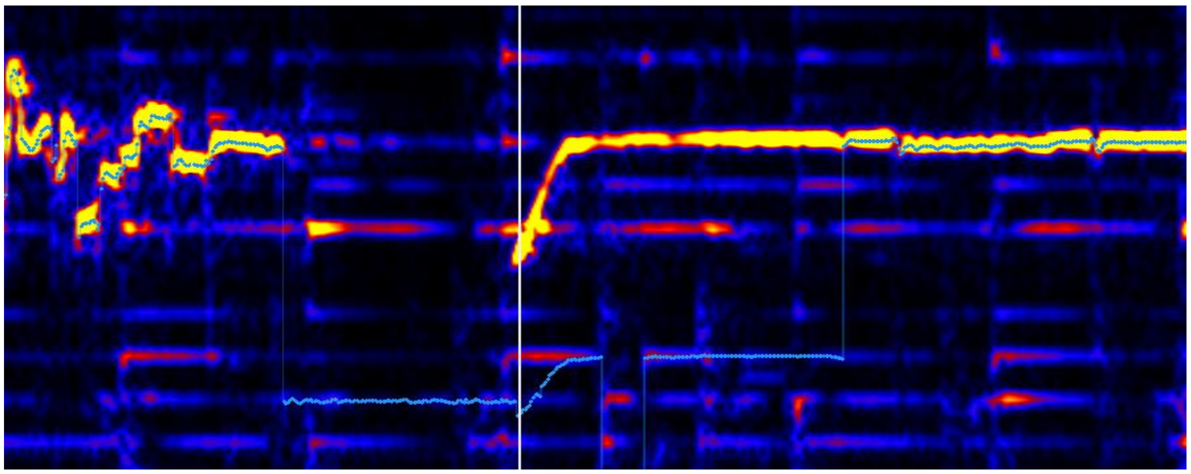
1. Octave errors,
2. Instrument errors.

An octave error is when the algorithm returns an overtone that exists in the melody but doesn't have the correct frequency to define the melody (Further discussion can be useful here in terms of the fundamental frequency).

An instrument error is when the algorithm focuses on an instrument that is creating ornaments but is not the melody, or when the algorithm cannot detect the correct instrument because of its energy level.

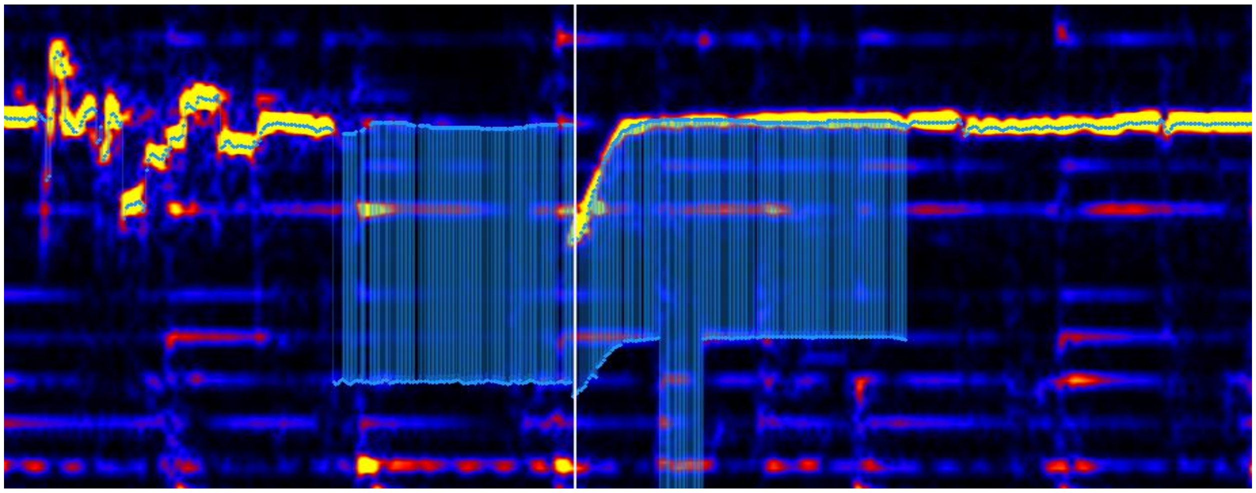
Because we have no algorithm that can perfectly extract the melody of a recording (at least to my knowledge), human verification and correction is required for having accurate pitch tracks. However, with a frame rate of 10msec, correcting a pitch track of a 4min recording manually can be long and demanding.

Problems

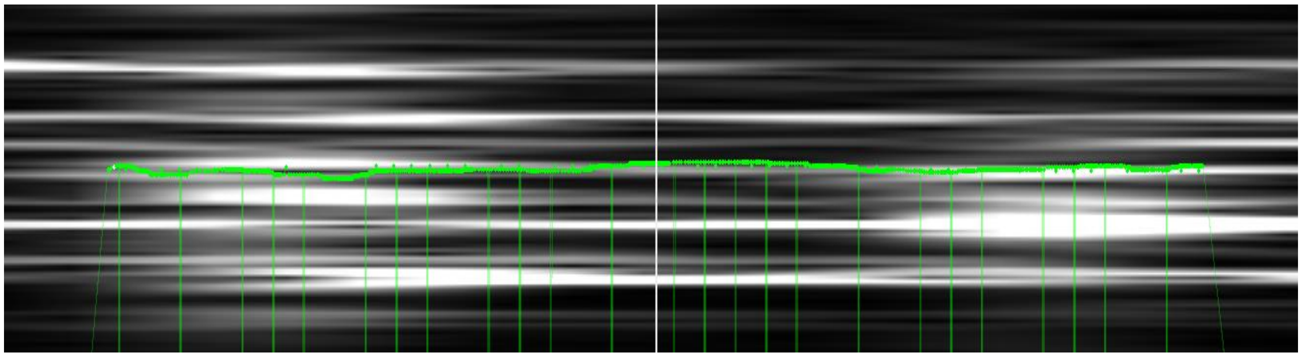


The above figure is an example of an octave error, where the algorithm returns the wrong octave over an interval. After deciding what the correct octave is, the traditional way of correcting it is to zoom really close to the correct frequency range and replicate the false curve by drawing it with the mouse.

Recep Oğuz Araz
30.07.2019



However, drawing the curve manually creates problems which are apparent in the above and below figures.



Here we see an example of an instrument error with the correction curve. The high density of the new points and the non-uniformity is apparent. The traditional way of correcting this track is to zoom on each point that is not included in the correcting curve and pass over it with the mouse. The problem is that for a frame rate of 10msec, to correct a musical phrase of 0.5sec takes a long time.

After spending hours on correcting the pitch track of a single recording, two observations are made:

- While guiding the pitch track using the mouse, the frame rate becomes *non-uniform* and *much denser* than the original pitch track.
- The hard part of guiding is not finding the correct path on the spectrogram but to zoom considerably, until the mouse passes over/under the original sample exactly.

Recep Oğuz Araz
30.07.2019

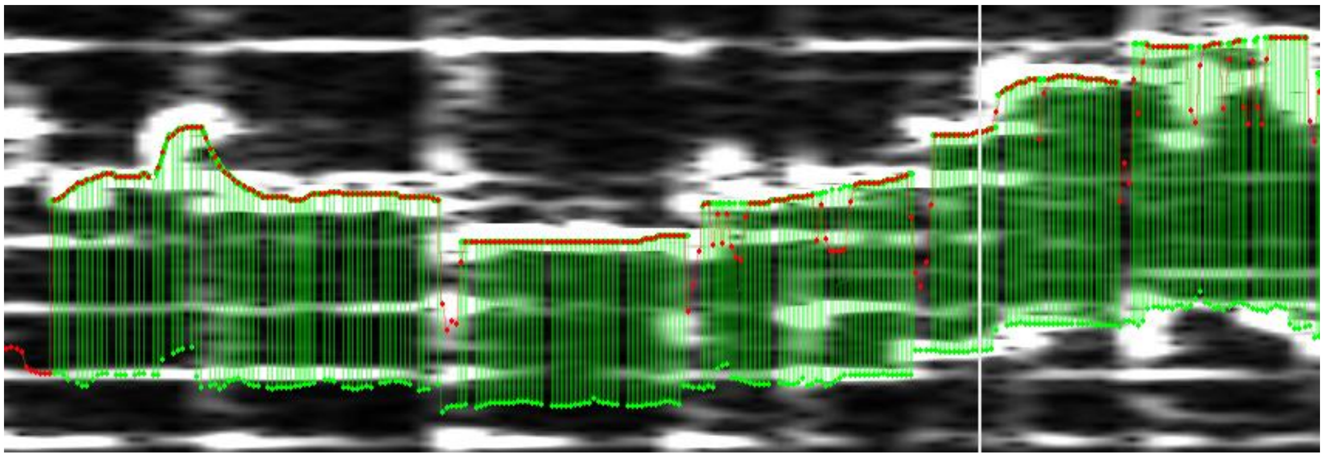
Solution

Half-Automatic Pitch Track Smoother

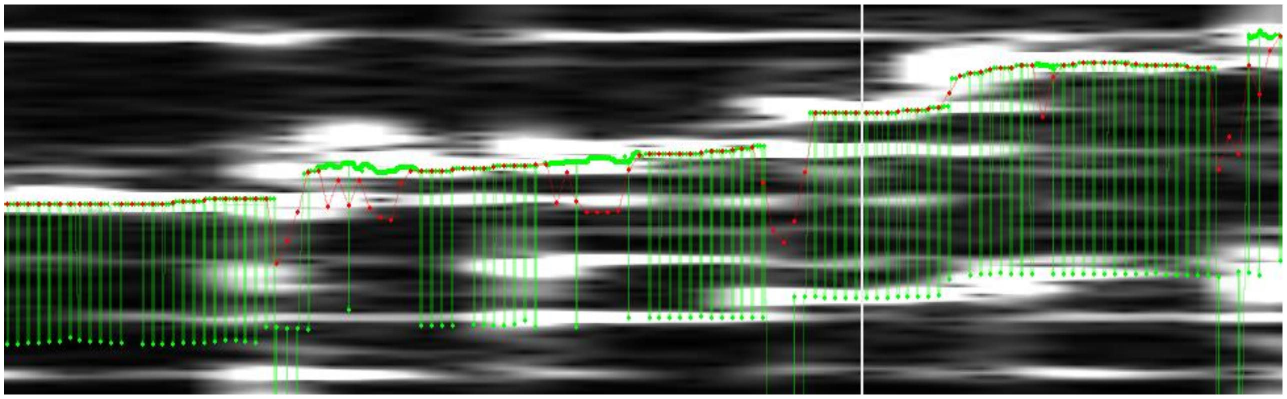
The half automatic pitch-track smoother has three modes:

1. **Full-automatic smoothing** a manually *edited* pitch track and unifying the frame rate without touching the regions that the track is correct
2. **Fixing Octave Errors** by taking time specifications up to 10msecs precision that there is an octave error with the factor of the error.(User interface is on the way)
3. **The fastest way** of smoothing, combination of the first two. The user *do not* edit the octave errors but specify them and correct the *instrument errors* manually and the algorithm takes care of everything.

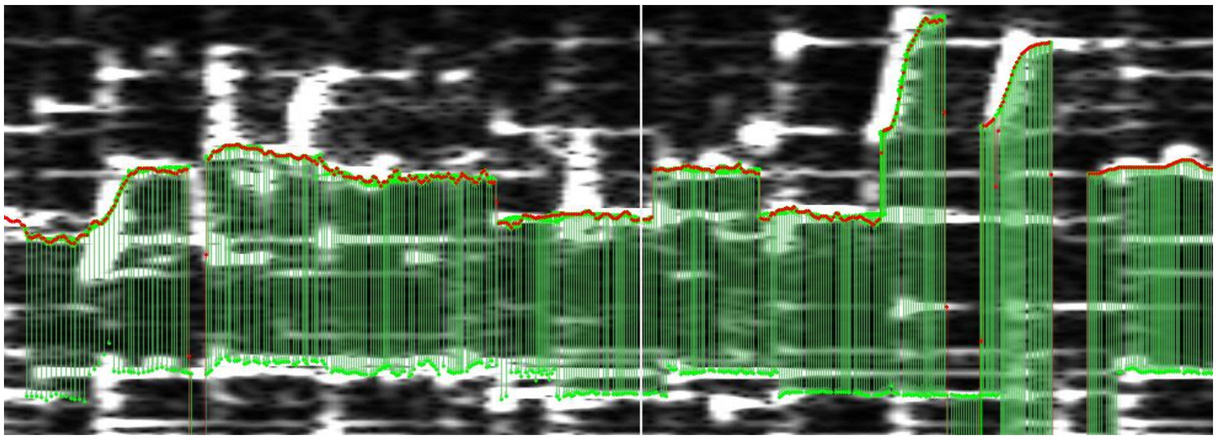
However, while drawing the pitch track the user should make sure that the new curve is dense enough, otherwise, errors can happen. For example,



The green part is the original pitch track together with the correction curve and the red curve is the output of the smoothing algorithm. As the correction curve is not dense enough everywhere, the algorithm is making mistakes. The correct way of drawing the curve is by ensuring density everywhere.

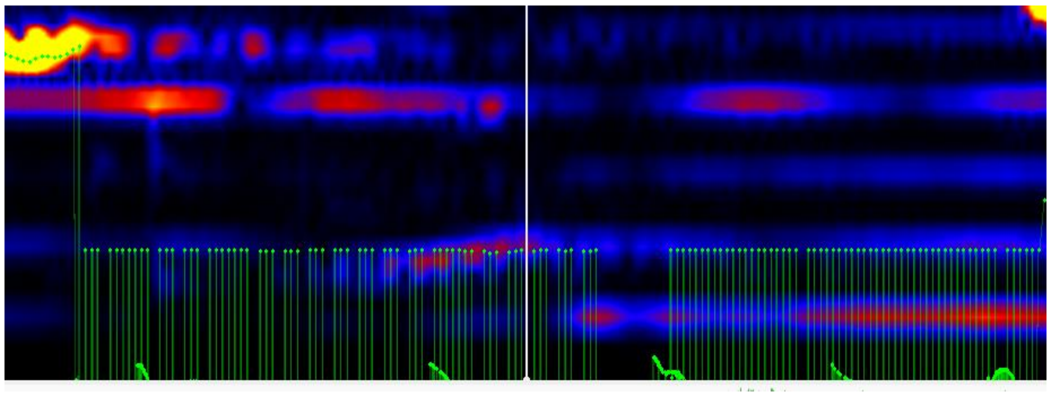


Which results in,



It can be seen that how well the red line, which is the correct pitch track, is reproduced from a dense correction curve.

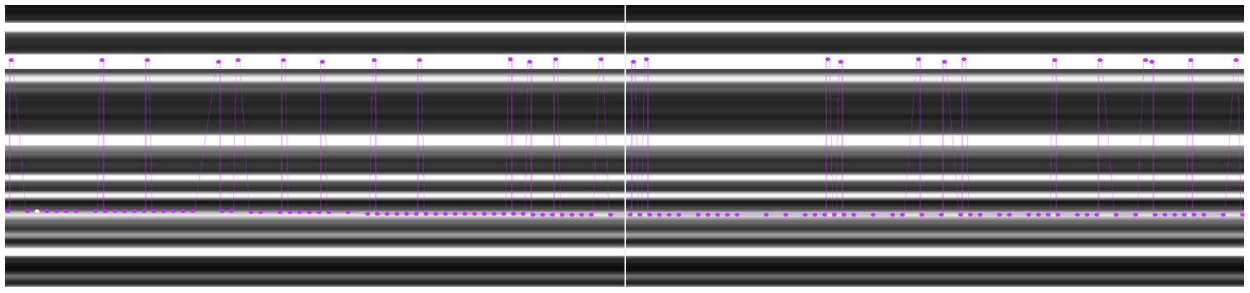
Another application of the algorithm is when the user needs to zero out an entire region. The algorithm takes a threshold from the user and zero outs all the frequencies below that threshold. Therefore, drawing a simple, random but dense curve that is under the threshold will result in the region zeroed out.



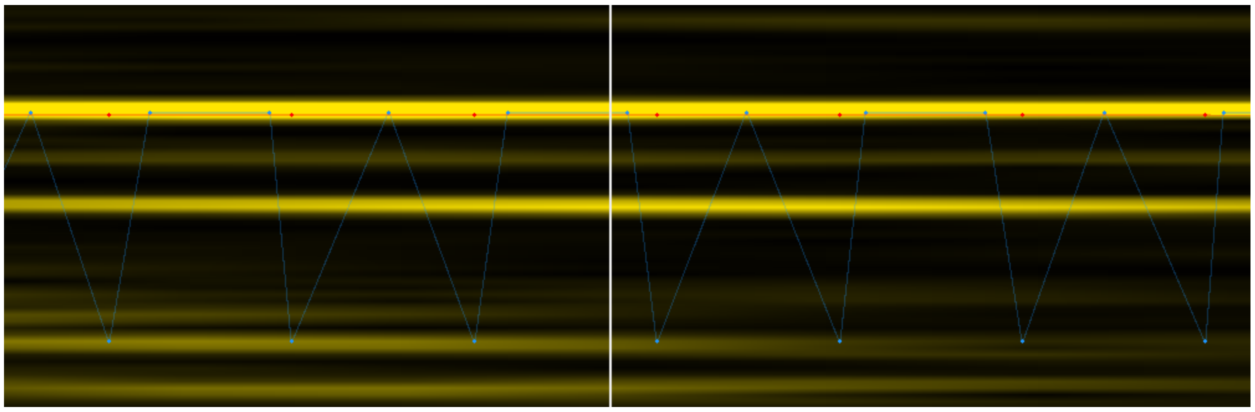
How It Works

A) Smoothing

The algorithm makes use of the observation that, a hand drawn pitch track will be much denser than the computer extracted pitch track and the density of the points are uniform where there is no correction.



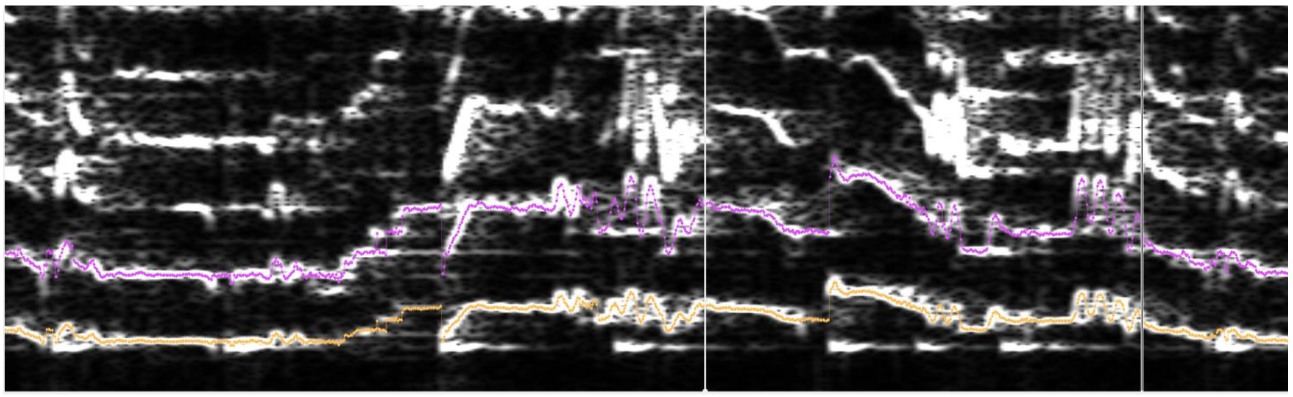
It searches for the points in the correction curve that are integer multiples of the frame rate and makes 2-point averaging around those point. Since the correction curve is dense, we can be sure that there will always be 2 points around the original sample with the correct frequency.



Here we see that the blue curve that is the correction curve is kept at the points that are not integer multiples of the frame rate and are averaged between the gaps, resulting in a smooth pitch track.

B) Octave Correction

Octave Correction mode requires the specification of the time instants where there is an octave error from the user with the factor of the error and depends on the assumption that melodic instruments show harmonicity, i.e. the overtones are integer multiples. (User interface on the way)

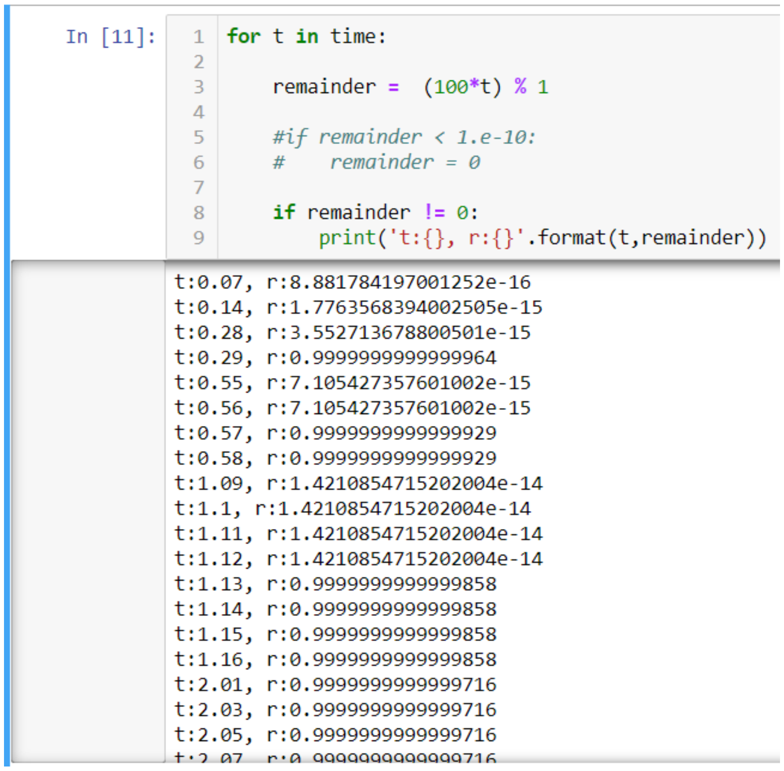


The user can decide which octave is the melody by using the listening mode of the Sonic Visualizer (Some discussion here about the fundamental frequency-melody relationship).

The advantage of using the octave correction mode is that, by multiplying or dividing each point's frequency by a constant, we obtain a precise track that fits exactly in another octave. The validation of this assumption is apparent on the above figure.

Issues

To identify the integer multiples of the frame rate, we have to use floating-point remainder operation. Because designing hardware and creating software that deals with floating point arithmetic requires trade-offs, some simple calculations can yield unintuitive results. For example,

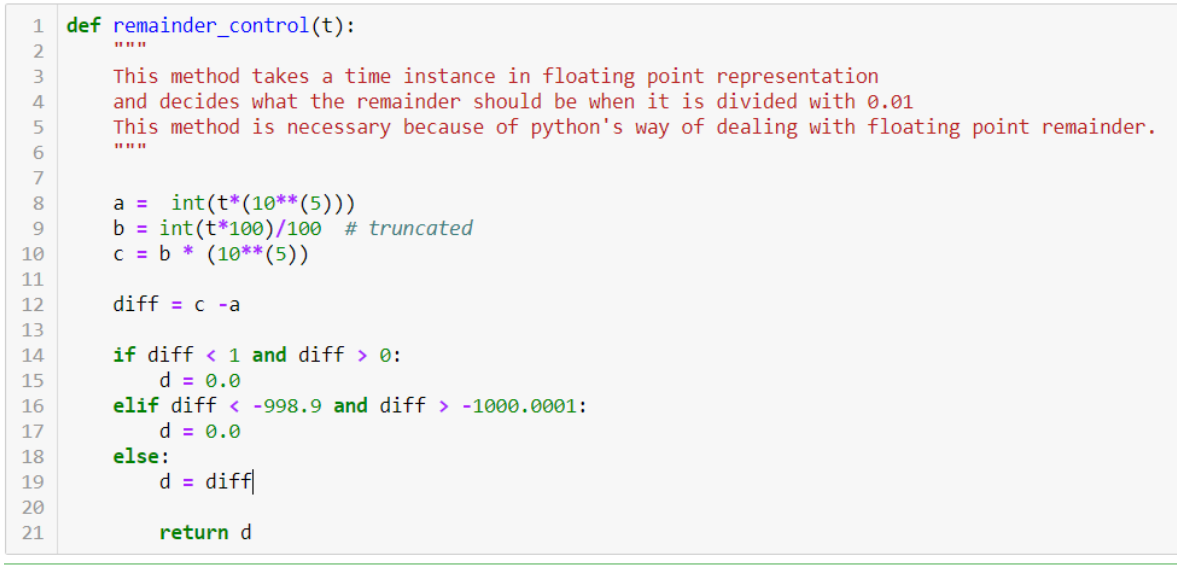

In [11]:

```python
for t in time:

    remainder =  (100*t) % 1

    #if remainder < 1.e-10:
    #    remainder = 0

    if remainder != 0:
        print('t:{}, r:{}'.format(t,remainder))
```

```
t:0.07, r:8.881784197001252e-16
t:0.14, r:1.7763568394002505e-15
t:0.28, r:3.552713678800501e-15
t:0.29, r:0.9999999999999964
t:0.55, r:7.105427357601002e-15
t:0.56, r:7.105427357601002e-15
t:0.57, r:0.9999999999999929
t:0.58, r:0.9999999999999929
t:1.09, r:1.4210854715202004e-14
t:1.1, r:1.4210854715202004e-14
t:1.11, r:1.4210854715202004e-14
t:1.12, r:1.4210854715202004e-14
t:1.13, r:0.9999999999999858
t:1.14, r:0.9999999999999858
t:1.15, r:0.9999999999999858
t:1.16, r:0.9999999999999858
t:2.01, r:0.9999999999999716
t:2.03, r:0.9999999999999716
t:2.05, r:0.9999999999999716
t:2.07, r:0.9999999999999716
```

The algorithm tries to find the points that should be included by checking whether the time instant is divisible by the time step. For example, it needs to include the point that corresponds to 0.07secs but it should not include 0.0701. The initial way to do this was as the second figure below. However, we can not trust floating point remainder operations after observing this. Therefore, a method that walks around this problem is proposed.

Recep Oğuz Araz
30.07.2019



```python
def remainder_control(t):
    """
    This method takes a time instance in floating point representation
    and decides what the remainder should be when it is divided with 0.01
    This method is necessary because of python's way of dealing with floating point remainder.
    """

    a =  int(t*(10**(5)))
    b = int(t*100)/100  # truncated
    c = b * (10**(5))

    diff = c -a

    if diff < 1 and diff > 0:
        d = 0.0
    elif diff < -998.9 and diff > -1000.0001:
        d = 0.0
    else:
        d = diff

        return d
```

The idea is to look at the difference between the original point up to five decimal points and up to two decimal points and amplifying this distance for filtering. Four, five and six decimal points are tried and five is chosen as it is the most stable one with the lowest number of multiplications.

This filtering method works correct and it is validated over 7 recordings spanning 24 minutes with numerous floating-point combinations. However, a more solid way of dealing with this problem can be good for using the algorithm over a range of computers.

Results

Before the algorithm, it was taking about *4-5 hours* for an unexperienced user to correct a *single* pitch track. After correcting the pitch tracks of *7 Makam* recordings manually and using the algorithm for smoothing, it is observed that a total of 11 hours is spent. Here we see a considerable reduction of the time spent. The validation of the algorithm's accuracy rate is open for testing in the makam-following tool.

Conclusion

An algorithm that automatizes the pitch track smoothing process is presented. It depends not on signal processing techniques but simple observations about the process of smoothing. It works incredibly fast with the octave errors compared to the traditional way and with the upcoming user interface it will speed up even more. Also, it makes the pitch track smoothing a more intuitive process, requiring only the guidance of the user. Overall, it can result in 50% time saving for a trained ear.

Future Work

- Octave error suggestions,
- Octave error user interface,

Recep Oğuz Araz
30.07.2019

- Solid floating point remainder.