Introduction to Musical Corpus Studies

Release 0.0.1

Fabian C. Moss

Nov 25, 2020

CONTENT

Warning: This material is still (heavily) under construction and might change throughout the course!

You can help improving the course and let me know about any errors and inconsistencies that you find or suggest other ways of improving the course.

Welcome!

These pages present the content of the course "Introduction to Musical Corpus Studies" at the Institute of Musicology, given at University of Cologne in Fall 2020.

In the last two decades *Musical Corpus Studies* evolved from a niche discipline into a veritable research area. The growing availability of digital and digitized musical data as well as the application and development of modern methodologies from computer science, machine learning, and data science cast new light on old musicological questions and generate entirely novel approaches to empirical music research.

Moreover, the general methodological and epistemological approach of Musical Corpus Studies allows to transcend traditional intra-musicological boundaries between its sub-disciplintes (historical/systematic/ethnological/...) without sacrificing the respective specific viewpoints and perspectives.

This course offers a fundamental and practical introduction into these topics. It demonstrates, explores, and critically reflects central thematic areas and methods by means of a number of case studies. In the engagement with these topics the course also introduces elementary methods from natural language and music processing, as well as statistics, data analysis and visualization.

The course is aimed at students at the undergraduate level who have little or no empirical background and are curious about quantitative approaches to musicology.

CHAPTER

ONE

ORGANIZATION

1.1 About this course

This course aims at providing an example-based introduction to the rapidly developing field of Musical Corpus Studies (MCS). Introducing a field that relies equally on musicological domain knowledge and skills in computational and statistical methods faces obvious challenges: while most people interested in this field come with a background in either area, few people are versed in both, and it can take years to bridge the musicological-computational gap.

In particular, systematic introductions to programming or specific musicological topics can be at times quite arduous, even boring, because it takes a long time to proceed from learning basic concepts to acually interesting problems. The problems and "toy examples" that are presented to introduce the basic concepts are necessarily remote from real-world applications and challenging research problems.

This course takes an alternative route. It does not start with an introduction to the programming language Python (which will be used throughout to carry out the computational analyses) but rather showcases a number of recent corpus studies that take on musicological research questions. The focus thus lies in understanding how aspects of music can be studied with computational methods and by analyzing musical corpora.

If this sparks your interest, it will be much easier to pick up the basics for yourself, knowing what they are *for* and being motivated intrinsically. If you are not particularly interested in doing this kind of work yourself, you will still see a broad range of applications that are much more useful to you than learning (or not learning) programming basics.

1.2 Overview

This year's course takes place on two weekends (13-14 November and 11-12 December 2020), comprising twelve sessions in total. The topics cover a broad range of musicological topics, from folk melodies and Jazz solos, over harmonies in Beethoven's string quartets and 20th century Pop music, to Renaissance candences and metric patterns in Malian drum music (see Table 1.2).

No.	Date	Time	Topics
1	Fr., 13.11.2020	16:00-17:20 Uhr	Introduction / Background
2		17:40-19:00 Uhr	Melody I: Folk song melodies, notes, form
3	Sa., 14.11.2020	09:00-10:20 Uhr	Melody II: The melodic arc, intervals
4		10:40-12:00 Uhr	Melody III: Jazz solos
		12:00-13:00 Uhr	*Lunch Break*
5		13:00-14:20 Uhr	Group work
6		14:40-16:00 Uhr	Beethoven's string quartets
7	Fr., 11.12.2020	16:00-17:20 Uhr	Cadences in Renaissance polyphony (guest: Richard Freedman)
8		17:40-19:00 Uhr	
9	Sa., 12.12.2020	09:00-10:20 Uhr	Rhythm & Meter: Malian percussion music
10		10:40-12:00 Uhr	Timbre: Electronic Music 1950-1990
		12:00-13:00 Uhr	*Lunch Break*
11		13:00-14:20 Uhr	Group work
12		14:40-16:00 Uhr	Recapitulation and conclusion

1.3 Credits

Active participation in this course is compensated with 3 credit points (CPs), equivalent to a work load of 90 hours. These are distributed as follows: 24 SWS (à 45 minutes) are allocated to presence in the block seminar. Additionally, 24 SWS are dedicated to the preparation and follow-up of the material. The remainder of 42 SWS goes to the reading of the relevant literature.

1.4 Deliverables and learning objectives

Apart from attending and following the presentations by the lecturer, course work consists of three main parts: preparing the relevant literature (reading), completing the assigned exercises (group work), and critically engaging with the course materials in the form of a report written together with your group (report).

These deliverables will broaden your knowledge and understanding of current musicological research, enhance your organizational and social skills, and help you to develop efficient work-load management strategies. Finally, compiling a report will advance your communication and writing abilities.

Reading

For each session, the relevant literature is cited in the text and provided on ILIAS. Careful preparation of the reading material is required in order to be able to follow the content of the course. Because the course will mainly talk about methods and general points of musical corpus research, the content (and musical topic) will mainly be introduced by the literature.

I am aware that the reading workload is relatively high since the course will be taught as a block seminar and doesn't spread out over the entire semester. I hope that the fact that the course is finished before the end of the year compensates for this.

Group work

At the beginning of the course, you will be randomly assigned to a group. Together with your group (which will stay fixed for the entire semester), you will work on a number of exercises during the course, e.g. in Zoom breakout rooms. You will collaborate on specific tasks related to the topic at hand, discuss methodological questions, and help each other in the understanding of some of the concepts that are introduced in the course.

Report

After the course has ended, your group will be randomly assigned a course topic (one of the twelve sessions in Table 1.2). It is your task to write a report on this theme. The should be 6–8 pages long.

Questions that you could address are: What did you learn? Which concepts are not clear? Which methods did you (not) understand? What is missing? How can the textual descriptions be improved? Who in your group did what? Was the presentation of the material adequate? If not, what was missing? Please also write about the organization of your group, challenges and benefits.

Recommended structure for the report

1. **Introduction:** general description and summary of the course and your assigned session in particular.
2. **Discussion:** summarize the main discussion, open questions, and how you would continue this line or research.
3. **Issues:** describe in detail what was crucial for your understanding of the topic, what was missing, etc.
4. **Various:** anything that you would like to write in the report
5. **Author contributions:** describe briefly how each of you specifically contributed to the report.

Important: Submit your report by **31 January 2021, 23:59h** to fabian.moss@epfl.ch as a single PDF file per group, named *intro_corpusmus_<group_number>.pdf*, e.g. *intro_corpusmus_1.pdf*.

CHAPTER

TWO

INTRODUCTION AND BACKGROUND

Note: The slides for the introduction can be found here: pdf

CHAPTER

THREE

MELODIES IN FOLK SONGS

On Jupyter Hub, change the kernel to Python 3.7!

[1]:
```
import pandas as pd
import music21 as m21
import numpy as np
import statsmodels.api as sm

import matplotlib.pyplot as plt
import matplotlib as mpl

import seaborn as sns
sns.set_context("notebook")
```

[2]:
```
## Tragen Sie hier bitte Ihren username ein:
# USERNAME = "fmoss"

## for jupyter hubs
# %env QT_QPA_PLATFORM=offscreen
# # new user, create music21 environment variables.
# m21.environment.set('musicxmlPath', value='/usr/bin/mscore')
# m21.environment.set('musescoreDirectPNGPath', value='/usr/bin/mscore')
# m21.environment.set('graphicsPath', value=f'/home/{USERNAME}') # change accordingly␣
↪for your own username!
```

3.1 The *Essen Folksong Collection*

In this session, we work with a corpus of melodies, the *Essen Folksong Collection* (EFC). There are several ways to access this corpus, for example through the interface provided by the Center for Computer Assisted Research in the Humanities (CCARH) at Stanford University: http://essen.themefinder.org/ or via http://kern.ccarh.org/browse?l=essen.

A more convenient way to work with the pieces is by using the Python library `music21`. This library was developed and is maintaned my Mike Cuthbert at the MIT and is the most popular library for the computational analysis of symbolic music (i.e. scores). You can find its documentation here: http://web.mit.edu/music21/

However, using `music21` requires some training and getting used to its particular API (the way how to interact with its functions). We will not get into too many details here but rather showcase how it can be used for our purposes.

The first thing we do is to load the entire EFC and store it in a variable named `corpora`.

[3]:
```
# load corpus
corpora = m21.corpus.getComposer('essenFolksong')
```

Calling the variable `corpora` shows that it consists of a list of file paths. Using the `len()` function, we can find out how many corpora are stored in the variable `corpora`.

```
[4]: len(corpora)
```

```
[4]: 31
```

We can also directly call the variable `corpora` to see what it contains:

```
[5]: corpora
```

```
[5]: [WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/altdeu10.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/altdeu20.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad10.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad20.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad30.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad40.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad50.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad60.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad70.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/ballad80.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/boehme10.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/boehme20.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/dva0.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/erk10.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/erk20.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/erk30.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/erk5.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/fink0.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/folkHaydn.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/han1.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/han2.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/irl.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/kinder0.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/lot.abc'),
```

(continues on next page)

(continued from previous page)

```
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/lux.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/test0.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/test1.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/testd.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/teste.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/variant0.abc'),
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
↪essenFolksong/zuccal0.abc')]
```

The variable `corpora` is a list of file paths, each of which points to a corpus in this collection. Note that the location depends on the location where `music21` is installed. If you would do this on your own computer, you would see different paths. The file names at the end of the file paths indicate what they contain, e.g. `altdeu10.abc` contains old German folksongs, `boehme10.abc` contains Czech folksongs, and `han1.abc` contains Chinese folksongs.

The `.abc` file ending refers to the ABC notation for encoding melodies. You find more information about the ABC encoding here: http://abcnotation.com/

For example, a song could be encoded like this:

[6]:
```
example_song = """
X:1
T:Speed the Plough
M:4/4
C:Trad.
K:G
|:GABc dedB|dedB dedB|c2ec B2dB|c2A2 A2BA|
  GABc dedB|dedB dedB|c2ec B2dB|A2F2 G4:|
|:g2gf gdBd|g2f2 e2d2|c2ec B2dB|c2A2 A2df|
  g2gf g2Bd|g2f2 e2d2|c2ec B2dB|A2F2 G4:|
"""
```

The tripple quotes (`"""`) surrounding the ABC notation are used by Python to store multi-line text.

What can we already understand from this encoding?

`music21` can load this string and display a graphical output of the score. This is done by a **parser**. A parser is a program that reads a file and produces a structured output.

[7]:
```
parsed_example_song = m21.converter.parse(example_song)
```

We did not need to give it the entire string again because we have already saved it in the `example_song` variable. The purpose of variables is that you can refer to them later in your code without explicitly needing to state its value.

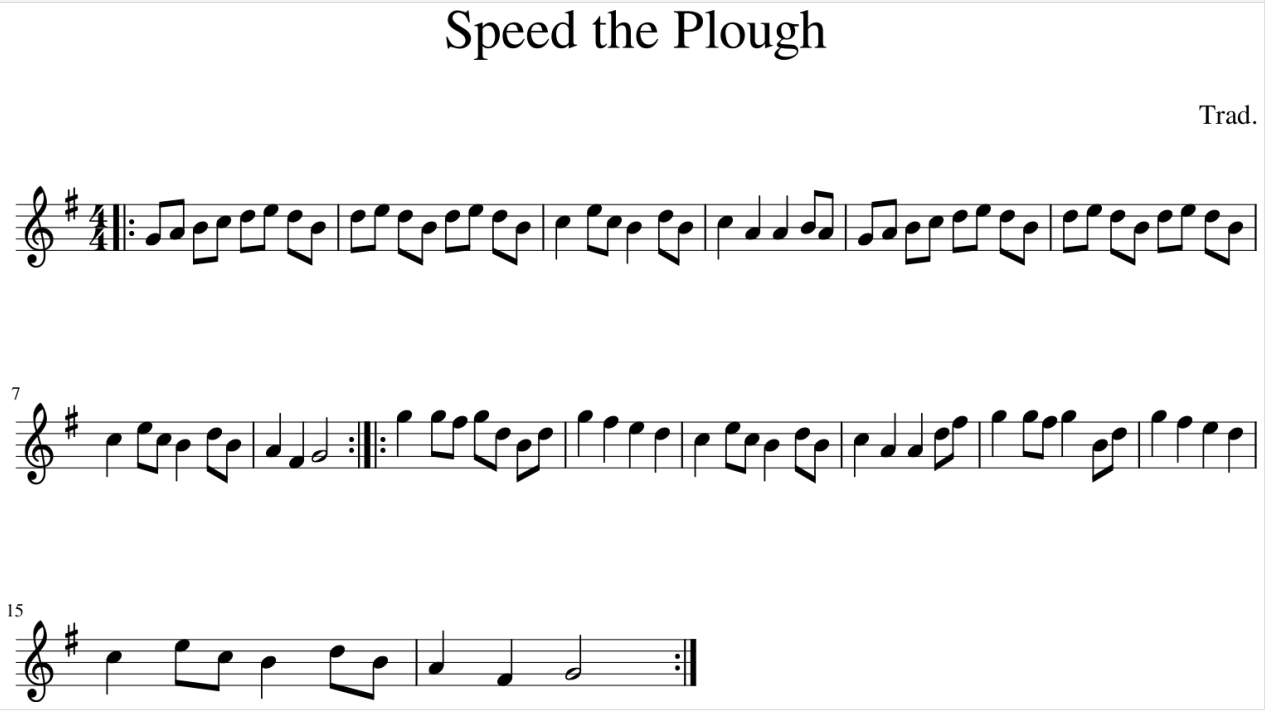
Calling the variable `parsed_example_song` now, however, does not really help us here...

[8]:
```
parsed_example_song
```

[8]:
```
<music21.stream.Score 0x1898a474340>
```

It returns a somewhat cryptic statement that says that the variable countains a `music21.stream.Score` object. Understanding the internal organization of `music21` goes beyond this class. For us, it is suffient to know that these objects have certain associated functions, called **methods**, that we can use on them. To look at the score of this example song, we use the method `.show()`.

[9]:
```
parsed_example_song.show()
```



Voilà, this is much better! Now, let us compare the score output to the ABC encoding of the song:

[10]:
```
print(example_song)
```

```
X:1
T:Speed the Plough
M:4/4
C:Trad.
K:G
|:GABc dedB|dedB dedB|c2ec B2dB|c2A2 A2BA|
  GABc dedB|dedB dedB|c2ec B2dB|A2F2 G4:|
|:g2gf gdBd|g2f2 e2d2|c2ec B2dB|c2A2 A2df|
  g2gf g2Bd|g2f2 e2d2|c2ec B2dB|A2F2 G4:|
```

Now the ABC notation makes already more sense. `T:Speed the Ploug` stands for the title, `M:4/4` for the meter, and `K:G` for the key of the song. The ABC documentation tells us that `X:1` encodes just a reference number, in case multiple pieces are stored in the same file (as in our case in the variable `corpora`, remember?). And the lines at the bottom encode the proper melody, where the letters represent note names that are organized into bars with or without repetition signs.

`music21` even gives us the option to listen to the song if we path the `midi` argument to the `.show()` method:

[11]:
```
parsed_example_song.show("midi")
```

```
<IPython.core.display.HTML object>
```

Now, what happens if we try to parse one of the corpora in the EFC? We can select a specific corpus by its **index** in the list. Python starts counting at 0, so the first file in the list corresponds to

```
[12]: corpora[0]
```

```
[12]: WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/
      ↪essenFolksong/altdeu10.abc')
```

As you can see, this is just the first file path in the variable `corpora`. Let's try to parse it!

```
[13]: first_corpus = m21.converter.parse(corpora[0])
```

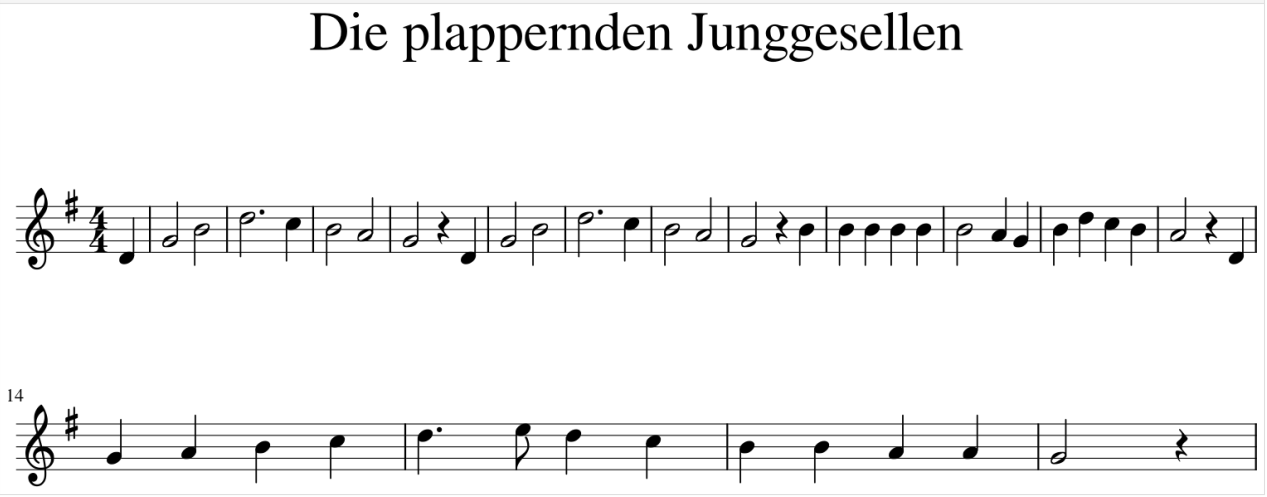
Looking at the new variable `first_corpus` shows a difference to the example song before; we don't have a `music21.stream.Score` object but a `music21.stream.Opus` object.

```
[14]: first_corpus
```

```
[14]: <music21.stream.Opus 0x1898b5ff4c0>
```

If we would call the `.show()` method on `first_corpus`, we would see the scores of all pieces that are in this particular corpus. But we don't know how many these are. It there are only three songs, it would not be a problem, but if there were thousands of songs, it could take a very long time to parse and display them all. Fortunately, all pieces in the collection have the `X:n` line that we saw above, so that we can directly reference them. With which number would we have to replace `n` if we wanted to look at the 7tst piece? Remember that Python starts counting at 0.

```
[15]: first_corpus[70].show()
```



A A B A'

```
[16]: first_corpus[70].show("midi")
```

```
<IPython.core.display.HTML object>
```

We have seen that we can select items from lists by **indexing** them, `list[i]`. We can get ranges of lists by using the `:` character. For example, `list[:10]` shows the first ten elements, `list[10:]` shows everything after the ninth element, and `list[3:6]` shows elements 3, 4, and 5 (not 6!) of the list.

3.2 Comparing songs

Looking at individual songs is interesting for music analysis but for that the computational approach is not really necessary. We could as easily do the same by just looking at a book of scores. The power of computational methods becomes clearer when we start comparing different songs, potentially in a large number.

To facilitate this comparison, we will first load all songs in all corpora of the EFC into a single list, called `songs` (this might take a couple of minutes).

[17]:
```
songs = [s for i in range(len(corpora)) for s in m21.converter.parse(corpora[i]) ]
```

This looks a bit complicated but all it does is to go through all corpora and extract all songs into a new list. The way we did it is called **list comprehension** in Python. It is not important if you don't understand this now but feel free to look it up!

Using the `len()` function again, we see how many songs we have in total.

[18]:
```
len(songs)
```

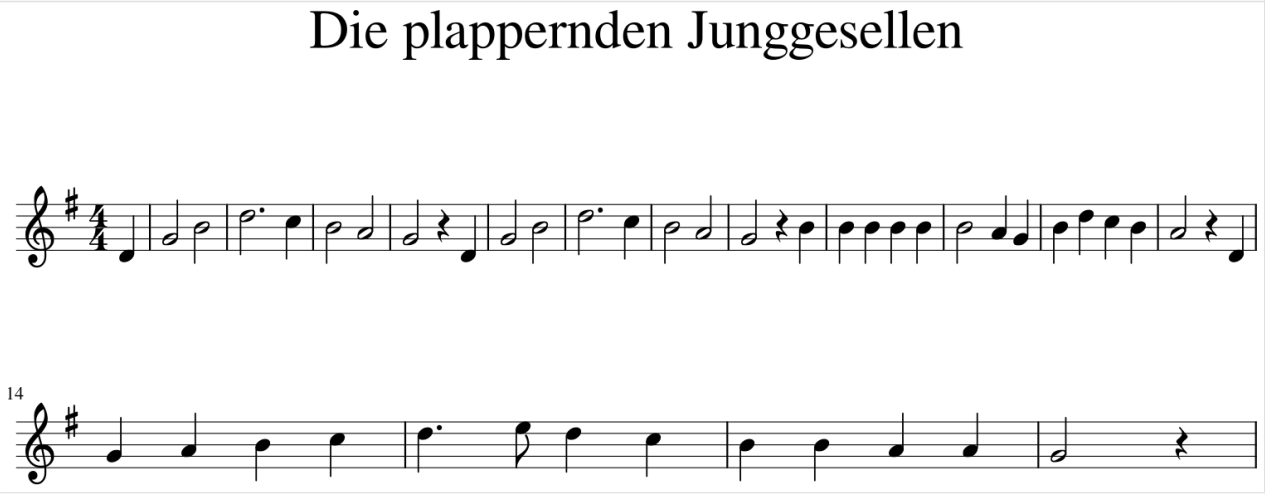
[18]:
```
8514
```

We can now use the list `songs` to compare two different songs. Again, we load the 71st song of the first corpus and store it now in a variable `german_song`, and we load chinese song with index 6200 into the variable `chinese_song`.

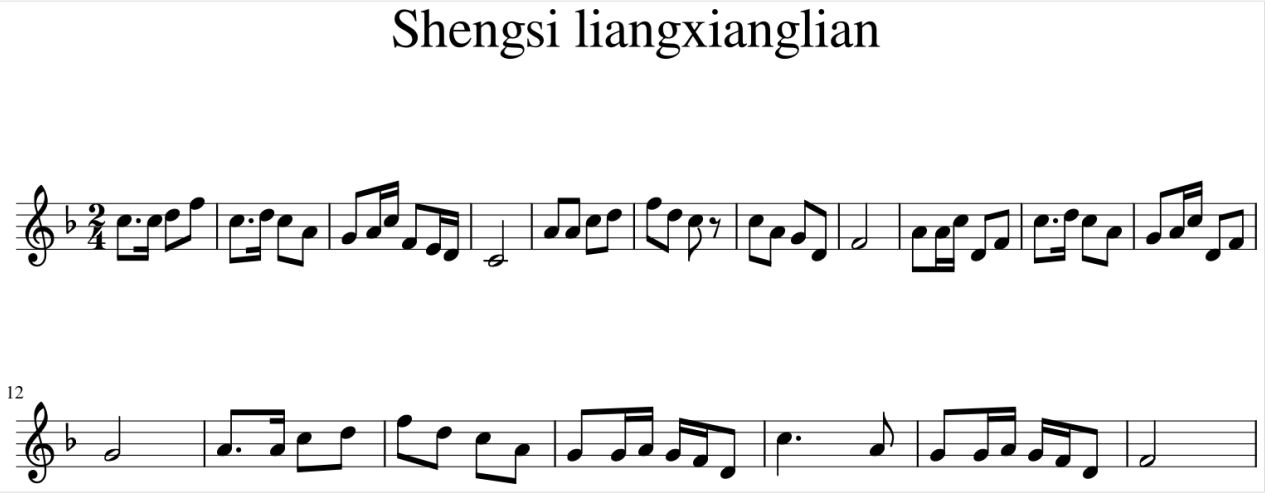
[19]:
```
german_song = songs[70]
chinese_song = songs[6200]
```

It is easy to display these songs now:

[20]:
```
german_song.show()
```



[21]:
```
chinese_song.show()
```



[22]:
```
chinese_song.show("midi")
```
```
<IPython.core.display.HTML object>
```

Analysis of songs...

3.3 Computational analysis

We now go on to a computational analysis of these two and all the other songs. Specifically, we wil compare their **melodic profiles**. To make things a bit simpler, we will just look at the notes.

A note can be easily represented as a pair of **pitch** (its height) and its **duration**. For example, the first note of the *Die plappernden Junggesellen* could be represented as `(D4, 1/4)`; it is a quarter note on the pitch D4 (the 4 indicates the octave in which the note is).

Another way to represent the pitch of notes is using **MIDI numbers**. MIDI stands for *Musical Instrument Digital Interface* and was developed for the communication between different electronic instruments such as keyboards. In MIDI, each note is simply associated with a number:

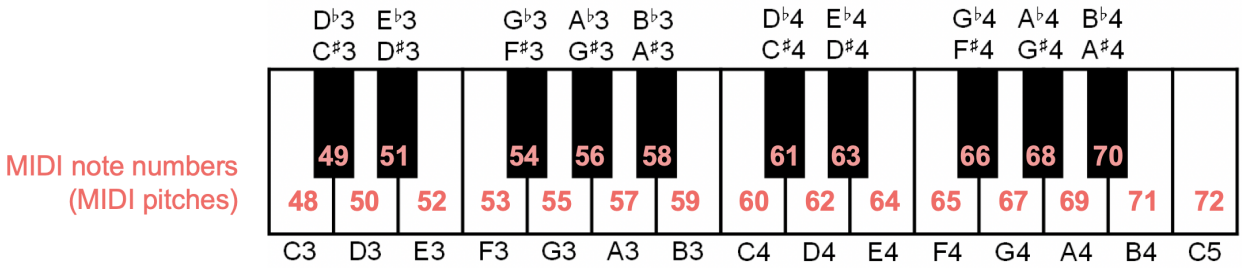


Image from https://www.audiolabs-erlangen.de/resources/MIR/FMP/C1/C1S2_MIDI.html.

We can see that D4 is associated with the number 62. The second note, the G4, is associated with 62+5=67 because G is five semitones above D.

To make it easier to work with pieces in this way, we define a **function** that gives us a list of notes for each piece.

[23]:
```
def notelist(piece):
    """
    This function takes a song as input and returns a list of (pitch, duration) pairs,
    where the duration is given in quarter notes.
```

(continues on next page)

(continued from previous page)

```
    """

    df = pd.DataFrame([ (note.pitch.midi, note.quarterLength) for note in piece.flat.
↪notes ], columns=["MIDI Pitch", "Duration"])
    df["Onset"] = df["Duration"].cumsum()

    return df
```

Note that the duration of a note is given in quarter notes, i.e. a quarter note has a duration of 1, a half note has a duration of 2, and an eighth note has a duration of 0.5.

Let's display the first phrase (the first eight notes) of the German song:

[24]:
```
notelist(german_song)[:8]
```

[24]:

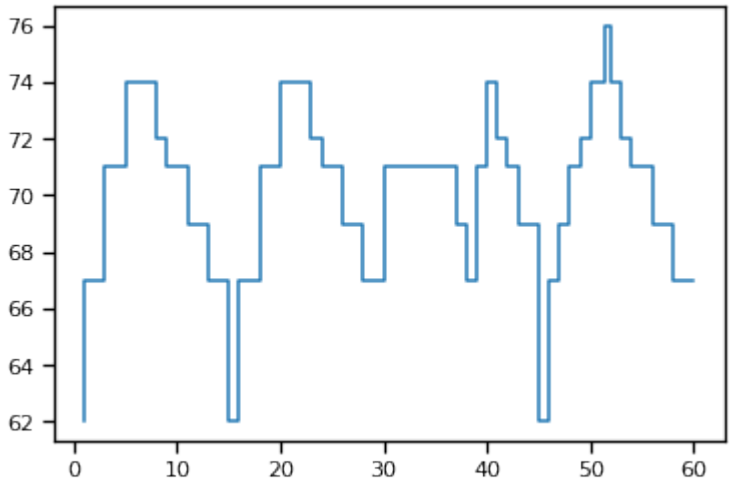
	MIDI Pitch	Duration	Onset
0	62	1.0	1.0
1	67	2.0	3.0
2	71	2.0	5.0
3	74	3.0	8.0
4	72	1.0	9.0
5	71	2.0	11.0
6	69	2.0	13.0
7	67	2.0	15.0

Note that we added another column, "Onset". What does it represent?

This allows us now to look at the **melodic profile** of a particular song.

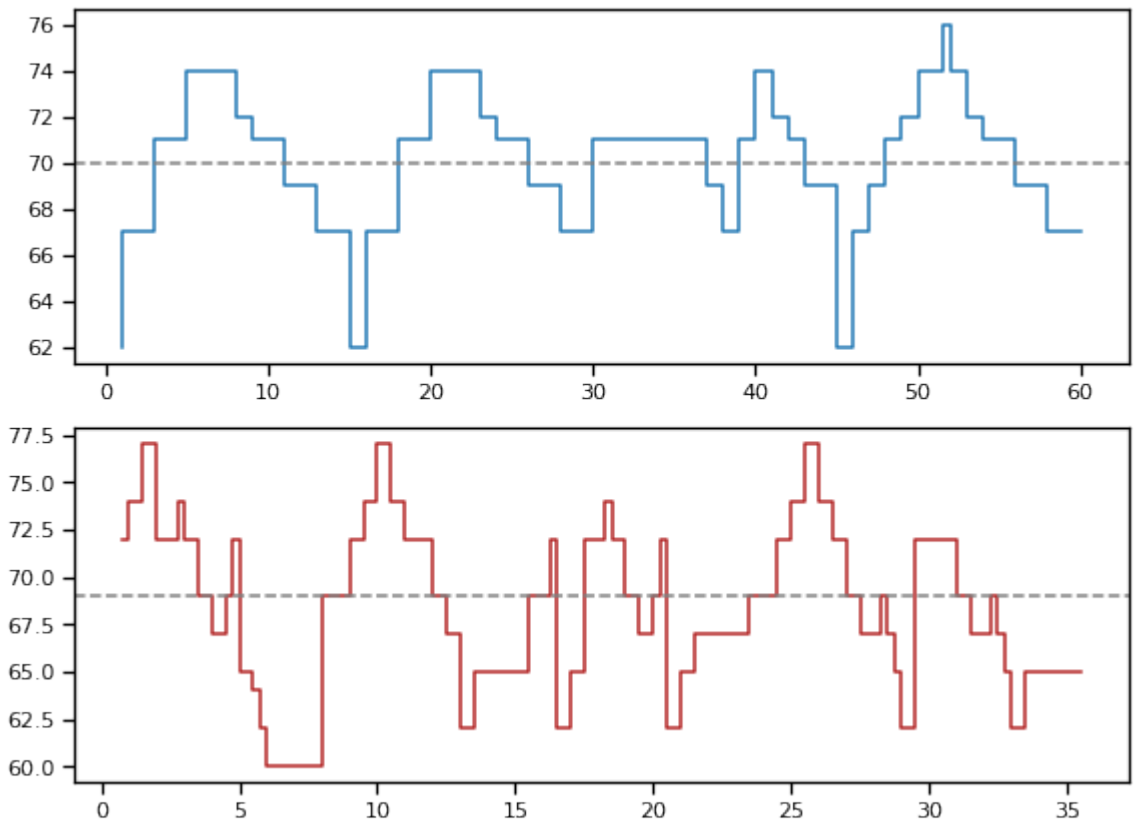
[25]:
```
def plot_melodic_profile(notelist, ax=None, c=None, mean=False, Z=False,␣
↪sections=False, standardized=False):

    if ax == None:
        ax = plt.gca()

    if standardized:
        x = notelist["Rel. Onset"]
        y = notelist["Rel. MIDI Pitch"]
    else:
        x = notelist["Onset"]
        y = notelist["MIDI Pitch"]

    ax.step(x,y, color=c)

    if mean:
        ax.axhline(y.mean(), color="gray", linestyle="--")

    if sections:
        for l in [ x.max() * i for i in [ 1/4, 1/2, 3/4] ]:
            ax.axvline(l, color="gray", linewidth=1, linestyle="--")
```

[26]:
```
plot_melodic_profile(notelist(german_song))
```



Likewise, we can as easily plot the melodic contour of the Chinese song (we will use a different color).

[27]:
```
fig, axes = plt.subplots(2,1, figsize=(8,6))

plot_melodic_profile(notelist(german_song), ax=axes[0], mean=True)
plot_melodic_profile(notelist(chinese_song), ax=axes[1], c="firebrick", mean=True)

plt.tight_layout()
plt.savefig("img/melodic_profiles.png")
```



The dashed grey lines in both plots show the average MIDI pitch of the song.

But still, it is quite difficult to compare them directly. They differ both with respect to their length (see the numbers on the "Onset" axis) and their pitches (see "MIDI Pitch" axis).

We need to transform them in a way that makes them directly comparable. To that end, we define a new function `standardize()`.

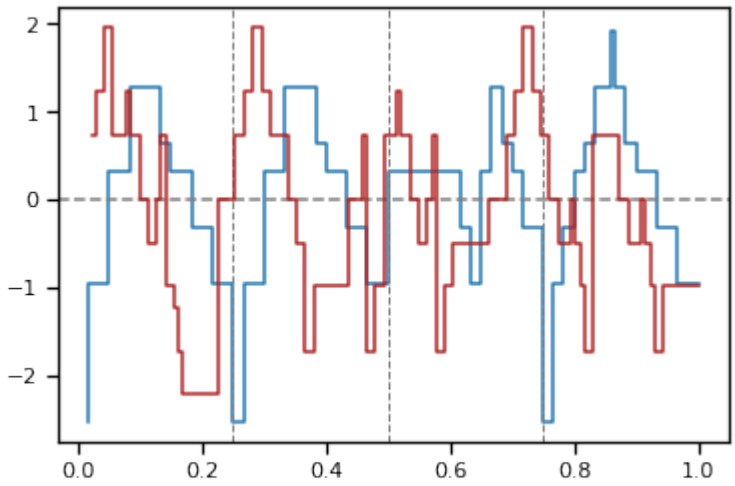
[28]:
```
def standardize(notelist):
    """
    Takes a notelist as input and returns a standardized version.
    """

    notelist["Rel. MIDI Pitch"] = (notelist["MIDI Pitch"] - notelist["MIDI Pitch"].
↪mean()) / notelist["MIDI Pitch"].std()
    notelist["Rel. Duration"] = notelist["Duration"] / notelist["Duration"].sum()
    notelist["Rel. Onset"] = notelist["Onset"] / notelist["Onset"].max()

    return notelist
```

[29]:
```
standardize(notelist(german_song))[:8]
```

[29]:

	MIDI Pitch	Duration	Onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset
0	62	1.0	1.0	-2.543827	0.016667	0.016667
1	67	2.0	3.0	-0.949300	0.033333	0.050000
2	71	2.0	5.0	0.326322	0.033333	0.083333
3	74	3.0	8.0	1.283038	0.050000	0.133333
4	72	1.0	9.0	0.645227	0.016667	0.150000
5	71	2.0	11.0	0.326322	0.033333	0.183333
6	69	2.0	13.0	-0.311489	0.033333	0.216667
7	67	2.0	15.0	-0.949300	0.033333	0.250000

[30]:
```
plot_melodic_profile(standardize(notelist(german_song)), mean=True, sections=True,␣
↪standardized=True)
plot_melodic_profile(standardize(notelist(chinese_song)), c="firebrick",␣
↪standardized=True)
```



Standardizing the songs makes it possible to compare them directly: They have now the same length 1 and their pitches are centered around the mean 0 with a standard deviation of 1.

However, already with two pieces this plot is quite crowded.

[31]:
```
dfs = [ ]

for i, song in enumerate(songs):
    df = standardize(notelist(song))
    df["Song ID"] = i
    dfs.append(df)

big_df = pd.concat(dfs).reset_index(drop=True)
```

[53]:
```
big_df
```

[53]:
```
        MIDI Pitch  Duration  Onset  Rel. MIDI Pitch  Rel. Duration  \
0               67      2.00   2.00        -1.819039       0.013158
1               70      2.00   4.00        -0.741977       0.013158
2               71      2.00   6.00        -0.382956       0.013158
3               72      2.00   8.00        -0.023935       0.013158
4               72      2.00  10.00        -0.023935       0.013158
...            ...       ...    ...              ...            ...
450591          71      0.25  28.50         0.691456       0.008197
450592          69      0.25  28.75         0.098779       0.008197
450593          73      0.25  29.00         1.284133       0.008197
450594          71      1.00  30.00         0.691456       0.032787
450595          69      0.50  30.50         0.098779       0.016393

        Rel. Onset  Song ID  Avg. MIDI Pitch  shifted_pitch
0         0.013158        0               72             -5
1         0.026316        0               72             -2
2         0.039474        0               72             -1
3         0.052632        0               72              0
4         0.065789        0               72              0
...            ...      ...              ...            ...
450591    0.934426     8513               68              3
450592    0.942623     8513               68              1
450593    0.950820     8513               68              5
450594    0.983607     8513               68              3
450595    1.000000     8513               68              1

[450596 rows x 9 columns]
```

[63]:
```
big_df.to_csv("data/big_df.csv") # comma-separated values
```

[66]:
```
big_df = pd.read_csv("data/big_df.csv")
```

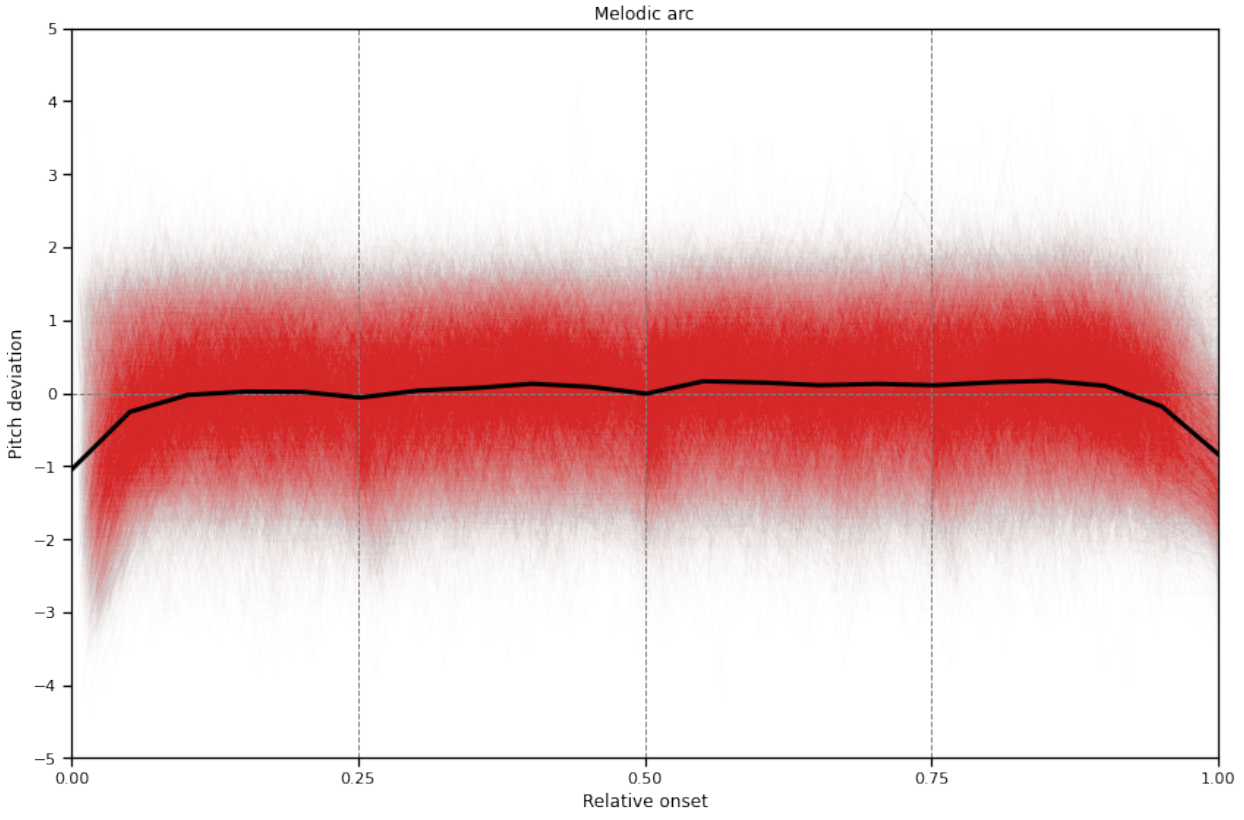
3.4 The melodic arc

[35]:
```
%%time

fig, ax = plt.subplots(figsize=(12,8))

grouped = big_df.groupby("Song ID")

for i, g in grouped:
    x = g["Rel. Onset"]
    y = g["Rel. MIDI Pitch"]
```

(continues on next page)

(continued from previous page)

```
    ax.plot(x,y, lw=.5, c="tab:red", alpha=1/100)

ax.axvline(.25, lw=1, ls="--", c="gray")
ax.axvline(.5, lw=1, ls="--", c="gray")
ax.axvline(.75, lw=1, ls="--", c="gray")
ax.axhline(0, lw=1, ls="--", c="gray")

lowess = sm.nonparametric.lowess
big_x = big_df["Rel. Onset"]
big_y = big_df["Rel. MIDI Pitch"]
big_z = lowess(big_y, big_x, frac=5/100, delta=1/20) # Locally-Weighted Scatterplot␣
↪Smoothing
ax.plot(big_z[:,0], big_z[:,1], c="black", lw=3)

plt.title("Melodic arc")
plt.xlabel("Relative onset")
plt.ylabel("Pitch deviation")
plt.xticks(np.linspace(0,1,5))
plt.yticks(np.linspace(-5,5,11))
plt.xlim(0,1)

plt.tight_layout()
plt.savefig("img/melodic_arc.png")
plt.show()
```



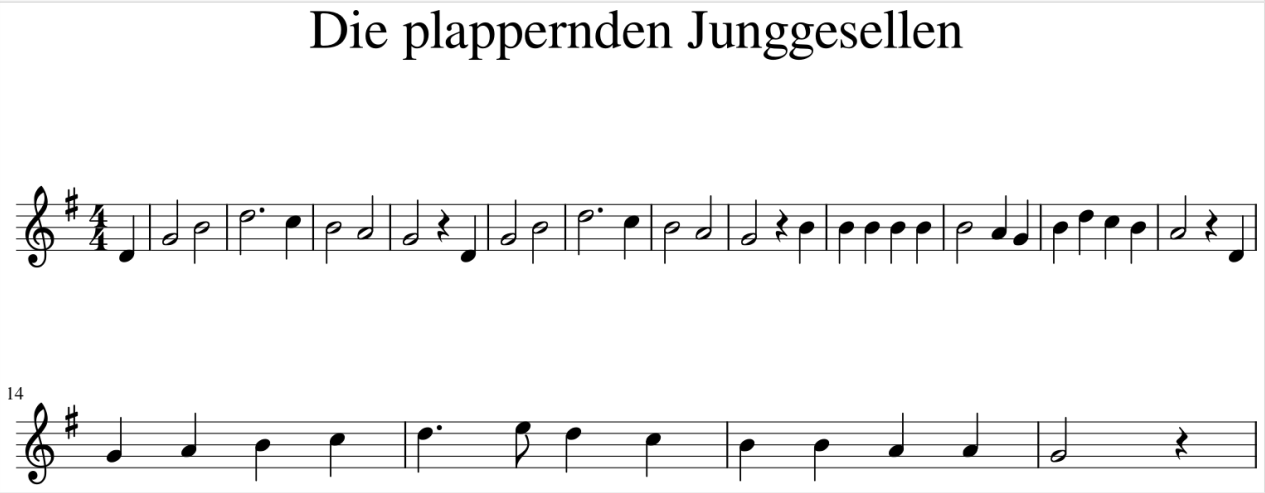

```
Wall time: 24.1 s
```

3.5 Intervals

We have seen that the melodic arc emerges as a stable shape over the entire EFC, and that sub-phrases of the songs likewise have an arc-like shape. In the remainder of this section, we look at **intervals**, the distance between two notes.

Let's come back to the song *Die plappernden Junggesellen*

[36]:
```
german_song.show()
```



We have already extracted its notes and stored them in a DataFrame:

[69]:
```
big_df[ big_df["Song ID"] == 70].head(8)
```

[69]:

	Unnamed: 0	Unnamed: 0.1	MIDI Pitch	Duration	Onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset	Song ID
2969	2969	2969	62	1.0	1.0	-2.543827	0.016667	0.016667	70
2970	2970	2970	67	2.0	3.0	-0.949300	0.033333	0.050000	70
2971	2971	2971	71	2.0	5.0	0.326322	0.033333	0.083333	70
2972	2972	2972	74	3.0	8.0	1.283038	0.050000	0.133333	70
2973	2973	2973	72	1.0	9.0	0.645227	0.016667	0.150000	70
2974	2974	2974	71	2.0	11.0	0.326322	0.033333	0.183333	70
2975	2975	2975	69	2.0	13.0	-0.311489	0.033333	0.216667	70
2976	2976	2976	67	2.0	15.0	-0.949300	0.033333	0.250000	70

The code above reads as "Select all rows in `big_df` for which the column `Song ID` is equal to 70". The `.head()` method displays the first 5 rows by default but you can specify the number of rows you want to be displayed (here 8).

Focusing on the "MIDI Pitch" column, the notes in the first phrase have MIDI pitch 62, 67, 71, 74, 72. Since intervals correspond to the difference between notes, the intervals for the beginning of this song are:

- +5 (67-62)
- +4 (71-67)

- +3 (74-71)
- -2 (72-74)
- -1 (71-72)
- -2 (69-71)
- -2 (67-69)

The sequence of intervals in this phrase is thus `[+5, +4, +3, -2, -1, -2, -2]`. The signs (+ or -) also reflect the arc-like shape of this first phrase, but the sizes of the intervals are not perfecly balanced. Note that `-2` (two descending semitones, or one descending whole tone) is the most frequent interval.

[38]:
```
all_ints = [ p2 - p1 for i, g in big_df.groupby("Song ID") for p1, p2 in zip(g["MIDI␣
↪Pitch"], g["MIDI Pitch"][1:]) ]
min_int = min(all_ints)
max_int = max(all_ints)
```

[39]:
```
min_int, max_int
```

[39]:
```
(-25, 25)
```

[40]:
```
len(all_ints)
```

[40]:
```
442082
```

[41]:
```
ints_df = pd.DataFrame(0, index=np.arange(min_int,max_int), columns=np.arange(min_int,
↪max_int+1))

for i, g in big_df.groupby("Song ID"):
    intervals = [ p2 - p1 for p1, p2 in zip(g["MIDI Pitch"], g["MIDI Pitch"][1:])]

    for i1, i2 in zip(intervals, intervals[1:]):
        ints_df.loc[i1,i2] += 1
```

[73]:
```
ints_df.loc[-10:10, -10:10]
```

[73]:

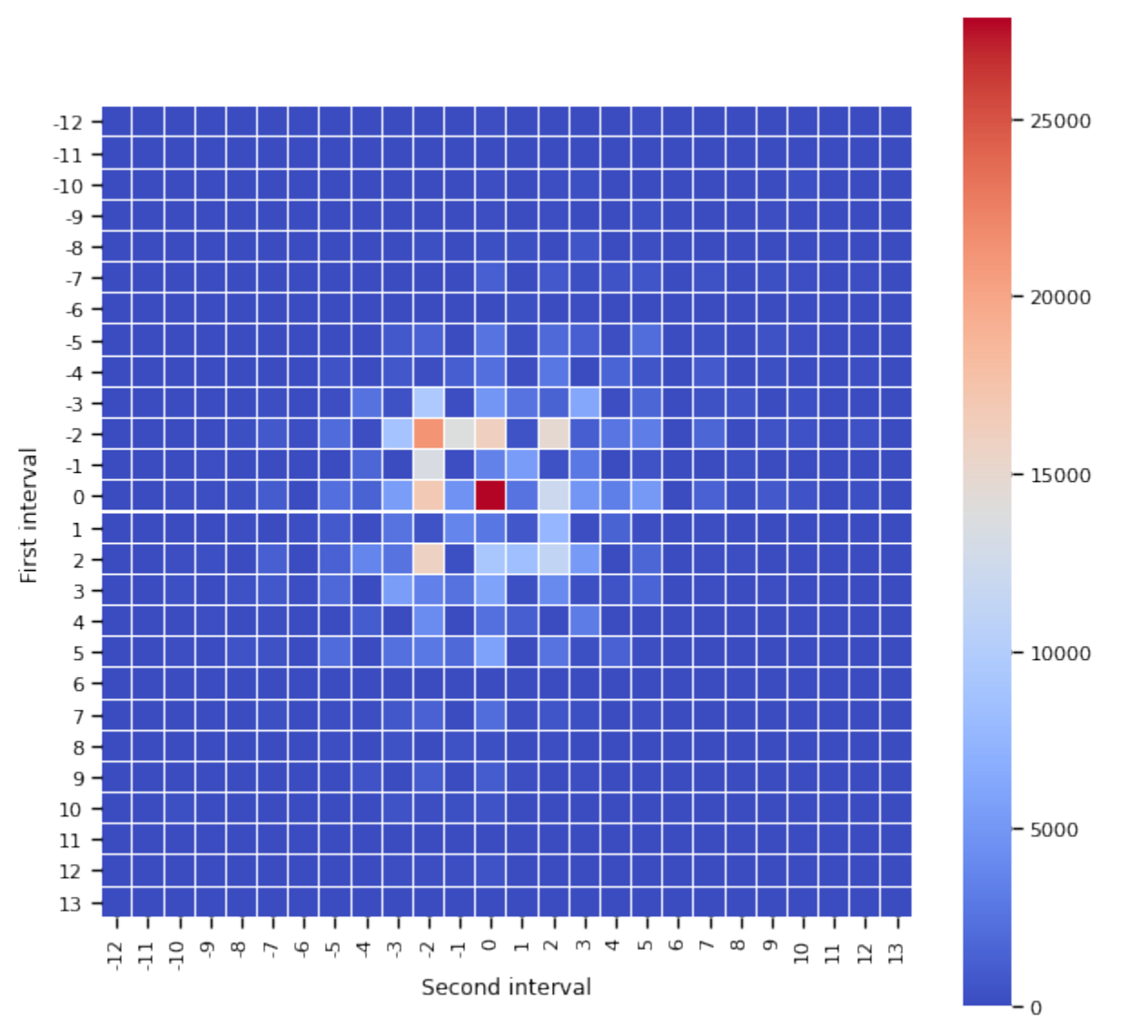
	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	...	1	\
-10	0	0	0	0	0	2	3	2	34	0	...	16	
-9	0	0	0	6	0	3	0	39	10	31	...	5	
-8	0	1	0	1	0	0	19	0	319	5	...	302	
-7	0	0	2	14	0	37	2	91	96	103	...	17	
-6	0	0	0	0	0	0	12	11	8	21	...	274	
-5	1	0	1	49	0	75	32	866	1361	25	...	230	
-4	3	0	27	11	5	461	18	692	167	1099	...	129	
-3	1	91	0	192	2	215	2490	416	9478	134	...	2547	
-2	67	14	260	858	132	1964	113	8871	21285	13896	...	454	
-1	5	174	4	68	47	32	1679	91	13445	205	...	5410	
0	186	130	310	1022	80	2270	1440	5506	16887	4603	...	2578	
1	55	6	174	43	110	944	165	2564	501	3765	...	747	
2	138	294	29	1288	15	1361	3754	2562	15795	254	...	8404	
3	272	23	373	655	122	1755	24	5509	3397	2400	...	295	
4	5	164	3	130	11	51	1026	64	4217	60	...	1133	
5	116	23	505	330	13	1996	82	2375	2834	1868	...	102	
6	2	4	1	4	23	1	14	18	44	34	...	42	
7	31	18	11	273	3	167	128	665	1338	59	...	119	
8	47	1	88	7	19	149	4	370	33	384	...	23	
9	1	61	1	20	3	20	442	18	1024	4	...	122	

(continues on next page)

(continued from previous page)

```
 10  267    0   56    23   28    58     0   517    163    295  ...      3

         2     3     4     5    6     7    8    9    10
-10    174   219    60    91    0    78   67   29  243
-9     159    39    40   313    2    30    0  111   27
-8      66   605     3   159    2     4   73    0   32
-7     859   300   527   651    1   399   18  219  141
-6       7   132    25    10   19     6    3    3    4
-5    1906  1272   148  2131   22   252   87  368  160
-4    2738    40  1610   616   13   884    8  269   31
-3    1467  6274   109  1684   13   403  508   59  261
-2   14883  1115  2616  3216   63  1681   88  537  407
-1     396  2878    58   477   44    23  241    3   52
 0   12142  4947  3340  5203   30  1353  292  824  478
 1    7649   193  1470   132   26   172    9   59    2
 2   11261  5249    86  1695    2   187  171   57   61
 3    4128   231   519  1523   35   180    0  122   11
 4      67  3153    20   102    7     1   14    0    2
 5    2557   270  1355   160    1    94    0   16    5
 6       8    20     0     0    0     0    0    0    0
 7     566   249    12   166    0     8    2    6    0
 8     159     0    23     6    0     1    0    0    0
 9      13   129     1     8    0     0    5    0    0
 10     78     0     4     5    0     0    0    0    0

[21 rows x 21 columns]
```

[74]:
```
fig, ax = plt.subplots(figsize=(10,10))
sns.heatmap(ints_df.loc[-12:13,-12:13], cmap="coolwarm", square=True, linewidths=0.01,
↪ax=ax)
plt.ylabel("First interval")
plt.xlabel("Second interval")
plt.show()
```



The two most common interval pairs are `(0,0)` and `(-2,-2)`. A much less frequent pair of intervals is `(5,0)`, but this is still much more frequent than, for example, `(9,9)`.

To which melodic fragments do these correspond?

[44]:
```
big_df["Avg. MIDI Pitch"] = 0

for i, group in big_df.groupby("Song ID"):
    grp_mean_pitch = int(group["MIDI Pitch"].mean())
    big_df.loc[big_df["Song ID"] == i, "Avg. MIDI Pitch"] = grp_mean_pitch
```

[45]:
```
big_df["shifted_pitch"] = big_df["MIDI Pitch"] - big_df["Avg. MIDI Pitch"]
```

[46]:
```
big_df.tail()
```

[46]:

	MIDI Pitch	Duration	Onset	Rel. MIDI Pitch	Rel. Duration	\
450591	71	0.25	28.50	0.691456	0.008197	
450592	69	0.25	28.75	0.098779	0.008197	

(continues on next page)

(continued from previous page)

```
450593          73      0.25  29.00          1.284133       0.008197
450594          71      1.00  30.00          0.691456       0.032787
450595          69      0.50  30.50          0.098779       0.016393

        Rel. Onset  Song ID  Avg. MIDI Pitch  shifted_pitch
450591    0.934426     8513               68              3
450592    0.942623     8513               68              1
450593    0.950820     8513               68              5
450594    0.983607     8513               68              3
450595    1.000000     8513               68              1
```

[47]:
```
idx = np.arange(big_df["shifted_pitch"].min(), big_df["shifted_pitch"].max() + 1)
idx
```

[47]:
```
array([-16, -15, -14, -13, -12, -11, -10,  -9,  -8,  -7,  -6,  -5,  -4,
        -3,  -2,  -1,   0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
        10,  11,  12,  13,  14,  15,  16,  17])
```

[48]:
```
transitions_df = pd.DataFrame(0, index=idx, columns=idx)
transitions_df
```

[48]:

	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	...	8	9	10
-16	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-15	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-14	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-13	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-12	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-11	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-10	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-9	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-8	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-7	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-6	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-5	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-4	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-3	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-2	0	0	0	0	0	0	0	0	0	0	...	0	0	0
-1	0	0	0	0	0	0	0	0	0	0	...	0	0	0
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0
5	0	0	0	0	0	0	0	0	0	0	...	0	0	0
6	0	0	0	0	0	0	0	0	0	0	...	0	0	0
7	0	0	0	0	0	0	0	0	0	0	...	0	0	0
8	0	0	0	0	0	0	0	0	0	0	...	0	0	0
9	0	0	0	0	0	0	0	0	0	0	...	0	0	0
10	0	0	0	0	0	0	0	0	0	0	...	0	0	0
11	0	0	0	0	0	0	0	0	0	0	...	0	0	0
12	0	0	0	0	0	0	0	0	0	0	...	0	0	0
13	0	0	0	0	0	0	0	0	0	0	...	0	0	0
14	0	0	0	0	0	0	0	0	0	0	...	0	0	0
15	0	0	0	0	0	0	0	0	0	0	...	0	0	0
16	0	0	0	0	0	0	0	0	0	0	...	0	0	0
17	0	0	0	0	0	0	0	0	0	0	...	0	0	0

(continues on next page)

(continued from previous page)

```
      11   12   13   14   15   16   17
-16    0    0    0    0    0    0    0
-15    0    0    0    0    0    0    0
-14    0    0    0    0    0    0    0
-13    0    0    0    0    0    0    0
-12    0    0    0    0    0    0    0
-11    0    0    0    0    0    0    0
-10    0    0    0    0    0    0    0
-9     0    0    0    0    0    0    0
-8     0    0    0    0    0    0    0
-7     0    0    0    0    0    0    0
-6     0    0    0    0    0    0    0
-5     0    0    0    0    0    0    0
-4     0    0    0    0    0    0    0
-3     0    0    0    0    0    0    0
-2     0    0    0    0    0    0    0
-1     0    0    0    0    0    0    0
 0     0    0    0    0    0    0    0
 1     0    0    0    0    0    0    0
 2     0    0    0    0    0    0    0
 3     0    0    0    0    0    0    0
 4     0    0    0    0    0    0    0
 5     0    0    0    0    0    0    0
 6     0    0    0    0    0    0    0
 7     0    0    0    0    0    0    0
 8     0    0    0    0    0    0    0
 9     0    0    0    0    0    0    0
 10    0    0    0    0    0    0    0
 11    0    0    0    0    0    0    0
 12    0    0    0    0    0    0    0
 13    0    0    0    0    0    0    0
 14    0    0    0    0    0    0    0
 15    0    0    0    0    0    0    0
 16    0    0    0    0    0    0    0
 17    0    0    0    0    0    0    0

[34 rows x 34 columns]
```

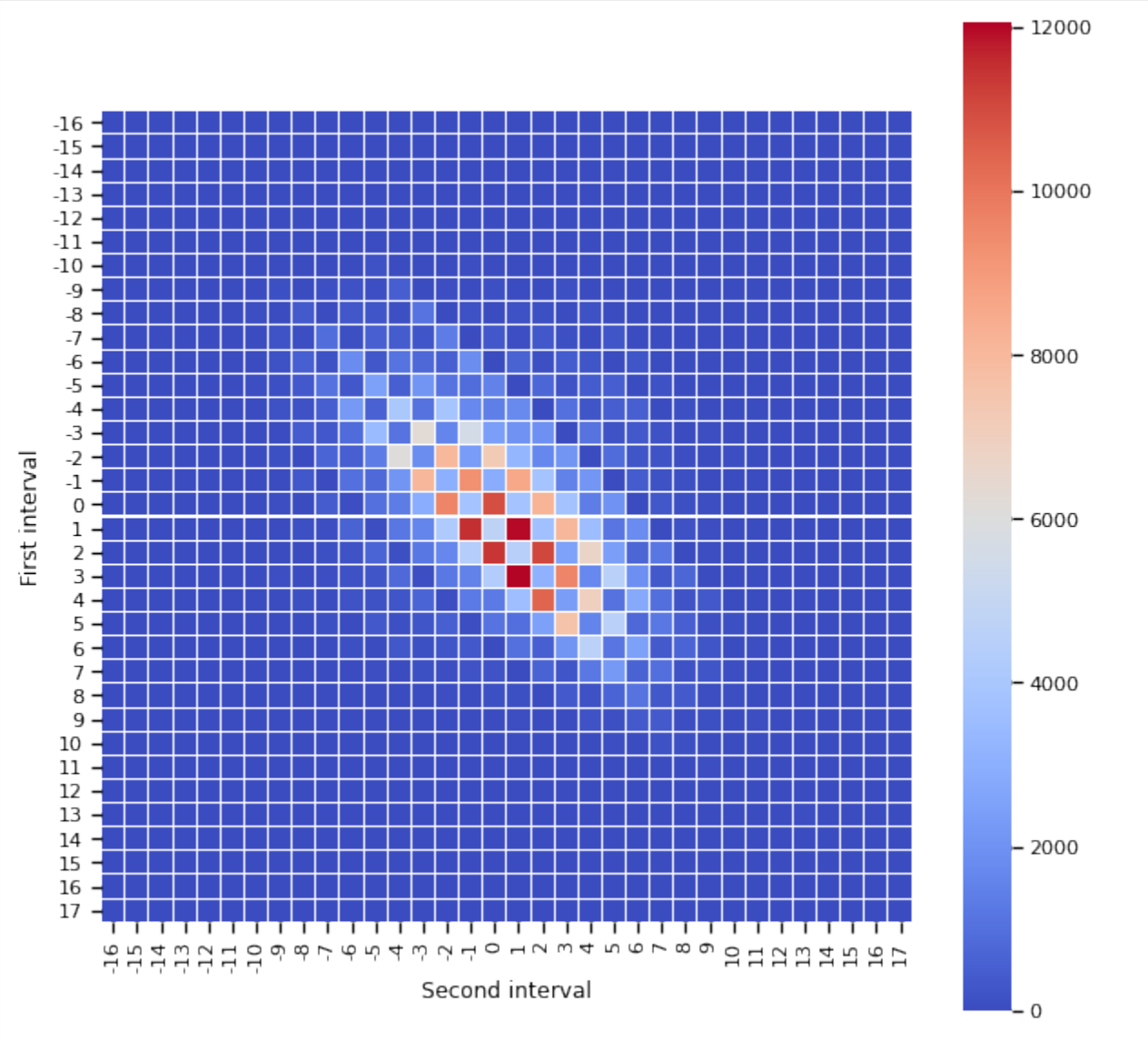
[49]:
```
%%time

for i, group in big_df.groupby("Song ID"):
    for bg in zip(group["shifted_pitch"], group["shifted_pitch"][1:]):
        transitions_df.loc[bg[0],bg[1]] +=1
```

```
Wall time: 1min 29s
```

[50]:
```
print(f"There are {transitions_df.sum().sum()} intervals in total in the corpus.")
```

```
There are 442082 intervals in total in the corpus.
```

[51]:
```
fig, ax = plt.subplots(figsize=(10,10))

g = sns.heatmap(transitions_df, cmap="coolwarm", linewidths=.01, square=True)
plt.ylabel("First interval")
plt.xlabel("Second interval")
plt.show()
```


First interval
Second interval
-16
-15
-14
-13
-12
-11
-10
-9
-8
-7
-6
-5
-4
-3
-2
-1
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
12000
10000
8000
6000
4000
2000
0

[ ]:

CHAPTER

FOUR

SOLOS IN THE *WEIMAR JAZZ DATABASE*

Disclaimer: I am not the expert here!

In this session, we will have a look at is the Jazzomat Research Project that contains the *Weimar Jazz Database* (WJazzD). Let us first browse the site.

One of the outcomes of this research project is the freely-available book:

- Pfleiderer, M., Frieler, K., Abeßer, J., Zaddach, W.-G., & Burkhard, B. (Eds.) (2017). Inside the Jazzomat. New Perspectives for Jazz Research. Mainz: Schott Campus (Open Access).

[1]:
```
import pandas as pd
import numpy as np
import statsmodels.api as sm
import sqlite3

import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("white")
sns.set_context("talk")
```

The WJazzD can be downloaded at https://jazzomat.hfm-weimar.de/download/download.html A local copy of the database is stored at `data/wjazz.db`. We use the `sqlite3` library to connect to this database.

[2]:
```
conn = sqlite3.connect("data/wjazzd.db")
```

[3]:
```
conn
```

[3]:
```
<sqlite3.Connection at 0x201747b7990>
```

We can now use `pandas` to read the data out of the database.

[4]:
```
solos = pd.read_sql("SELECT * FROM melody", con=conn)
```

The `"SELECT * FROM melody"` means "Select everything from the table 'melody' in the database". Let's look at the first ten entries.

Likewise, we can select the `composition_info` table that contains a lot of metadata for the solos:

[5]:
```
solos_meta = pd.read_sql("SELECT * from solo_info", con=conn)
```

The `.shape` attribute shows us how many solos are in the database.

[6]:
```
solos_meta.shape
```

[6]:
```
(456, 17)
```

The `.sample()` method draws a number of rows at random from a DataFrame.

[13]:
```
solos_meta.sample(5)
```

[13]:
```
     melid  trackid  compid  recordid      performer        title titleaddon  \
429    430      260     236       140  Wayne Shorter   Eighty-One
440    441      335     295       180     Woody Shaw     Rosewood
240    241      152     137        76  Johnny Hodges        Bunny
326    327      174     158        87    Miles Davis      Walkin'
224    225      199     177       103  John Coltrane  Impressions       1963

     solopart instrument    style  avgtempo tempoclass  rhythmfeel    key  \
429         1         ts  POSTBOP     138.4     MEDIUM       SWING  F-maj
440         1         tp  POSTBOP     171.7  MEDIUM UP  LATIN/FUNK
240         1         as    SWING     114.4     MEDIUM       SWING  C-maj
326         1         tp  HARDBOP     127.7     MEDIUM       SWING  F-maj
224         1         ts  POSTBOP     278.7         UP       SWING  D-min

    signature                                        chord_changes  chorus_count
429       4/4  A1: ||Fsus7   |Bbsus7   |Fsus7   |Fsus7   |Bbs...             5
440       4/4  A1: ||G-79 F-79 |G-79 F-79 |C-79 Bb-79 |Gb79  ...             1
240       4/4  A1: ||A-7 D7 |G E7 |A-7 D7 |D-7 G7 |C F7 |Bb B...             1
326       4/4  A1: ||F7   |Bb7   |F7   |F7   |Bb7   |Bb7   |F...             7
224       4/4  A1: ||D-7   |D-7   |D-7   |D-7   |D-7   |D-7  ...            13
```

The first rows of a DataFrame can be accessed with the `.head()` method...

[19]:
```
solos.head()
```

[19]:
```
   eventid  melid      onset  pitch  duration  period  division  bar  beat  \
0        1      1  10.343492   65.0  0.138776       4         1    0     1
1        2      1  10.637642   63.0  0.171247       4         4    0     2
2        3      1  10.843719   58.0  0.081270       4         4    0     2
3        4      1  10.948209   61.0  0.235102       4         1    0     3
4        5      1  11.232653   63.0  0.130612       4         1    0     4

   tatum  ...  f0_mod  loud_max   loud_med   loud_sd  loud_relpos loud_cent  \
0      1  ...          0.126209  66.526087  5.541147     0.307692  0.389466
1      1  ...          0.349751  69.133321  2.912412     0.250000  0.468687
2      4  ...          0.094051  66.352130  3.564563     0.428571  0.531354
3      1  ...          0.521187  66.484173  2.414298     0.818182  0.559333
4      1  ...          0.560737  71.699054  2.185794     0.166667  0.438973

   loud_s2b   f0_range  f0_freq_hz  f0_med_dev
0  1.056169  37.794261   12.932532   -0.328442
1  1.120317   6.365930    6.956935   11.135423
2  1.310389  68.010392         NaN   32.366787
3  0.984047  15.443906    5.867151   -3.374696
4  1.061262  11.444363    8.329975    6.377737

[5 rows x 26 columns]
```

... and the last rows with the `.tail()` method.

[16]:
```
solos.tail()
```

[16]:
```
        eventid  melid      onset  pitch  duration  period  division  bar  \
200804   200805    456  63.135057   57.0  0.168345       4         2   53
200805   200806    456  63.303401   55.0  0.087075       4         3   54
```

(continues on next page)

(continued from previous page)

```
200806   200807    456  63.390476   57.0  0.191565       4         3   54
200807   200808    456  63.640091   59.0  0.406349       4         1   54
200808   200809    456  64.058050   52.0  1.433832       4         2   54

        beat  tatum  ...    f0_mod  loud_max   loud_med   loud_sd  loud_relpos  \
200804     4      2  ...           1.113380  72.169552  6.896394     0.687500
200805     1      1  ...           0.491496  69.732265  1.814723     0.500000
200806     1      2  ...     slide 1.187058  76.628621  2.628726     0.411765
200807     2      1  ...           0.972676  66.042058  3.690577     0.000000
200808     3      2  ...   vibrato 0.368321  58.174931  9.418678     0.053030

       loud_cent  loud_s2b    f0_range  f0_freq_hz  f0_med_dev
200804  0.581956  1.271747  191.074095   10.966972  -11.891698
200805  0.595212  1.339060   40.375449         NaN  -99.173779
200806  0.590950  1.432802  104.823845   11.148561   -2.911604
200807  0.334937  1.082549  165.810976    2.659723   14.311001
200808  0.400571  1.278890   66.932198    2.153916   -9.381310

[5 rows x 26 columns]
```

As we already know, the `.shape` attribute shows the overall size of the table.

[20]:
```
solos.shape
```

[20]:
```
(200809, 26)
```

The `solos` table contains 26 columns that cannot be displayed at once. We can have a look at the column names by using the `.columns` attribute.

[21]:
```
solos.columns
```

[21]:
```
Index(['eventid', 'melid', 'onset', 'pitch', 'duration', 'period', 'division',
       'bar', 'beat', 'tatum', 'subtatum', 'num', 'denom', 'beatprops',
       'beatdur', 'tatumprops', 'f0_mod', 'loud_max', 'loud_med', 'loud_sd',
       'loud_relpos', 'loud_cent', 'loud_s2b', 'f0_range', 'f0_freq_hz',
       'f0_med_dev'],
      dtype='object')
```

A description of what these columns contain is stated on the website: https://jazzomat.hfm-weimar.de/dbformat/dbformat.html

For our analyses it will be usefull to have also the name of the performer in the `solos` DataFrame. We create a **dictionary** that maps the `melid` (unique identification number for each solo) to the name of the performer.

[26]:
```
mapper = dict( solos_meta[["melid", "performer"]].values )
mapper
```

[26]:
```
{1: 'Art Pepper',
 2: 'Art Pepper',
 3: 'Art Pepper',
 4: 'Art Pepper',
 5: 'Art Pepper',
 6: 'Art Pepper',
 7: 'Benny Carter',
 8: 'Benny Carter',
 9: 'Benny Carter',
 10: 'Benny Carter',
 11: 'Benny Carter',
```

(continues on next page)

(continued from previous page)

```
12: 'Benny Carter',
13: 'Benny Carter',
14: 'Benny Goodman',
15: 'Benny Goodman',
16: 'Benny Goodman',
17: 'Benny Goodman',
18: 'Benny Goodman',
19: 'Benny Goodman',
20: 'Benny Goodman',
21: 'Ben Webster',
22: 'Ben Webster',
23: 'Ben Webster',
24: 'Ben Webster',
25: 'Ben Webster',
26: 'Bix Beiderbecke',
27: 'Bix Beiderbecke',
28: 'Bix Beiderbecke',
29: 'Bix Beiderbecke',
30: 'Bix Beiderbecke',
31: 'Bob Berg',
32: 'Bob Berg',
33: 'Bob Berg',
34: 'Bob Berg',
35: 'Bob Berg',
36: 'Bob Berg',
37: 'Bob Berg',
38: 'Branford Marsalis',
39: 'Branford Marsalis',
40: 'Branford Marsalis',
41: 'Branford Marsalis',
42: 'Branford Marsalis',
43: 'Branford Marsalis',
44: 'Buck Clayton',
45: 'Buck Clayton',
46: 'Buck Clayton',
47: 'Cannonball Adderley',
48: 'Cannonball Adderley',
49: 'Cannonball Adderley',
50: 'Cannonball Adderley',
51: 'Cannonball Adderley',
52: 'Charlie Parker',
53: 'Charlie Parker',
54: 'Charlie Parker',
55: 'Charlie Parker',
56: 'Charlie Parker',
57: 'Charlie Parker',
58: 'Charlie Parker',
59: 'Charlie Parker',
60: 'Charlie Parker',
61: 'Charlie Parker',
62: 'Charlie Parker',
63: 'Charlie Parker',
64: 'Charlie Parker',
65: 'Charlie Parker',
66: 'Charlie Parker',
67: 'Charlie Parker',
68: 'Charlie Parker',
```

(continues on next page)

(continued from previous page)

```
 69: 'Charlie Shavers',
 70: 'Chet Baker',
 71: 'Chet Baker',
 72: 'Chet Baker',
 73: 'Chet Baker',
 74: 'Chet Baker',
 75: 'Chet Baker',
 76: 'Chet Baker',
 77: 'Chet Baker',
 78: 'Chris Potter',
 79: 'Chris Potter',
 80: 'Chris Potter',
 81: 'Chris Potter',
 82: 'Chris Potter',
 83: 'Chris Potter',
 84: 'Chris Potter',
 85: 'Chu Berry',
 86: 'Chu Berry',
 87: 'Clifford Brown',
 88: 'Clifford Brown',
 89: 'Clifford Brown',
 90: 'Clifford Brown',
 91: 'Clifford Brown',
 92: 'Clifford Brown',
 93: 'Clifford Brown',
 94: 'Clifford Brown',
 95: 'Clifford Brown',
 96: 'Coleman Hawkins',
 97: 'Coleman Hawkins',
 98: 'Coleman Hawkins',
 99: 'Coleman Hawkins',
100: 'Coleman Hawkins',
101: 'Coleman Hawkins',
102: 'Curtis Fuller',
103: 'Curtis Fuller',
104: 'David Liebman',
105: 'David Liebman',
106: 'David Liebman',
107: 'David Liebman',
108: 'David Liebman',
109: 'David Liebman',
110: 'David Liebman',
111: 'David Liebman',
112: 'David Liebman',
113: 'David Liebman',
114: 'David Liebman',
115: 'David Murray',
116: 'David Murray',
117: 'David Murray',
118: 'David Murray',
119: 'David Murray',
120: 'David Murray',
121: 'Dexter Gordon',
122: 'Dexter Gordon',
123: 'Dexter Gordon',
124: 'Dexter Gordon',
125: 'Dexter Gordon',
```

(continues on next page)

(continued from previous page)

```
126: 'Dexter Gordon',
127: 'Dickie Wells',
128: 'Dickie Wells',
129: 'Dickie Wells',
130: 'Dickie Wells',
131: 'Dickie Wells',
132: 'Dickie Wells',
133: 'Dizzy Gillespie',
134: 'Dizzy Gillespie',
135: 'Dizzy Gillespie',
136: 'Dizzy Gillespie',
137: 'Dizzy Gillespie',
138: 'Dizzy Gillespie',
139: 'Don Byas',
140: 'Don Byas',
141: 'Don Byas',
142: 'Don Byas',
143: 'Don Byas',
144: 'Don Byas',
145: 'Don Byas',
146: 'Don Byas',
147: 'Don Ellis',
148: 'Don Ellis',
149: 'Don Ellis',
150: 'Don Ellis',
151: 'Don Ellis',
152: 'Don Ellis',
153: 'Eric Dolphy',
154: 'Eric Dolphy',
155: 'Eric Dolphy',
156: 'Eric Dolphy',
157: 'Eric Dolphy',
158: 'Eric Dolphy',
159: 'Fats Navarro',
160: 'Fats Navarro',
161: 'Fats Navarro',
162: 'Fats Navarro',
163: 'Fats Navarro',
164: 'Fats Navarro',
165: 'Freddie Hubbard',
166: 'Freddie Hubbard',
167: 'Freddie Hubbard',
168: 'Freddie Hubbard',
169: 'Freddie Hubbard',
170: 'Freddie Hubbard',
171: 'George Coleman',
172: 'Gerry Mulligan',
173: 'Gerry Mulligan',
174: 'Gerry Mulligan',
175: 'Gerry Mulligan',
176: 'Gerry Mulligan',
177: 'Gerry Mulligan',
178: 'Hank Mobley',
179: 'Hank Mobley',
180: 'Hank Mobley',
181: 'Hank Mobley',
182: 'Harry Edison',
```

(continues on next page)

(continued from previous page)

```
183: 'Henry Allen',
184: 'Herbie Hancock',
185: 'Herbie Hancock',
186: 'Herbie Hancock',
187: 'Herbie Hancock',
188: 'Herbie Hancock',
189: 'J.C. Higginbotham',
190: 'J.J. Johnson',
191: 'J.J. Johnson',
192: 'J.J. Johnson',
193: 'J.J. Johnson',
194: 'J.J. Johnson',
195: 'J.J. Johnson',
196: 'J.J. Johnson',
197: 'J.J. Johnson',
198: 'Joe Henderson',
199: 'Joe Henderson',
200: 'Joe Henderson',
201: 'Joe Henderson',
202: 'Joe Henderson',
203: 'Joe Henderson',
204: 'Joe Henderson',
205: 'Joe Henderson',
206: 'Joe Lovano',
207: 'Joe Lovano',
208: 'Joe Lovano',
209: 'Joe Lovano',
210: 'Joe Lovano',
211: 'Joe Lovano',
212: 'Joe Lovano',
213: 'Joe Lovano',
214: 'John Abercrombie',
215: 'John Coltrane',
216: 'John Coltrane',
217: 'John Coltrane',
218: 'John Coltrane',
219: 'John Coltrane',
220: 'John Coltrane',
221: 'John Coltrane',
222: 'John Coltrane',
223: 'John Coltrane',
224: 'John Coltrane',
225: 'John Coltrane',
226: 'John Coltrane',
227: 'John Coltrane',
228: 'John Coltrane',
229: 'John Coltrane',
230: 'John Coltrane',
231: 'John Coltrane',
232: 'John Coltrane',
233: 'John Coltrane',
234: 'John Coltrane',
235: 'Johnny Dodds',
236: 'Johnny Dodds',
237: 'Johnny Dodds',
238: 'Johnny Dodds',
239: 'Johnny Dodds',
```

(continues on next page)

(continued from previous page)

```
240: 'Johnny Dodds',
241: 'Johnny Hodges',
242: 'Johnny Hodges',
243: 'Joshua Redman',
244: 'Joshua Redman',
245: 'Joshua Redman',
246: 'Joshua Redman',
247: 'Joshua Redman',
248: 'Kai Winding',
249: 'Kenny Dorham',
250: 'Kenny Dorham',
251: 'Kenny Dorham',
252: 'Kenny Dorham',
253: 'Kenny Dorham',
254: 'Kenny Dorham',
255: 'Kenny Dorham',
256: 'Kenny Garrett',
257: 'Kenny Garrett',
258: 'Kenny Wheeler',
259: 'Kenny Wheeler',
260: 'Kenny Wheeler',
261: 'Kid Ory',
262: 'Kid Ory',
263: 'Kid Ory',
264: 'Kid Ory',
265: 'Kid Ory',
266: 'Lee Konitz',
267: 'Lee Konitz',
268: 'Lee Konitz',
269: 'Lee Konitz',
270: 'Lee Konitz',
271: 'Lee Konitz',
272: 'Lee Konitz',
273: 'Lee Konitz',
274: 'Lee Morgan',
275: 'Lee Morgan',
276: 'Lee Morgan',
277: 'Lee Morgan',
278: 'Lester Young',
279: 'Lester Young',
280: 'Lester Young',
281: 'Lester Young',
282: 'Lester Young',
283: 'Lester Young',
284: 'Lester Young',
285: 'Lionel Hampton',
286: 'Lionel Hampton',
287: 'Lionel Hampton',
288: 'Lionel Hampton',
289: 'Lionel Hampton',
290: 'Lionel Hampton',
291: 'Louis Armstrong',
292: 'Louis Armstrong',
293: 'Louis Armstrong',
294: 'Louis Armstrong',
295: 'Louis Armstrong',
296: 'Louis Armstrong',
```

(continues on next page)

(continued from previous page)

```
297: 'Louis Armstrong',
298: 'Louis Armstrong',
299: 'Michael Brecker',
300: 'Michael Brecker',
301: 'Michael Brecker',
302: 'Michael Brecker',
303: 'Michael Brecker',
304: 'Michael Brecker',
305: 'Michael Brecker',
306: 'Michael Brecker',
307: 'Michael Brecker',
308: 'Michael Brecker',
309: 'Miles Davis',
310: 'Miles Davis',
311: 'Miles Davis',
312: 'Miles Davis',
313: 'Miles Davis',
314: 'Miles Davis',
315: 'Miles Davis',
316: 'Miles Davis',
317: 'Miles Davis',
318: 'Miles Davis',
319: 'Miles Davis',
320: 'Miles Davis',
321: 'Miles Davis',
322: 'Miles Davis',
323: 'Miles Davis',
324: 'Miles Davis',
325: 'Miles Davis',
326: 'Miles Davis',
327: 'Miles Davis',
328: 'Milt Jackson',
329: 'Milt Jackson',
330: 'Milt Jackson',
331: 'Milt Jackson',
332: 'Milt Jackson',
333: 'Milt Jackson',
334: 'Nat Adderley',
335: 'Nat Adderley',
336: 'Ornette Coleman',
337: 'Ornette Coleman',
338: 'Ornette Coleman',
339: 'Ornette Coleman',
340: 'Ornette Coleman',
341: 'Pat Martino',
342: 'Pat Metheny',
343: 'Pat Metheny',
344: 'Pat Metheny',
345: 'Pat Metheny',
346: 'Paul Desmond',
347: 'Paul Desmond',
348: 'Paul Desmond',
349: 'Paul Desmond',
350: 'Paul Desmond',
351: 'Paul Desmond',
352: 'Paul Desmond',
353: 'Paul Desmond',
```

(continues on next page)

(continued from previous page)

```
354: 'Pepper Adams',
355: 'Pepper Adams',
356: 'Pepper Adams',
357: 'Pepper Adams',
358: 'Pepper Adams',
359: 'Phil Woods',
360: 'Phil Woods',
361: 'Phil Woods',
362: 'Phil Woods',
363: 'Phil Woods',
364: 'Phil Woods',
365: 'Red Garland',
366: 'Rex Stewart',
367: 'Roy Eldridge',
368: 'Roy Eldridge',
369: 'Roy Eldridge',
370: 'Roy Eldridge',
371: 'Roy Eldridge',
372: 'Roy Eldridge',
373: 'Sidney Bechet',
374: 'Sidney Bechet',
375: 'Sidney Bechet',
376: 'Sidney Bechet',
377: 'Sidney Bechet',
378: 'Sonny Rollins',
379: 'Sonny Rollins',
380: 'Sonny Rollins',
381: 'Sonny Rollins',
382: 'Sonny Rollins',
383: 'Sonny Rollins',
384: 'Sonny Rollins',
385: 'Sonny Rollins',
386: 'Sonny Rollins',
387: 'Sonny Rollins',
388: 'Sonny Rollins',
389: 'Sonny Rollins',
390: 'Sonny Rollins',
391: 'Sonny Stitt',
392: 'Sonny Stitt',
393: 'Sonny Stitt',
394: 'Sonny Stitt',
395: 'Sonny Stitt',
396: 'Sonny Stitt',
397: 'Stan Getz',
398: 'Stan Getz',
399: 'Stan Getz',
400: 'Stan Getz',
401: 'Stan Getz',
402: 'Stan Getz',
403: 'Steve Coleman',
404: 'Steve Coleman',
405: 'Steve Coleman',
406: 'Steve Coleman',
407: 'Steve Coleman',
408: 'Steve Coleman',
409: 'Steve Coleman',
410: 'Steve Coleman',
```

(continues on next page)

(continued from previous page)

```
 411: 'Steve Coleman',
 412: 'Steve Coleman',
 413: 'Steve Lacy',
 414: 'Steve Lacy',
 415: 'Steve Lacy',
 416: 'Steve Lacy',
 417: 'Steve Lacy',
 418: 'Steve Lacy',
 419: 'Steve Turre',
 420: 'Steve Turre',
 421: 'Steve Turre',
 422: 'Von Freeman',
 423: 'Warne Marsh',
 424: 'Warne Marsh',
 425: 'Warne Marsh',
 426: 'Wayne Shorter',
 427: 'Wayne Shorter',
 428: 'Wayne Shorter',
 429: 'Wayne Shorter',
 430: 'Wayne Shorter',
 431: 'Wayne Shorter',
 432: 'Wayne Shorter',
 433: 'Wayne Shorter',
 434: 'Wayne Shorter',
 435: 'Wayne Shorter',
 436: 'Woody Shaw',
 437: 'Woody Shaw',
 438: 'Woody Shaw',
 439: 'Woody Shaw',
 440: 'Woody Shaw',
 441: 'Woody Shaw',
 442: 'Woody Shaw',
 443: 'Woody Shaw',
 444: 'Wynton Marsalis',
 445: 'Wynton Marsalis',
 446: 'Wynton Marsalis',
 447: 'Wynton Marsalis',
 448: 'Wynton Marsalis',
 449: 'Wynton Marsalis',
 450: 'Wynton Marsalis',
 451: 'Zoot Sims',
 452: 'Zoot Sims',
 453: 'Zoot Sims',
 454: 'Zoot Sims',
 455: 'Zoot Sims',
 456: 'Zoot Sims'}
```

We can now use this dictionary to create a new column `performer` in the `solos` DataFrame.

[28]:
```
solos["performer"] = solos["melid"].map(mapper)
```

[29]:
```
solos.head()
```

[29]:

	eventid	melid	onset	pitch	duration	period	division	bar	beat	\
0	1	1	10.343492	65.0	0.138776	4	1	0	1	
1	2	1	10.637642	63.0	0.171247	4	4	0	2	
2	3	1	10.843719	58.0	0.081270	4	4	0	2	

(continues on next page)

(continued from previous page)

```
3        4      1  10.948209   61.0  0.235102       4         1    0     3
4        5      1  11.232653   63.0  0.130612       4         1    0     4

   tatum  ...  loud_max   loud_med   loud_sd loud_relpos  loud_cent  loud_s2b  \
0      1  ...  0.126209  66.526087  5.541147    0.307692   0.389466  1.056169
1      1  ...  0.349751  69.133321  2.912412    0.250000   0.468687  1.120317
2      4  ...  0.094051  66.352130  3.564563    0.428571   0.531354  1.310389
3      1  ...  0.521187  66.484173  2.414298    0.818182   0.559333  0.984047
4      1  ...  0.560737  71.699054  2.185794    0.166667   0.438973  1.061262

    f0_range  f0_freq_hz  f0_med_dev   performer
0  37.794261   12.932532   -0.328442  Art Pepper
1   6.365930    6.956935   11.135423  Art Pepper
2  68.010392         NaN   32.366787  Art Pepper
3  15.443906    5.867151   -3.374696  Art Pepper
4  11.444363    8.329975    6.377737  Art Pepper

[5 rows x 27 columns]
```

[30]:
```
solos.tail()
```

[30]:
```
        eventid  melid      onset  pitch  duration  period  division  bar  \
200804   200805    456  63.135057   57.0  0.168345       4         2   53
200805   200806    456  63.303401   55.0  0.087075       4         3   54
200806   200807    456  63.390476   57.0  0.191565       4         3   54
200807   200808    456  63.640091   59.0  0.406349       4         1   54
200808   200809    456  64.058050   52.0  1.433832       4         2   54

        beat  tatum  ...  loud_max   loud_med   loud_sd loud_relpos  \
200804     4      2  ...  1.113380  72.169552  6.896394    0.687500
200805     1      1  ...  0.491496  69.732265  1.814723    0.500000
200806     1      2  ...  1.187058  76.628621  2.628726    0.411765
200807     2      1  ...  0.972676  66.042058  3.690577    0.000000
200808     3      2  ...  0.368321  58.174931  9.418678    0.053030

        loud_cent  loud_s2b    f0_range  f0_freq_hz  f0_med_dev  performer
200804   0.581956  1.271747  191.074095   10.966972  -11.891698  Zoot Sims
200805   0.595212  1.339060   40.375449         NaN  -99.173779  Zoot Sims
200806   0.590950  1.432802  104.823845   11.148561   -2.911604  Zoot Sims
200807   0.334937  1.082549  165.810976    2.659723   14.311001  Zoot Sims
200808   0.400571  1.278890   66.932198    2.153916   -9.381310  Zoot Sims

[5 rows x 27 columns]
```

4.1 Melodic arc?

Does the melodic arc also appear in the Jazz solos?

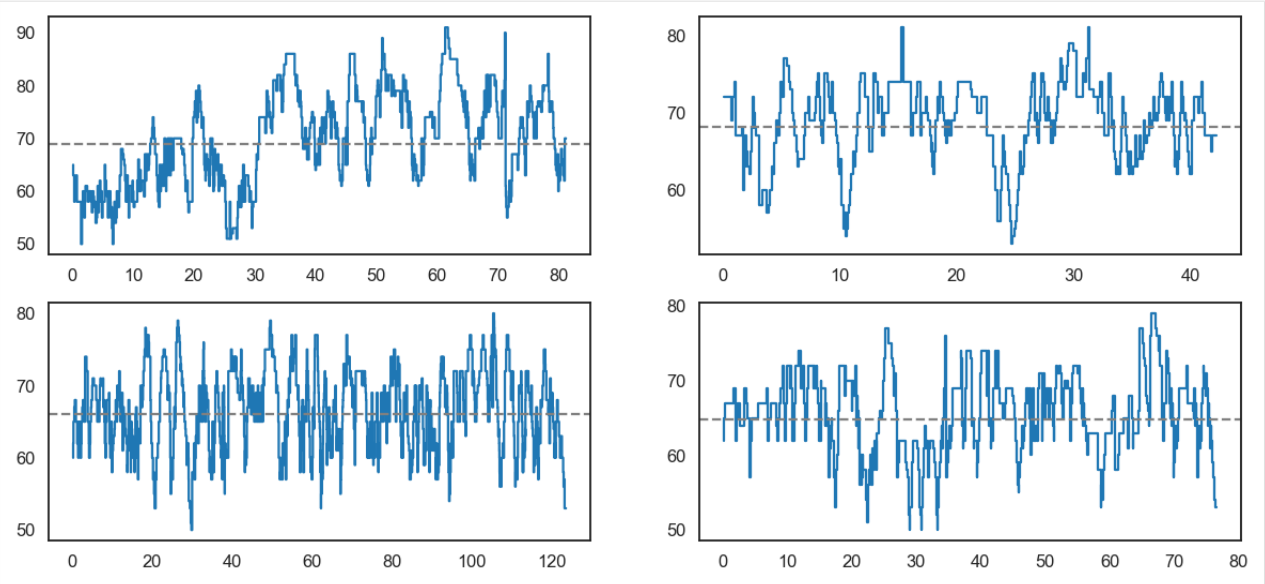
[31]:
```
def notelist(melid):

    solo = solos[solos["melid"] == melid]

    solo = solo[["pitch", "duration"]]
    solo["onset"] = solo["duration"].cumsum()
    return solo
```

[32]:
```
notelist(1)
```

[32]:
```
     pitch  duration      onset
0     65.0  0.138776   0.138776
1     63.0  0.171247   0.310023
2     58.0  0.081270   0.391293
3     61.0  0.235102   0.626395
4     63.0  0.130612   0.757007
..     ...       ...        ...
525   66.0  0.137143  80.645238
526   65.0  0.101587  80.746825
527   63.0  0.104490  80.851315
528   62.0  0.110295  80.961610
529   70.0  0.187211  81.148821

[530 rows x 3 columns]
```

[33]:
```
def plot_melodic_profile(notelist, ax=None, c=None, mean=False, Z=False,␣
↪sections=False, standardized=False):

    if ax == None:
        ax = plt.gca()

    if standardized:
        x = notelist["Rel. Onset"]
        y = notelist["Rel. MIDI Pitch"]
    else:
        x = notelist["onset"]
        y = notelist["pitch"]

    ax.step(x,y, color=c)

    if mean:
        ax.axhline(y.mean(), color="gray", linestyle="--")

    if sections:
        for l in [ x.max() * i for i in [ 1/4, 1/2, 3/4] ]:
            ax.axvline(l, color="gray", linewidth=1, linestyle="--")
```

[34]:
```
fig, axes = plt.subplots(2,2, figsize=(20,9))
axes = axes.flatten()

plot_melodic_profile(notelist(1), ax=axes[0], mean=True)
plot_melodic_profile(notelist(77), ax=axes[1], mean=True)
plot_melodic_profile(notelist(50), ax=axes[2], mean=True)
plot_melodic_profile(notelist(233), ax=axes[3], mean=True)
```



[35]:
```
def standardize(notelist):
    """
    Takes a notelist as input and returns a standardized version.
    """

    notelist["Rel. MIDI Pitch"] = (notelist["pitch"] - notelist["pitch"].mean()) /␣
↪notelist["pitch"].std()
    notelist["Rel. Duration"] = notelist["duration"] / notelist["duration"].sum()
    notelist["Rel. Onset"] = notelist["onset"] / notelist["onset"].max()

    return notelist
```

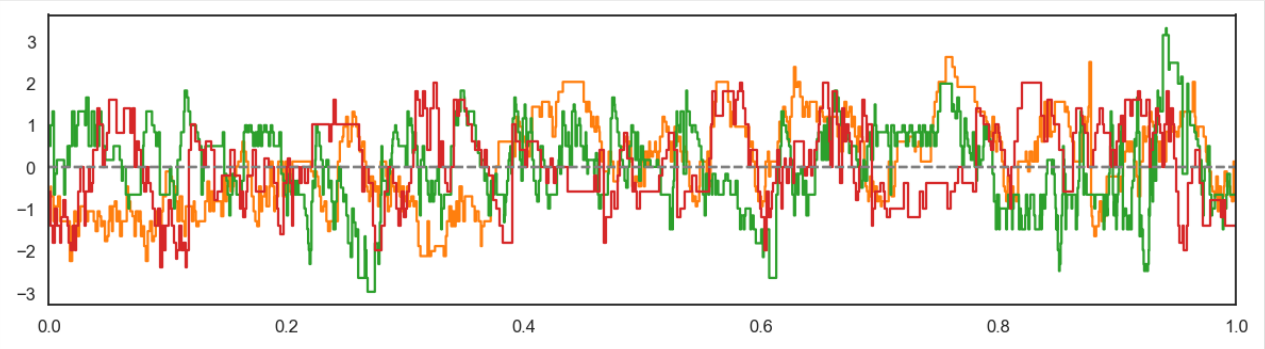
[37]:
```
standardize(notelist(1))
```

[37]:

	pitch	duration	onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset
0	65.0	0.138776	0.138776	-0.460594	0.001710	0.001710
1	63.0	0.171247	0.310023	-0.697714	0.002110	0.003820
2	58.0	0.081270	0.391293	-1.290513	0.001001	0.004822
3	61.0	0.235102	0.626395	-0.934833	0.002897	0.007719
4	63.0	0.130612	0.757007	-0.697714	0.001610	0.009329
..	...	...	...	...	...	...
525	66.0	0.137143	80.645238	-0.342034	0.001690	0.993794
526	65.0	0.101587	80.746825	-0.460594	0.001252	0.995046
527	63.0	0.104490	80.851315	-0.697714	0.001288	0.996334
528	62.0	0.110295	80.961610	-0.816274	0.001359	0.997693
529	70.0	0.187211	81.148821	0.132205	0.002307	1.000000

[530 rows x 6 columns]

[51]:
```
fig, ax = plt.subplots(figsize=(20,5))

for i in range(4):
    plot_melodic_profile(standardize(notelist(i)),
                         mean=True,
                         standardized=True)
plt.xlim(0,1)
plt.show()
```



[41]:
```
big_df = pd.concat([standardize(notelist(i)) for i in range(solos_meta.shape[0])])
```

[44]:
```
solos
```

[44]:

	eventid	melid	onset	pitch	duration	period	division	bar	\
0	1	1	10.343492	65.0	0.138776	4	1	0	
1	2	1	10.637642	63.0	0.171247	4	4	0	
2	3	1	10.843719	58.0	0.081270	4	4	0	
3	4	1	10.948209	61.0	0.235102	4	1	0	
4	5	1	11.232653	63.0	0.130612	4	1	0	
...	...	...	...	...	...	...	...	...	
200804	200805	456	63.135057	57.0	0.168345	4	2	53	
200805	200806	456	63.303401	55.0	0.087075	4	3	54	
200806	200807	456	63.390476	57.0	0.191565	4	3	54	
200807	200808	456	63.640091	59.0	0.406349	4	1	54	
200808	200809	456	64.058050	52.0	1.433832	4	2	54	

	beat	tatum	...	loud_max	loud_med	loud_sd	loud_relpos	\
0	1	1	...	0.126209	66.526087	5.541147	0.307692	
1	2	1	...	0.349751	69.133321	2.912412	0.250000	
2	2	4	...	0.094051	66.352130	3.564563	0.428571	
3	3	1	...	0.521187	66.484173	2.414298	0.818182	
4	4	1	...	0.560737	71.699054	2.185794	0.166667	
...	...	...	...	...	...	...	...	
200804	4	2	...	1.113380	72.169552	6.896394	0.687500	
200805	1	1	...	0.491496	69.732265	1.814723	0.500000	
200806	1	2	...	1.187058	76.628621	2.628726	0.411765	
200807	2	1	...	0.972676	66.042058	3.690577	0.000000	
200808	3	2	...	0.368321	58.174931	9.418678	0.053030	

	loud_cent	loud_s2b	f0_range	f0_freq_hz	f0_med_dev	performer
0	0.389466	1.056169	37.794261	12.932532	-0.328442	Art Pepper
1	0.468687	1.120317	6.365930	6.956935	11.135423	Art Pepper
2	0.531354	1.310389	68.010392	NaN	32.366787	Art Pepper
3	0.559333	0.984047	15.443906	5.867151	-3.374696	Art Pepper
4	0.438973	1.061262	11.444363	8.329975	6.377737	Art Pepper
...	...	...	...	...	...	...
200804	0.581956	1.271747	191.074095	10.966972	-11.891698	Zoot Sims
200805	0.595212	1.339060	40.375449	NaN	-99.173779	Zoot Sims
200806	0.590950	1.432802	104.823845	11.148561	-2.911604	Zoot Sims
200807	0.334937	1.082549	165.810976	2.659723	14.311001	Zoot Sims
200808	0.400571	1.278890	66.932198	2.153916	-9.381310	Zoot Sims

[200809 rows x 27 columns]

[43]:
```
big_df
```

[43]:
```
        pitch  duration      onset  Rel. MIDI Pitch  Rel. Duration  Rel. Onset
0        65.0  0.138776   0.138776        -0.460594       0.001710    0.001710
1        63.0  0.171247   0.310023        -0.697714       0.002110    0.003820
2        58.0  0.081270   0.391293        -1.290513       0.001001    0.004822
3        61.0  0.235102   0.626395        -0.934833       0.002897    0.007719
4        63.0  0.130612   0.757007        -0.697714       0.001610    0.009329
...       ...       ...        ...              ...            ...         ...
200585   62.0  0.870748  68.588934         0.014206       0.012540    0.987794
200586   57.0  0.133515  68.722449        -0.896471       0.001923    0.989717
200587   62.0  0.139320  68.861769         0.014206       0.002006    0.991723
200588   61.0  0.133515  68.995283        -0.167930       0.001923    0.993646
200589   60.0  0.441179  69.436463        -0.350065       0.006354    1.000000

[200590 rows x 6 columns]
```

[62]:
```
solos_meta["performer"].unique()
```

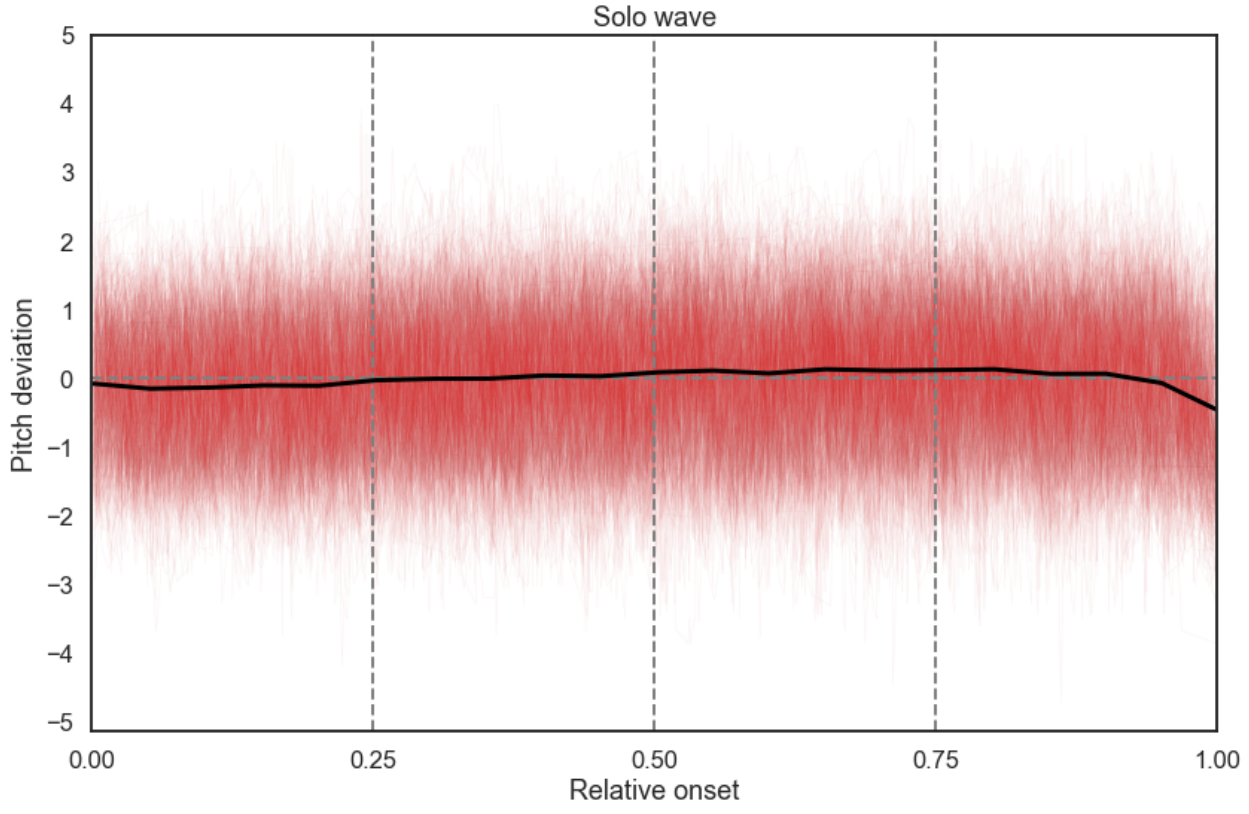
[62]:
```
array(['Art Pepper', 'Benny Carter', 'Benny Goodman', 'Ben Webster',
       'Bix Beiderbecke', 'Bob Berg', 'Branford Marsalis', 'Buck Clayton',
       'Cannonball Adderley', 'Charlie Parker', 'Charlie Shavers',
       'Chet Baker', 'Chris Potter', 'Chu Berry', 'Clifford Brown',
       'Coleman Hawkins', 'Curtis Fuller', 'David Liebman',
       'David Murray', 'Dexter Gordon', 'Dickie Wells', 'Dizzy Gillespie',
       'Don Byas', 'Don Ellis', 'Eric Dolphy', 'Fats Navarro',
       'Freddie Hubbard', 'George Coleman', 'Gerry Mulligan',
       'Hank Mobley', 'Harry Edison', 'Henry Allen', 'Herbie Hancock',
       'J.C. Higginbotham', 'J.J. Johnson', 'Joe Henderson', 'Joe Lovano',
       'John Abercrombie', 'John Coltrane', 'Johnny Dodds',
       'Johnny Hodges', 'Joshua Redman', 'Kai Winding', 'Kenny Dorham',
       'Kenny Garrett', 'Kenny Wheeler', 'Kid Ory', 'Lee Konitz',
       'Lee Morgan', 'Lester Young', 'Lionel Hampton', 'Louis Armstrong',
       'Michael Brecker', 'Miles Davis', 'Milt Jackson', 'Nat Adderley',
       'Ornette Coleman', 'Pat Martino', 'Pat Metheny', 'Paul Desmond',
       'Pepper Adams', 'Phil Woods', 'Red Garland', 'Rex Stewart',
       'Roy Eldridge', 'Sidney Bechet', 'Sonny Rollins', 'Sonny Stitt',
       'Stan Getz', 'Steve Coleman', 'Steve Lacy', 'Steve Turre',
       'Von Freeman', 'Warne Marsh', 'Wayne Shorter', 'Woody Shaw',
       'Wynton Marsalis', 'Zoot Sims'], dtype=object)
```

[67]:
```
%%time

fig, ax = plt.subplots(figsize=(12,8))

artists = ["Louis Armstrong"]

# for i, (artist, group) in enumerate(solos.groupby("performer")):
#     if artist in artists:
#         for j, group in group.groupby("melid"):
#             solo = standardize(notelist(j))
#             x = solo["Rel. Onset"]
#             y = solo["Rel. MIDI Pitch"]
#             ax. plot(x,y, lw=.5, c="tab:red", alpha=.5)

for ID in range(solos_meta.shape[0]):
    solo = standardize(notelist(ID))
    x = solo["Rel. Onset"]
```

(continues on next page)

(continued from previous page)

```
    y = solo["Rel. MIDI Pitch"]
    ax. plot(x,y, lw=.5, c="tab:red", alpha=.05)

ax.axvline(.25, lw=2, ls="--", c="gray")
ax.axvline(.5, lw=2, ls="--", c="gray")
ax.axvline(.75, lw=2, ls="--", c="gray")
ax.axhline(0, lw=2, ls="--", c="gray")

lowess = sm.nonparametric.lowess
big_x = big_df["Rel. Onset"]
big_y = big_df["Rel. MIDI Pitch"]
big_z = lowess(big_y, big_x, frac=1/10, delta=1/20)
ax.plot(big_z[:,0], big_z[:,1], c="black", lw=3)

plt.title("Solo wave")
plt.xlabel("Relative onset")
plt.ylabel("Pitch deviation")
plt.xticks(np.linspace(0,1,5))
plt.yticks(np.linspace(-5,5,11))
plt.xlim(0,1)

plt.tight_layout()
plt.savefig("img/jazz_melodic_arc.png")
plt.show()
```




```
Wall time: 7.76 s
```

4.2 Pitch vs loudness

Above we have already analyzed some melodic profiles and seen that, on average, the Jazz solos tend not to follow the melodic arch on a global scale. Now, we ask whether the pitch of the notes in the solos are related to another important feature of performance: loudness. The WJazzD contains several measures for loudness (compare the columns in the `solos` DataFrame). Here, we focus on the "Median loudness" which is stored in the `loud_med` column.

Let us look at an example.

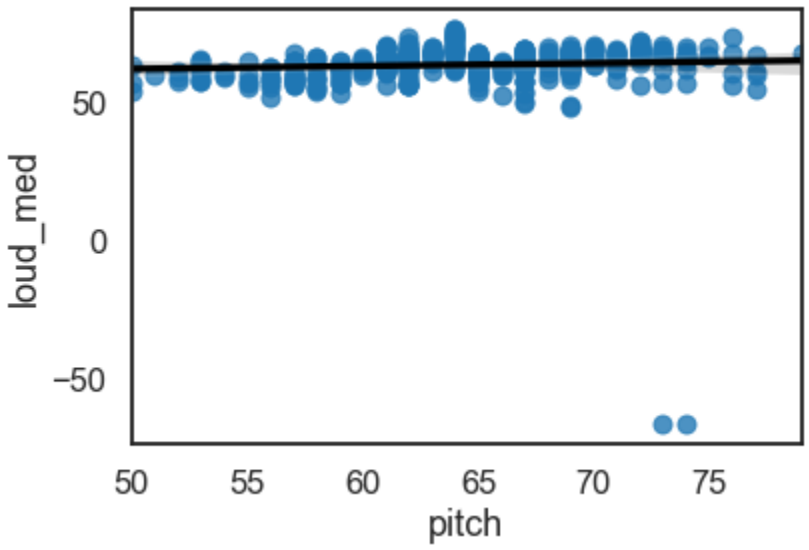
[68]:
```
example_solo = solos[ solos["melid"] == 233 ][["pitch", "loud_med"]]
```

[69]:
```
example_solo
```

[69]:
```
        pitch   loud_med
115075   62.0  67.082700
115076   65.0  65.345677
115077   67.0  66.323539
115078   69.0  69.204257
115079   62.0  69.059581
...       ...        ...
115549   59.0  61.083907
115550   57.0  58.345887
115551   55.0  65.132786
115552   54.0  59.595735
115553   53.0  58.390132

[479 rows x 2 columns]
```

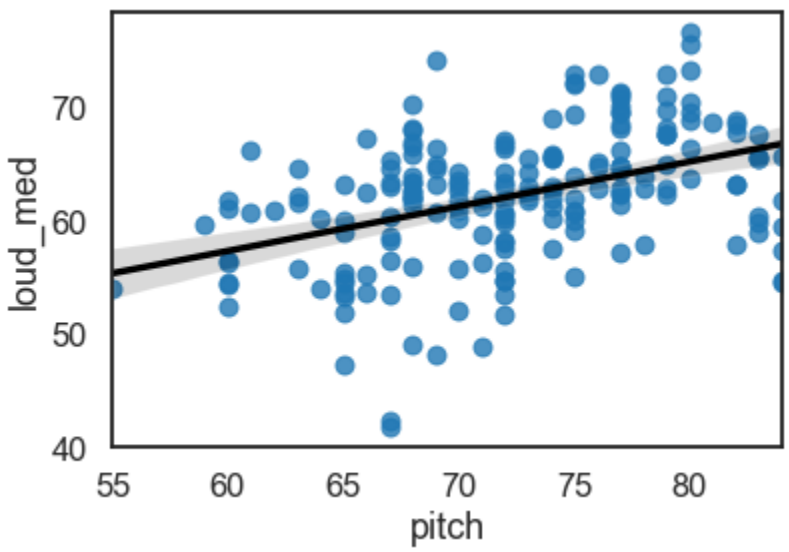
We can get a visual impression of whether there might be a direct relation between the two features by plotting it and drawing a regression line. For this, the `regplot()` function of the `seaborn` library is well-suited.

[71]:
```
sns.regplot(data=example_solo, x="pitch", y="loud_med", line_kws={"color":"black"});
```



There seems to be no clear relation; no matter how high the pitch, the loudness stays more or less the same. Let's look at another example!

[73]:
```
example_solo2 = solos[ solos["melid"] == 333 ][["pitch", "loud_med"]]

sns.regplot(data=example_solo2, x="pitch", y="loud_med", line_kws={"color":"black"});
```

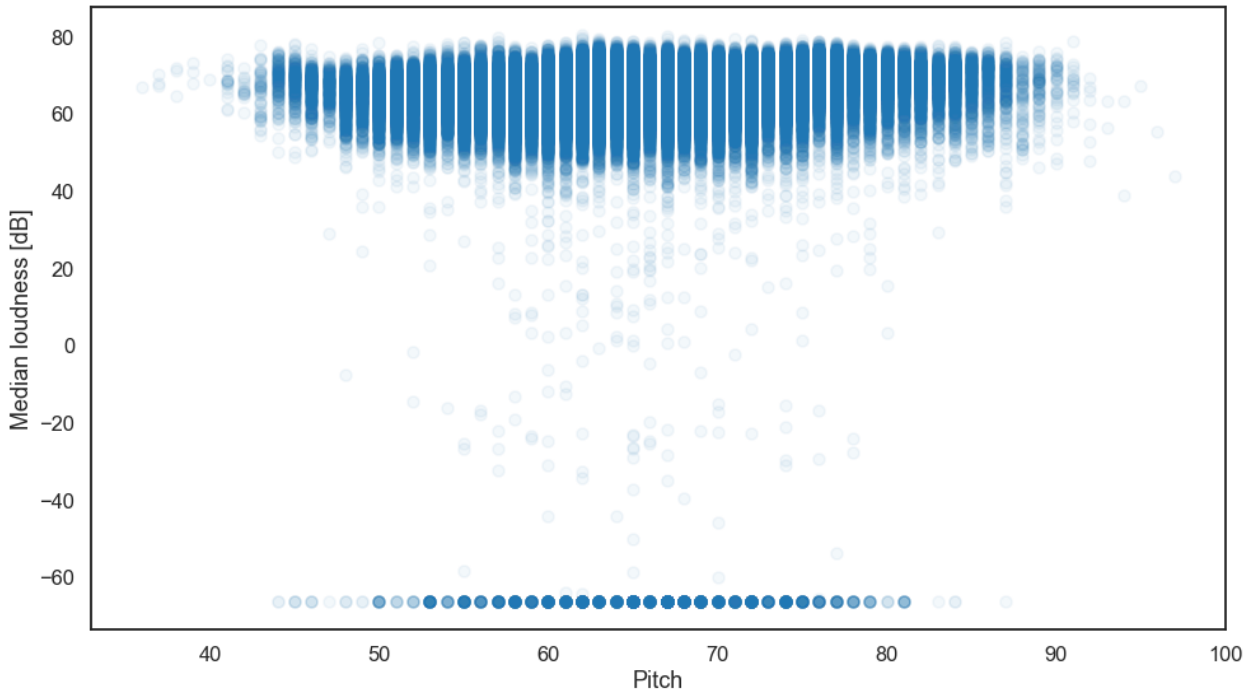


In this case, there is a positive trend. The higher the pitch, the louder the performer plays. Since we have now two different examples - in one case no relation, in the other case a positive correlation - we should now look at whether there is a trend emerging from all solos taken together.

4.3 The "rain cloud" of Jazz solos

We now take all 200'809 notes from all solos and look at the relation between their pitch and their median loudness.

[75]:
```
X = solos[["pitch", "loud_med"]].values
x = X[:,0]
y = X[:,1]

fig, ax = plt.subplots(figsize=(16,9))
ax.scatter(x,y, alpha=0.05)

plt.xlabel("Pitch")
plt.ylabel("Median loudness [dB]")
plt.show()
```



The visual impression is that of a cloud from which rain drops down and forms a puddle. Which trends can we observe?

4.4 Comparing performers

Taking all pieces together was not really informative. Maybe a somewhat closer look brings more to the front. Let us some specific performers whose solos we want to compare.

[76]:
```
selected_performers = ["Charlie Parker", "Miles Davis", "Louis Armstrong", "Herbie␣
↪Hancock", "Von Freeman", "Red Garland"]
```

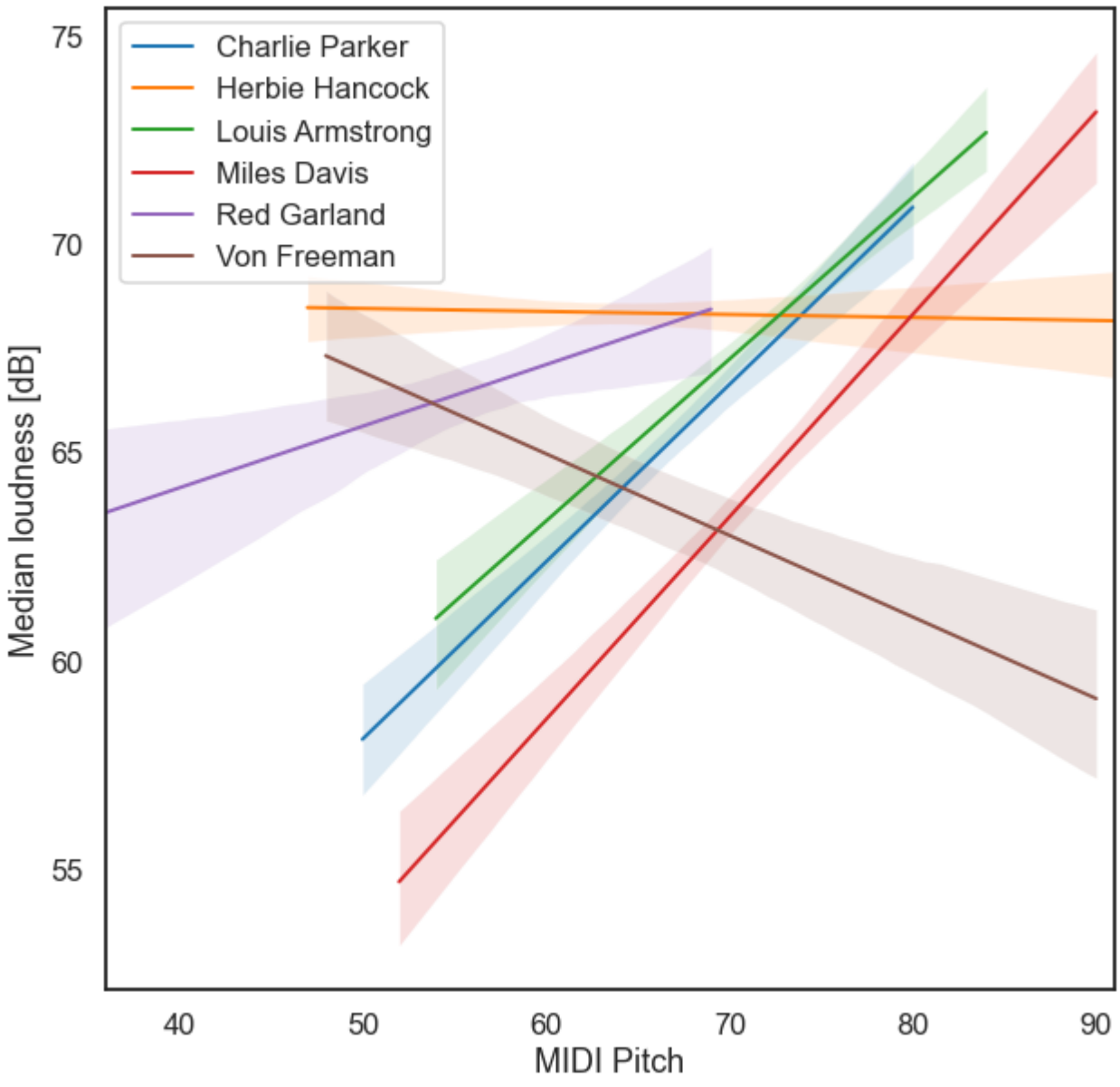
[78]:
```
grouped_df = solos.groupby("performer")
```

[79]:
```
fig, ax = plt.subplots(figsize=(10,10))

for performer, df in grouped_df:
    if performer in selected_performers:
        sns.regplot(
            data=df,
            x="pitch",
            y="loud_med",
            x_jitter=.1,
            y_jitter=.1,
            scatter_kws={"alpha":.01, "color":"grey"},
            line_kws={"lw":2},
            label=performer,
            scatter=False,
            ax=ax
        )
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("MIDI Pitch")
plt.ylabel("Median loudness [dB]")
plt.legend()
plt.show()
```



Observations:

1. Most performers increase loudness with increasing pitch.
2. Charlie Parker (sax) and Louis Armstrong (t) show very similar patterns but Armstrong is generally higher.
3. Miles Davis (t) is similar to the two but plays generally softer than both.
4. Von Freeman (sax) strongly and Herbie Hancock (p) weakly decrease loudness with increasing pitch (almost all other performers show positive correlations).

5. Red Garland (p) plays generally lower than Herbie Hancock (p) but does show a positive correlation between pitch and loudness (NB: there is only one solo in the database).

Does this tell us something about performer styles or about instruments?