INTRODUCTION TO DIGITAL MUSICOLOGY

Fabian C. Moss

2025-07-05

Table of contents

Preface

This page contains material for the course **Introduction to Digital Musicology**, held at Julius-Maximilians-Universität, Würzburg (Germany) in Fall 2025.

⚠ Warning

This is work in progress.

Test citation Knuth (1984).

Part I.

INTRODUCTION

1. What is Digital Musicology?

Introduction and terminology

Goal

Understanding what “digital musicology” means.

- Introduction
- Overview of the field
- Terminology
  - e.g. digital vs computational; the latter in 2nd semester
  - digital vs empirical vs quantiative
  - how does DM relate to “traditional” subdivisions of musicology?

Exercise

Read.

2. Digital Musicology today

Goal

Acquiring and overview of current activities in Digital Musicology.

- Current research topics
- Important institutions and people (also, e.g. NFDI4Culture)
- Central journals and conferences

3. The history of Digital Musicology

Goal

Knowing the beginnings and the major stages of DM.

Part II.

DATA ABOUT MUSIC

4. RISM metadata

Goal

Learn what metadata are and how to search for music sources on RISM Online.

- What is RISM?
- What is RISM Online?

Exercise

Understand basic SPARQL and design queries via prompting.

5. Spotify and MusicBrainz metadata

Goal

Understand the kind of metadata provided by Spotify vs MusicBrainz.

6. Music and the streaming industry

Goal

Gain first insights into the music market and its workings.

Exercise

Work with sales data.

7. Analyzing song survival

In this session, we will analyze songs from the Billboard 100 charts and trace their ‘course of life’ in the charts.

The data was obtained from Kaggle, a large community website for data analysis challenges.

As before, we first import the `pandas` library for data analysis and load the data using the `read_csv` fundtion that takes as its main argument the path to the data file, in our case `charts.csv`.

```python
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("data/charts.csv")
```

Inspecting the first 5 lines with the `.head()` method of pandas DataFrames, we obtain an understanding of the structure of the data.

```python
df.head()
```

	date	rank	song	artist	last-week	peak-rank	weeks-on-bo
0	2021-11-06	1	Easy On Me	Adele	1.0	1	3
1	2021-11-06	2	Stay	The Kid LAROI & Justin Bieber	2.0	1	16
2	2021-11-06	3	Industry Baby	Lil Nas X & Jack Harlow	3.0	1	14
3	2021-11-06	4	Fancy Like	Walker Hayes	4.0	3	19
4	2021-11-06	5	Bad Habits	Ed Sheeran	5.0	2	18

Think: What do the columns represent? Provide verbal descriptions of their meaning and write it down.

After this general overview, we might want to achieve a slightly deeper understanding. For instance, it is not difficult to interpret the `date` column, but from only the first few entries, we cannot know the temporal extend of our data.

Let’s find out what the earliest and latest dates are using the `.min()` and `.max()` methods, respectively.

```python
df["date"].min(), df["date"].max()
```

```
('1958-08-04', '2021-11-06')
```

This tells us that the data stored in `charts.csv` runs from August 1958 to November 2021 and thus allows us to trace the movement of songs in the Billboard charts across more than 60 years.

```python
# Top artists
df.artist.value_counts()
```

```
artist
Taylor Swift                                                 1023
Elton John                                                    889
Madonna                                                       857
Drake                                                         787
Kenny Chesney                                                 769
                                                             ...
YoungBoy Never Broke Again Featuring Sherhonda Gaulden          1
Drake Featuring Chris Brown                                     1
Kehlani Featuring Jhene Aiko                                    1
DaBaby Featuring A Boogie Wit da Hoodie & London On Da Track    1
The Shins                                                       1
Name: count, Length: 10205, dtype: int64
```

```python
# Longest in charts
df.sort_values(by="weeks-on-board", ascending=True).iloc[50_000:]
```

	date	rank	song	artist	last-week	peak-rank	we
213768	1980-11-22	69	Turn And Walk Away	The Babys	79.0	69	2
106440	2001-06-16	41	Fill Me In	Craig David	69.0	41	2
106443	2001-06-16	44	Bootylicious	Destiny's Child	66.0	44	2
106448	2001-06-16	49	All Or Nothing	O-Town	60.0	49	2
213674	1980-11-29	75	My Mother's Eyes	Bette Midler	85.0	75	2
...	...	...	...	...	...	...	...
39148	2014-05-10	49	Radioactive	Imagine Dragons	48.0	3	87
1215	2021-08-14	16	Blinding Lights	The Weeknd	17.0	1	87
1117	2021-08-21	18	Blinding Lights	The Weeknd	16.0	1	88
1020	2021-08-28	21	Blinding Lights	The Weeknd	18.0	1	89
919	2021-09-04	20	Blinding Lights	The Weeknd	21.0	1	90

```
df["date"] = pd.to_datetime(df["date"])
```

```
df[df.artist=="Drake"].song.value_counts()
```

```
song
Hotline Bling     36
God's Plan        36
Controlla         26
Fake Love         25
Nice For What     25
                  ..
Trust Issues       1
Too Much           1
Own It             1
Tuscan Leather     1
Come Thru          1
Name: count, Length: 108, dtype: int64
```

```
df[df.artist=="Elton John"].song.value_counts()
```

```
song
Candle In The Wind 1997/Something About The Way You Look Tonight           42
Can You Feel The Love Tonight (From "The Lion King")                       26
I Guess That's Why They Call It The Blues                                  23
The One                                                                    22
Candle In The Wind                                                         21
Little Jeannie                                                             21
The Last Song                                                              20
Recover Your Soul                                                          20
Believe                                                                    20
Circle Of Life (From "The Lion King")                                      20
Blessed                                                                    20
Sad Songs (say So Much)                                                    19
I Don't Wanna Go On With You Like That                                     18
Nikita                                                                     18
Bennie And The Jets                                                        18
Mama Can't Buy You Love                                                    18
Blue Eyes                                                                  18
You Can Make History (Young Again)                                         17
Sacrifice                                                                  17
Empty Garden (Hey Hey Johnny)                                              17
```

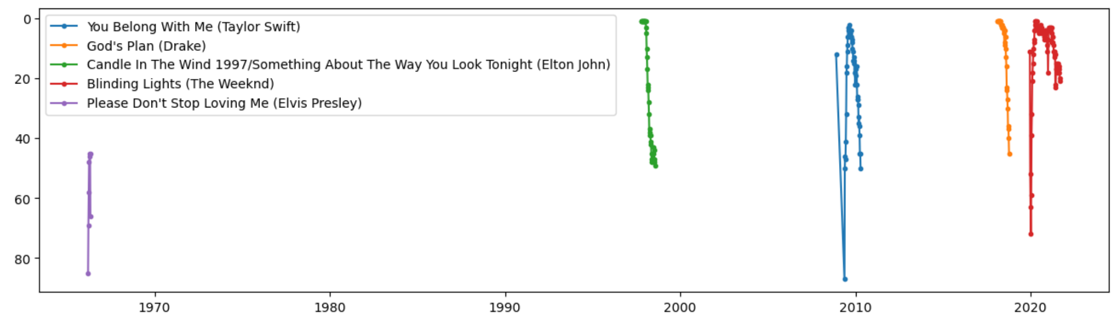
```
Crocodile Rock                                                              17
Goodbye Yellow Brick Road                                                   17
I'm Still Standing                                                          16
Club At The End Of The Street                                               16
Simple Life                                                                 16
Someday Out Of The Blue                                                     15
Don't Let The Sun Go Down On Me                                             15
Island Girl                                                                 15
Daniel                                                                      15
Rocket Man                                                                  15
Healing Hands                                                               15
The Bitch Is Back                                                           14
Who Wears These Shoes?                                                      14
Wrap Her Up                                                                 14
Sorry Seems To Be The Hardest Word                                          14
Lucy In The Sky With Diamonds                                               14
Your Song                                                                   14
Nobody Wins                                                                 13
A Word In Spanish                                                           13
Someone Saved My Life Tonight                                               13
You Gotta Love Someone                                                      13
In Neon                                                                     13
Chloe                                                                       13
(Sartorial Eloquence) Don't Ya Wanna Play This Game No More?                12
Kiss The Bride                                                              12
Saturday Night's Alright For Fighting                                       12
Grow Some Funk Of Your Own/I Feel Like A Bullet (In The Gun Of Robert Ford)   11
Made In England                                                             10
Levon                                                                       10
Victim Of Love                                                              10
Part-Time Love                                                              10
Honky Cat                                                                   10
Friends                                                                      9
Heartache All Over The World                                                 8
Ego                                                                          8
Tiny Dancer                                                                  7
Bite Your Lip (Get up and dance!)                                            6
Border Song                                                                  5
Name: count, dtype: int64
```

```python
def chart_performance(artist, song):
    data = df[(df["artist"] == artist) & (df["song"] == song)]
    data = data.sort_values(by="date").reset_index(drop=True)
    data["date_rel"] = pd.to_timedelta(data["date"] - data["date"][0]).dt.days
    return data
```

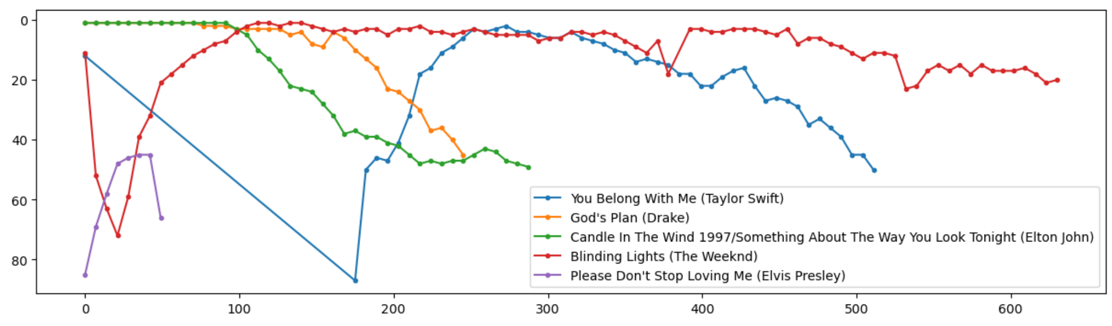
```
test_cases = {
    "Taylor Swift": "You Belong With Me",
    "Drake": "God's Plan",
    "Elton John": "Candle In The Wind 1997/Something About The Way You Look Tonight",
    "The Weeknd": "Blinding Lights",
    "Elvis Presley": "Please Don't Stop Loving Me"
}
```

```
taylor = chart_performance("Taylor Swift", "You Belong With Me")
drake = chart_performance("Drake", "God's Plan")
elton = chart_performance("Elton John", "Candle In The Wind 1997/Something About The Way You I
weeknd = chart_performance("The Weeknd", "Blinding Lights")
elvis = chart_performance("Elvis Presley", "Please Don't Stop Loving Me")
```

```
_, ax = plt.subplots(figsize=(15,4))

for artist, song in test_cases.items():
    data = chart_performance(artist, song)
    x = data["date"].values
    y = data["rank"].values

    ax.plot(x, y, marker=".", label=f"{song} ({artist})")

plt.gca().invert_yaxis()
plt.legend()
plt.show()
```



```
_, ax = plt.subplots(figsize=(15,4))

for artist, song in test_cases.items():
    data = chart_performance(artist, song)
    x = data["date_rel"].values
    y = data["rank"].values

    ax.plot(x, y, marker=".", label=f"{song} ({artist})")

plt.gca().invert_yaxis()
plt.legend()
plt.show()
```



```
# TODO: remove lines for missing weeks (gaps in curves)
# add two cases:
#   - short duration but high peak
#   - long duration but low peak
```

```
# Q: can we predict a song's survival using the features given in the data?
# --> at least introduce notion of training/test data and discuss the epistemologic
# sources for explanation
```

```
# Try other data: https://www.kaggle.com/datasets/thedevastator/billboard-hot-100-a
```

```
df_charts = pd.read_csv("Hot Stuff.csv", index_col=0)
df_charts["WeekID"] = pd.to_datetime(df_charts["WeekID"])
```

```
df_charts.head()
```

index	url	WeekID	Week Position	Song
0	http://www.billboard.com/charts/hot-100/1965-0...	1965-07-17	34	Don't Just Stand Tl
1	http://www.billboard.com/charts/hot-100/1965-0...	1965-07-24	22	Don't Just Stand Tl
2	http://www.billboard.com/charts/hot-100/1965-0...	1965-07-31	14	Don't Just Stand Tl
3	http://www.billboard.com/charts/hot-100/1965-0...	1965-08-07	10	Don't Just Stand Tl
4	http://www.billboard.com/charts/hot-100/1965-0...	1965-08-14	8	Don't Just Stand Tl

```
df_audio = pd.read_csv("Hot 100 Audio Features.csv", index_col=0)
```

```
df_audio.head()
```

index	SongID	Performer	Song
0	-twistin'-White Silver SandsBill Black's Combo	Bill Black's Combo	-twistin'-White Silver Sands
1	¿Dònde Està Santa Claus? (Where Is Santa Claus...	Augie Rios	¿Dònde Està Santa Claus?
2	......And Roses And RosesAndy Williams	Andy Williams	......And Roses And Roses
3	...And Then There Were DrumsSandy Nelson	Sandy Nelson	...And Then There Were Dr
4	...Baby One More TimeBritney Spears	Britney Spears	...Baby One More Time

```
d = df_charts.merge(df_audio)
```

```
d.shape
```

```
(330208, 29)
```

```
d["WeekID"] = pd.to_datetime(d["WeekID"])
```

```
d.sample(10)
```

	url	WeekID	Week Position	Song
275610	http://www.billboard.com/charts/hot-100/1964-0...	1964-06-20	85	My Dreams
189145	http://www.billboard.com/charts/hot-100/2014-0...	2014-02-08	97	Radio
245340	http://www.billboard.com/charts/hot-100/1969-0...	1969-08-02	92	Let's Call It A Day
141579	http://www.billboard.com/charts/hot-100/2009-0...	2009-08-08	3	Knock You Down
117570	http://www.billboard.com/charts/hot-100/1960-0...	1960-02-20	93	Sleepy Lagoon
150750	http://www.billboard.com/charts/hot-100/1966-0...	1966-09-17	36	Flamingo

	url	WeekID	Week Position	Song
23582	http://www.billboard.com/charts/hot-100/1985-1...	1985-10-05	63	Soul K
21493	http://www.billboard.com/charts/hot-100/2003-0...	2003-04-12	21	Rock
71236	http://www.billboard.com/charts/hot-100/2008-1...	2008-12-13	52	My Li
208184	http://www.billboard.com/charts/hot-100/1984-1...	1984-12-15	60	Missin

```
## BOOTSTRAP!

# d = d.sample(500_000, replace=True)
```

```
d.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 330208 entries, 0 to 330207
Data columns (total 29 columns):
 #   Column                      Non-Null Count   Dtype
---  ------                      --------------   -----
 0   url                         330208 non-null  object
 1   WeekID                      330208 non-null  datetime64[ns]
 2   Week Position               330208 non-null  int64
 3   Song                        330208 non-null  object
 4   Performer                   330208 non-null  object
 5   SongID                      330208 non-null  object
 6   Instance                    330208 non-null  int64
 7   Previous Week Position      298048 non-null  float64
 8   Peak Position               330208 non-null  int64
 9   Weeks on Chart              330208 non-null  int64
 10  spotify_genre               315700 non-null  object
 11  spotify_track_id            287066 non-null  object
 12  spotify_track_preview_url   169915 non-null  object
 13  spotify_track_duration_ms   287066 non-null  float64
 14  spotify_track_explicit      287066 non-null  object
 15  spotify_track_album         287004 non-null  object
 16  danceability                286508 non-null  float64
 17  energy                      286508 non-null  float64
 18  key                         286508 non-null  float64
 19  loudness                    286508 non-null  float64
 20  mode                        286508 non-null  float64
 21  speechiness                 286508 non-null  float64
 22  acousticness                286508 non-null  float64
 23  instrumentalness            286508 non-null  float64
 24  liveness                    286508 non-null  float64
 25   valence                     286508 non-null   float64
 26   tempo                       286508 non-null   float64
 27   time_signature              286508 non-null   float64
 28   spotify_track_popularity    287066 non-null   float64
dtypes: datetime64[ns](1), float64(15), int64(4), object(9)
memory usage: 73.1+ MB
```

```
from IPython.display import Audio, HTML
```

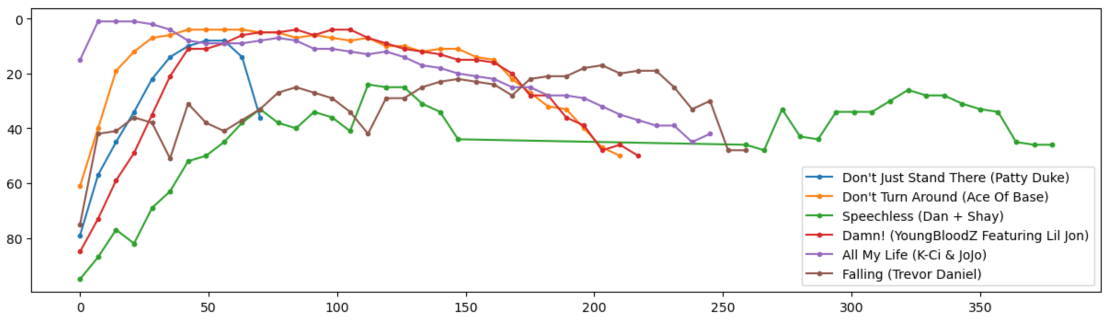
```
Audio(url=d.loc[1000,"spotify_track_preview_url"])
```

```
<IPython.lib.display.Audio object>
```

```
def curves(performer, song):
    data = d[(d.Performer == performer) & (d.Song == song)].sort_values(by="WeekID").reset_inc
    data["date_rel"] = pd.to_timedelta(data["WeekID"] - data["WeekID"][0]).dt.days
    x = data["date_rel"].values # or date_rel or WeekID
    y = data["Week Position"].values
    return x,y
```

```
test_cases2 = {
    "Patty Duke": "Don't Just Stand There",
    "Ace Of Base": "Don't Turn Around",
    "Dan + Shay": "Speechless",
    "YoungBloodZ Featuring Lil Jon": "Damn!",
    "K-Ci & JoJo": "All My Life",
    "Trevor Daniel": "Falling"
}
```

```
_, ax = plt.subplots(figsize=(15,4))

for performer, song in test_cases2.items():
    x,y = curves(performer, song)
    ax.plot(x, y, marker=".", label=f"{song} ({performer})")

plt.gca().invert_yaxis()
plt.legend()
plt.show()
```



Modeling the life of a song in the Top 100:

We assume that once a song has left the Top 100, it is impossible to re-enter (even though that does happen, of course)

1. Each song has a starting rank r_0.
2. For each following week, there is a bernoulli dropout probability θ that determines whether a song remains in the charts.
3.

```
# Observation: Genres tend to leave the Top 100 higher than they entered them
```

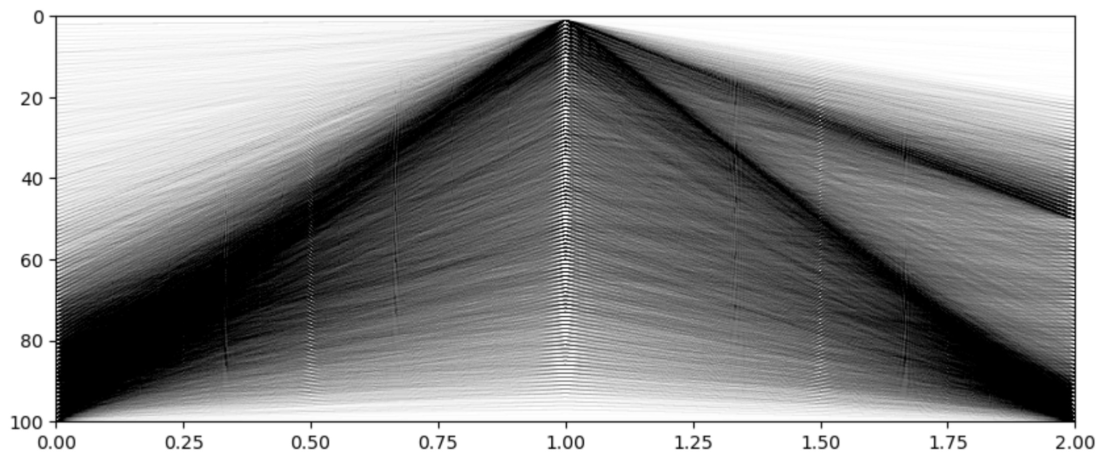
```
entrances = []
peaks = []
exits = []

for _, group in d.groupby("SongID"):
    weeks = group.sort_values(by="WeekID")["Week Position"].values
    entrances.append(weeks[0])
    peaks.append(weeks.min())
    exits.append(weeks[-1])
```

```
import numpy as np
```

```
# from matplotlib.collections import LineCollection
```

```
_, ax = plt.subplots(figsize=(10,4))

K = len(entrances) + 1

for a, b, c in zip(entrances[:K], peaks[:K], exits[:K]):
    if a != b != c: # remove constants
        ax.plot([0, 1, 2], [a, b, c], c="k", lw=.5, alpha=.01)
```

```
plt.xlim(0,2)
plt.ylim(0,100)
plt.gca().invert_yaxis() # smaller is better
plt.savefig("img/rise-decline.png", dpi=600)
plt.show()
```



OBSERVATION: At least 3 types:

- constants
- low in, peak, low out
- low in, peak, mid out

Try to disentangle what causes the difference

Part III.

MUSIC AS DATA

8. Audio

Goal

Understand what an audio signal is and how it is represented digitally .

- Waveform to spectrogram
- Harmonics
- Timbre
- Audible range and volume
- reading melodies from a spectrogram
- digital audio: sampling

9. MIDI

Goal

Be able to name use cases for MIDI. Translate MIDI numbers to pitches.

10. MEI - header

Goal

Understand basic XML encoding and the skeleton structure of MEI.

- mei friend

11. MEI - the body

Goal

Understand the relation between CWMN and the MEI music element.

- MuseScore export
- mei friend

Part IV.

WORKING WITH MUSIC DATA

12. Digital music analysis: harmony

Goal

Understand what labeling is and why labels can be useful.

- further MuseScore practice
- segmentation and labeling
- Counting chords, finding cadences

13. Digital music analysis: melody

Goal

Understand how melodic pattern matching works in principle.

- Pattern finding in melodies (Non-Western)

14. Melodies in Folk Songs

On Jupyter Hub, change the kernel to Python 3.7!

```
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")
```

```
## 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
```

14.1. 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`.

```
# 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`.

```
len(corpora)
```

```
31
```

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

```
corpora
```

```
[WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenF
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/kir
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/lot
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/lux
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/tes
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/tes
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/tes
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/tes
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/var
 WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/zuc
```

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:

```
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.

```
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.

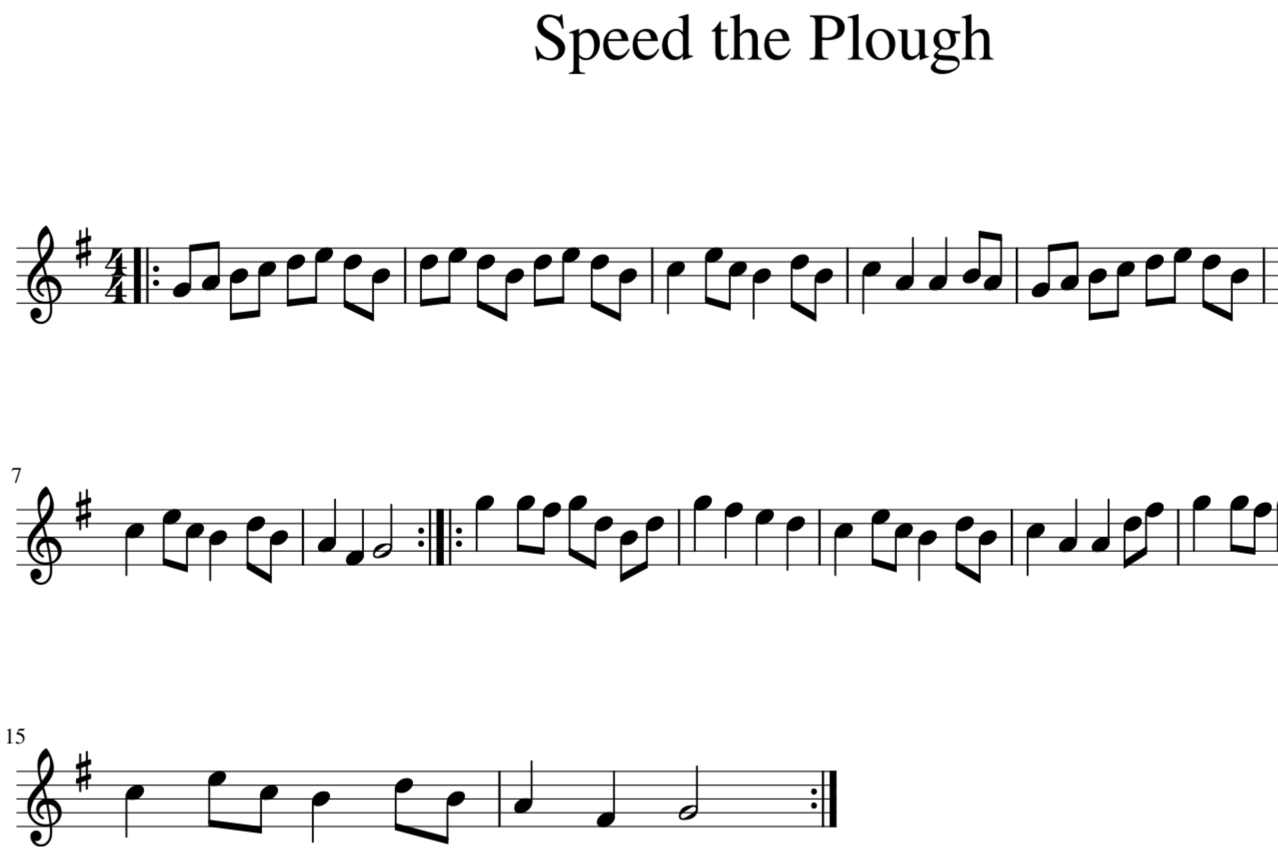
Calling the variable `parsed_example_song` now, however, does not really help us here…

```
parsed_example_song
```

```
<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()`.

```
parsed_example_song.show()
```



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

```
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:

```
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

```
corpora[0]
```

```
WindowsPath('C:/Users/fabianmoss/anaconda3/Lib/site-packages/music21/corpus/essenFolksong/altd
```

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

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

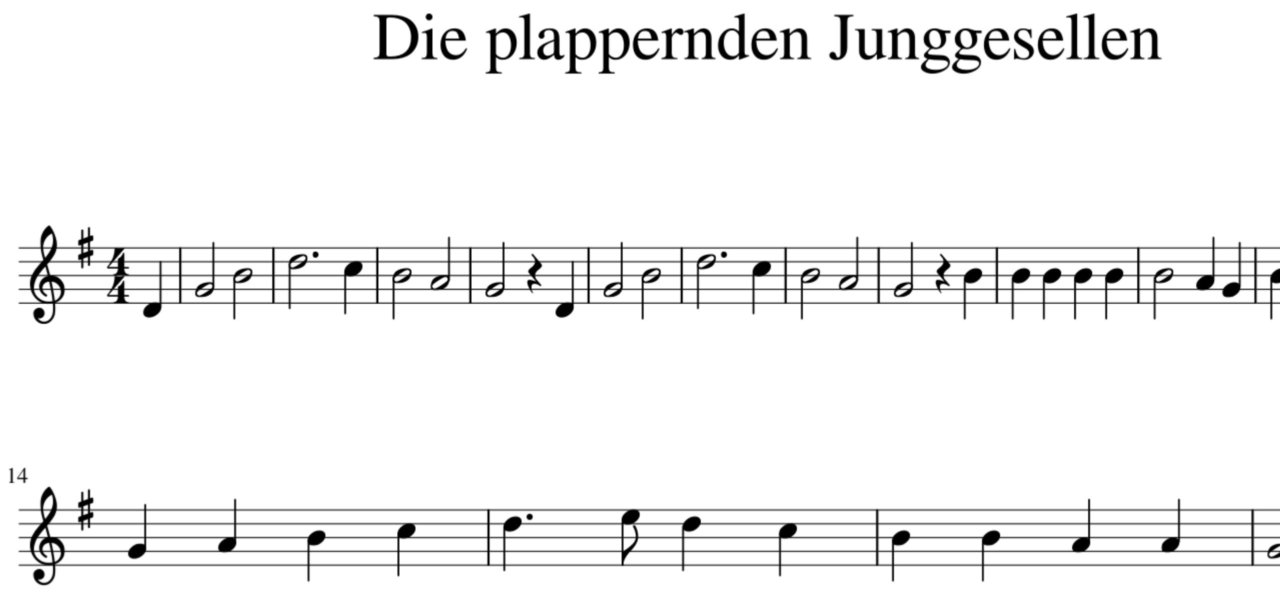
Looking at the new variable `first_corpus` shows a difference to the example song before; we don't have a `music21.stream.Score` object but a `music21.stream.Opus` object.

```
first_corpus
```

```
<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.

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



A A B A'

```
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.

14.2. 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).

```
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.

```
len(songs)
```

```
8514
```

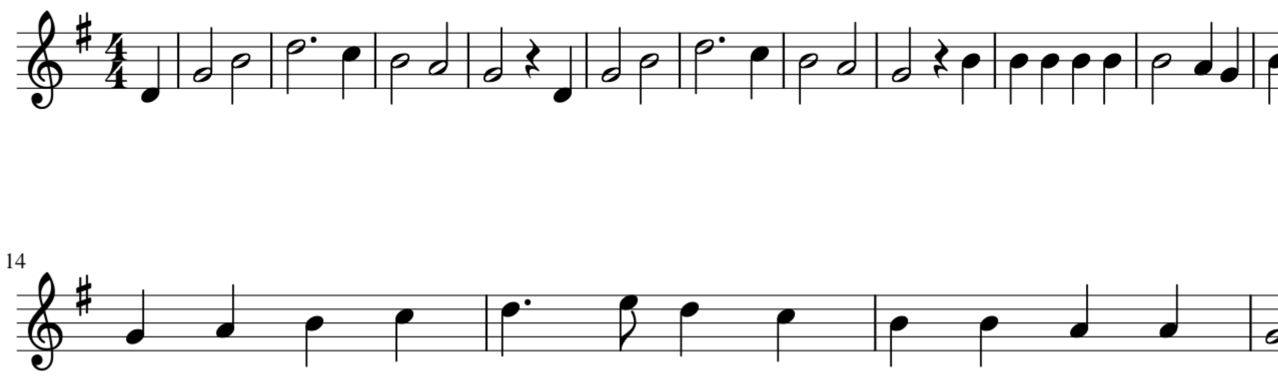
We can now use the list `songs` to compare two different songs. Again, we load the 71st song of the first corpus and store it now in a variable `german_song`, and we load chinese song with index 6200 into the variable `chinese_song`.

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

It is easy to display these songs now:

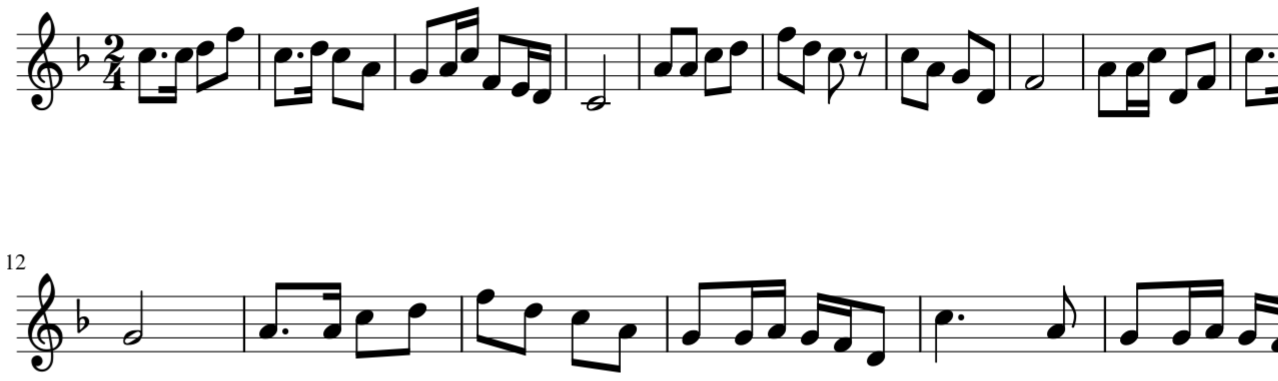
```
german_song.show()
```

Die plappernden Junggesellen



```
chinese_song.show()
```

Shengsi liangxianglian



```
chinese_song.show("midi")
```

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

Analysis of songs…

14.3. 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:

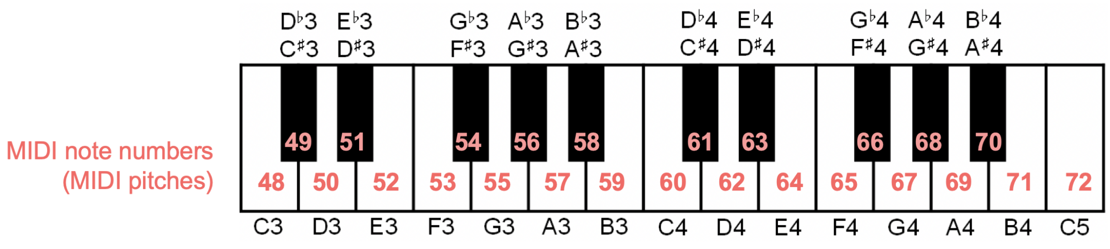

D♭3 E♭3 G♭3 A♭3 B♭3 D♭4 E♭4 G♭4 A♭4 B♭4
C♯3 D♯3 F♯3 G♯3 A♯3 C♯4 D♯4 F♯4 G♯4 A♯4
MIDI note numbers
(MIDI pitches)
49 51 54 56 58 61 63 66 68 70
48 50 52 53 55 57 59 60 62 64 65 67 69 71 72
C3 D3 E3 F3 G3 A3 B3 C4 D4 E4 F4 G4 A4 B4 C5

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.

```
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.
    """

    df = pd.DataFrame([ (note.pitch.midi, note.quarterLength) for note in piece.flat.notes ],
    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:

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

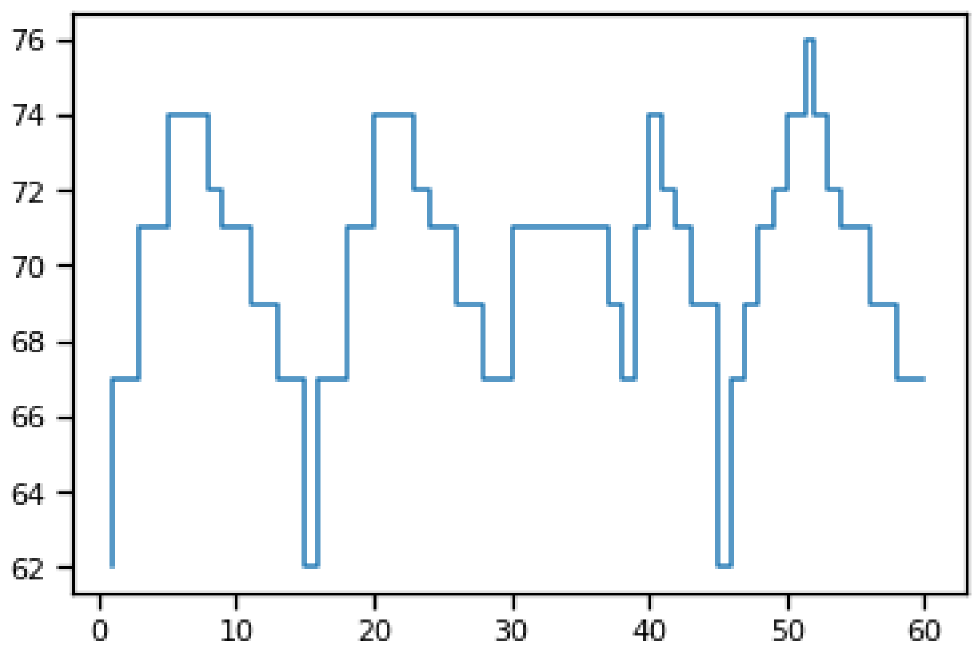
	MIDI Pitch	Duration	Onset
0	62	1.0	1.0
1	67	2.0	3.0
2	71	2.0	5.0
3	74	3.0	8.0
4	72	1.0	9.0
5	71	2.0	11.0
6	69	2.0	13.0
7	67	2.0	15.0

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.

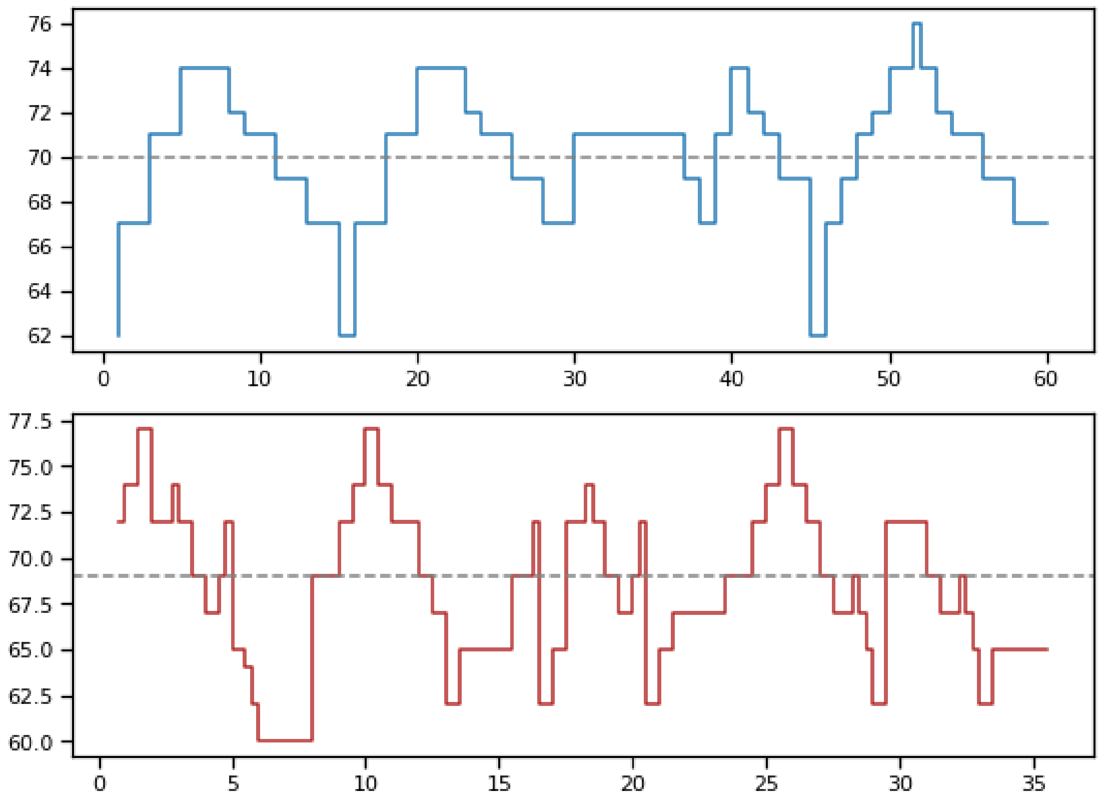
```
def plot_melodic_profile(notelist, ax=None, c=None, mean=False, Z=False, sections=F

    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="--")
```

```
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).

```
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).

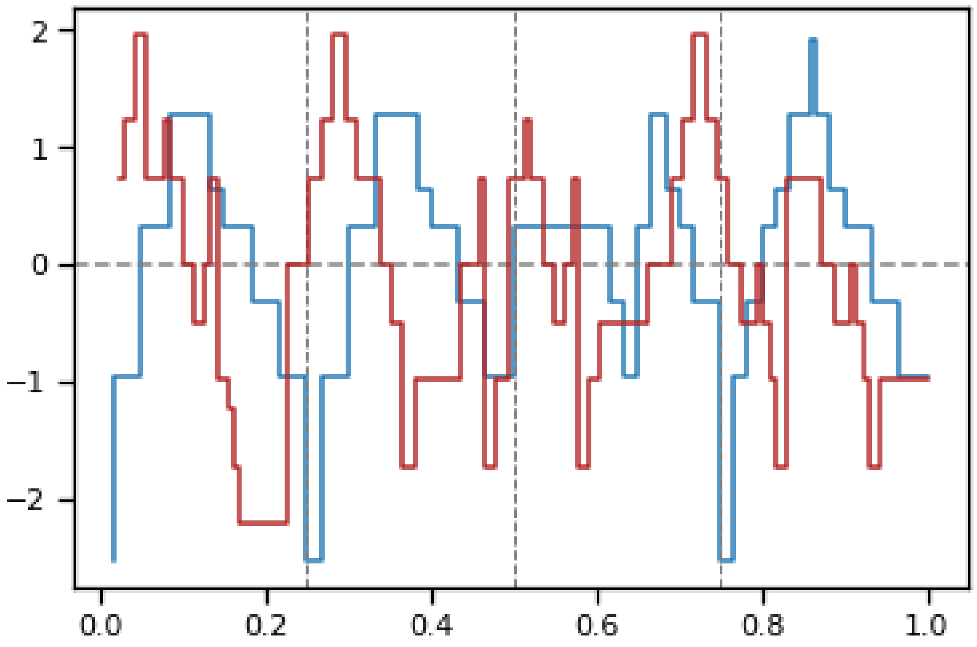
We need to transform them in a way that makes them directly comparable. To that end, we define a new function `standardize()`.

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

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

    return notelist
```

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

	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

```
plot_melodic_profile(standardize(notelist(german_song)), mean=True, sections=True, standardize
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.

```
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)
```

```
big_df
```

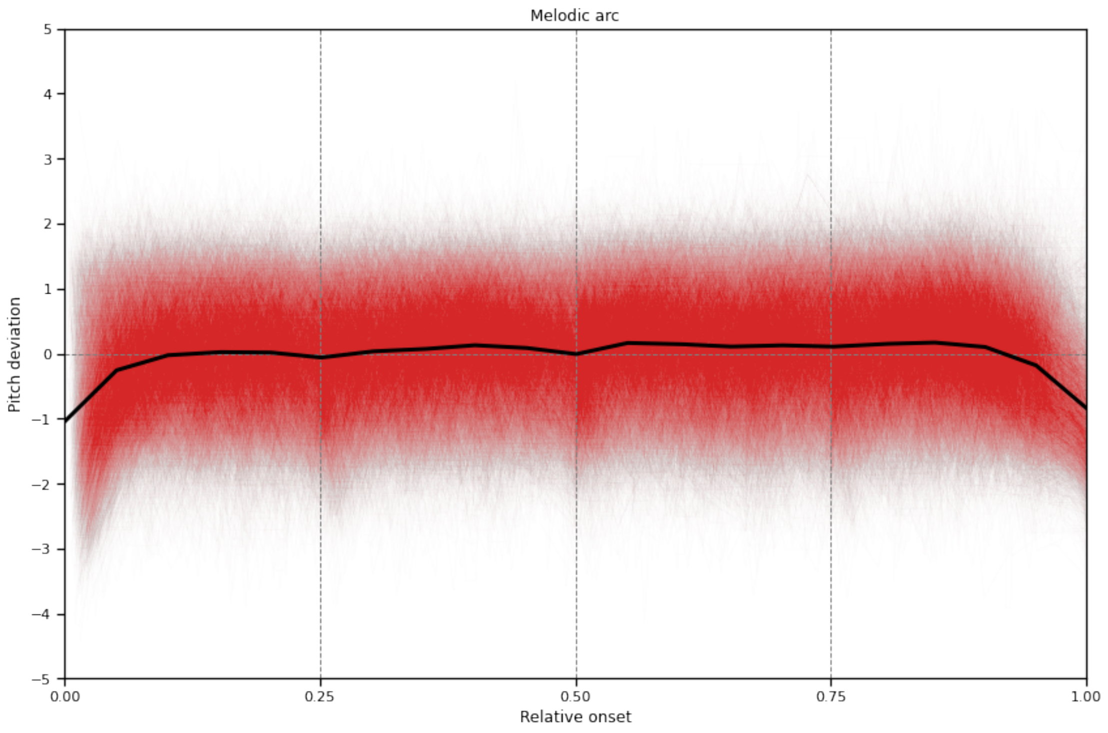
	MIDI Pitch	Duration	Onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset	Song ID
0	67	2.00	2.00	-1.819039	0.013158	0.013158	0
1	70	2.00	4.00	-0.741977	0.013158	0.026316	0
2	71	2.00	6.00	-0.382956	0.013158	0.039474	0
3	72	2.00	8.00	-0.023935	0.013158	0.052632	0
4	72	2.00	10.00	-0.023935	0.013158	0.065789	0
...	...	...	...	...	...	...	...
450591	71	0.25	28.50	0.691456	0.008197	0.934426	8513
450592	69	0.25	28.75	0.098779	0.008197	0.942623	8513
450593	73	0.25	29.00	1.284133	0.008197	0.950820	8513
450594	71	1.00	30.00	0.691456	0.032787	0.983607	8513
450595	69	0.50	30.50	0.098779	0.016393	1.000000	8513

```
big_df.to_csv("data/big_df.csv") # comma-separated values
```

```
big_df = pd.read_csv("data/big_df.csv")
```

14.4. The melodic arc

```
%%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"]
    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()
```



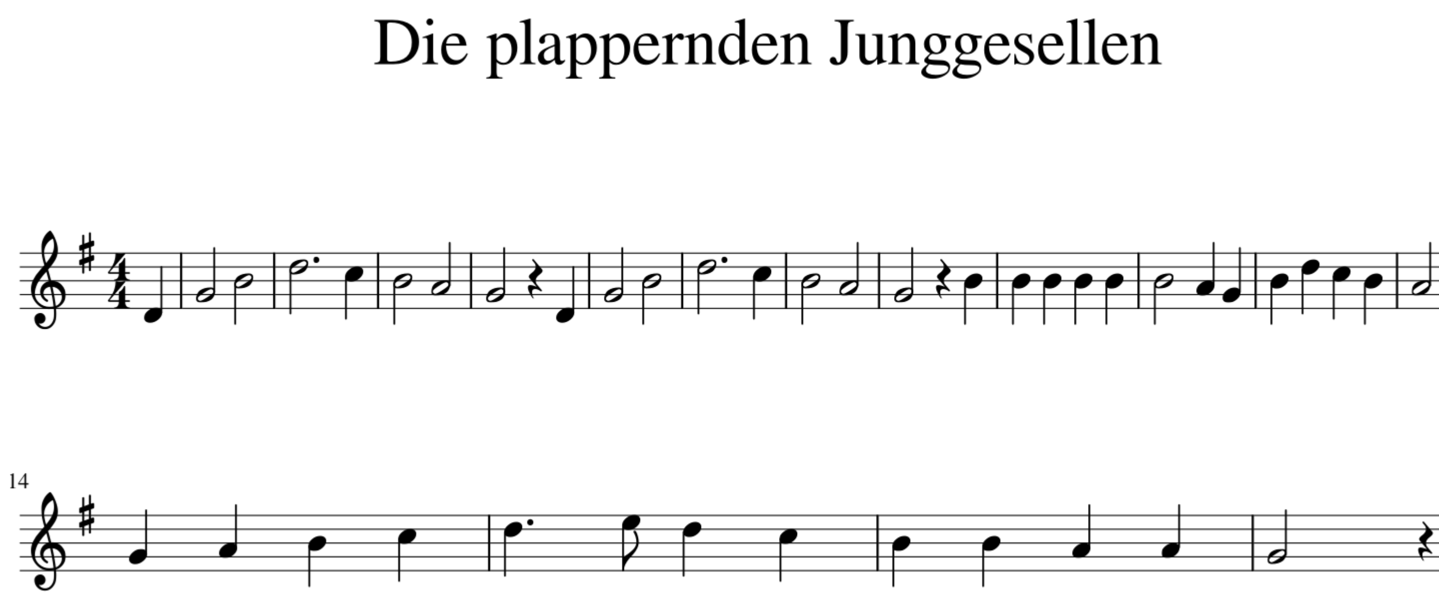
```
Wall time: 24.1 s
```

14.5. 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*

```
german_song.show()
```



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

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

	Unnamed: 0	Unnamed: 0.1	MIDI Pitch	Duration	Onset	Rel. MIDI Pitch	Rel. Duration	Rel.
2969	2969	2969	62	1.0	1.0	-2.543827	0.016667	0.01
2970	2970	2970	67	2.0	3.0	-0.949300	0.033333	0.05
2971	2971	2971	71	2.0	5.0	0.326322	0.033333	0.08
2972	2972	2972	74	3.0	8.0	1.283038	0.050000	0.13
2973	2973	2973	72	1.0	9.0	0.645227	0.016667	0.15
2974	2974	2974	71	2.0	11.0	0.326322	0.033333	0.18
2975	2975	2975	69	2.0	13.0	-0.311489	0.033333	0.21
2976	2976	2976	67	2.0	15.0	-0.949300	0.033333	0.25

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.

```
all_ints = [ p2 - p1 for i, g in big_df.groupby("Song ID") for p1, p2 in zip(g["MID
min_int = min(all_ints)
max_int = max(all_ints)
```

```
min_int, max_int
```

```
(-25, 25)
```

```
len(all_ints)
```

```
442082
```

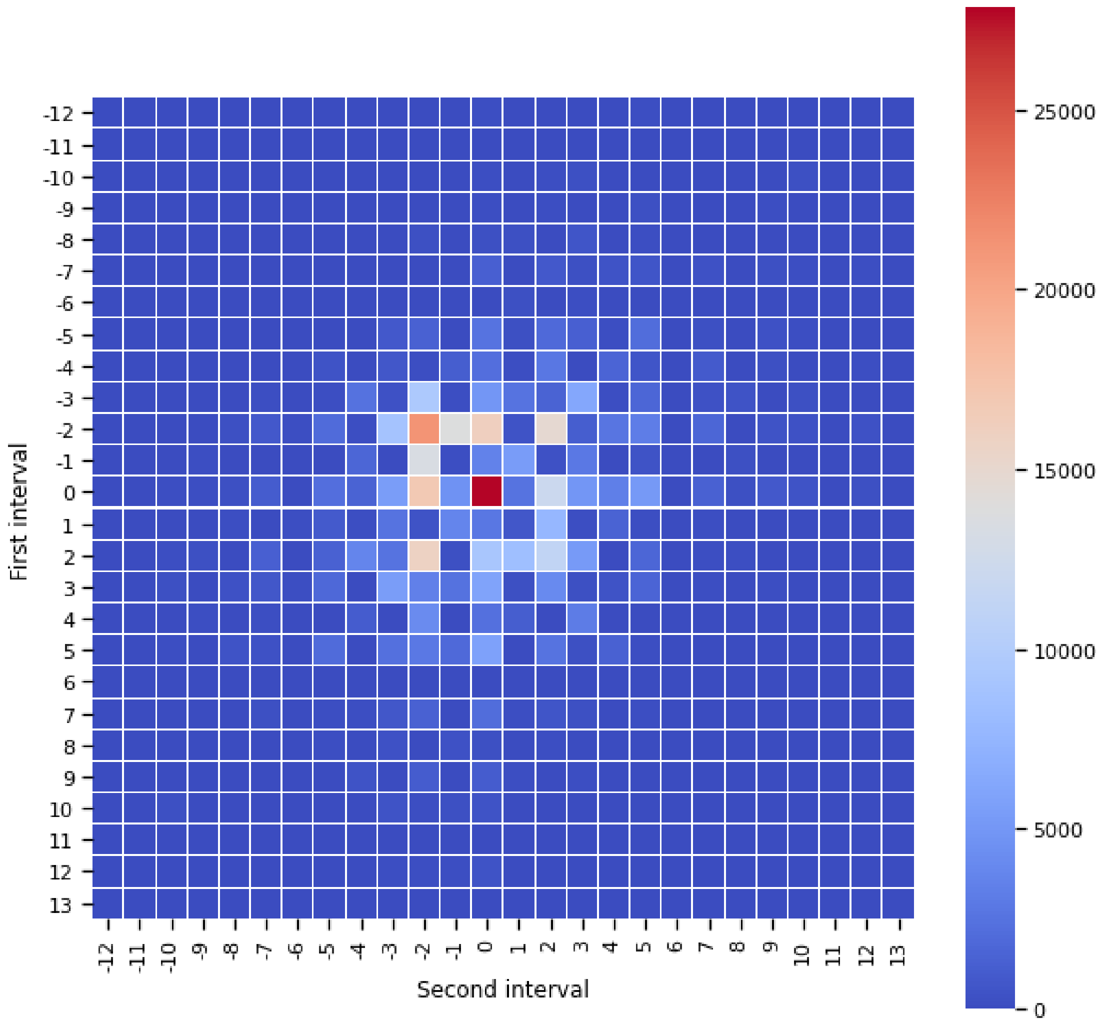
```
ints_df = pd.DataFrame(0, index=np.arange(min_int,max_int), columns=np.arange(min_i

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
```

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

	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	...	1	2	3
-10	0	0	0	0	0	2	3	2	34	0	...	16	174	219
-9	0	0	0	6	0	3	0	39	10	31	...	5	159	39
-8	0	1	0	1	0	0	19	0	319	5	...	302	66	605
-7	0	0	2	14	0	37	2	91	96	103	...	17	859	300
-6	0	0	0	0	0	0	12	11	8	21	...	274	7	132
-5	1	0	1	49	0	75	32	866	1361	25	...	230	1906	1272
-4	3	0	27	11	5	461	18	692	167	1099	...	129	2738	40
-3	1	91	0	192	2	215	2490	416	9478	134	...	2547	1467	6274

	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	...	1	2	3	4	5
-2	67	14	260	858	132	1964	113	8871	21285	13896	...	454	14883	1115	2616	3216
-1	5	174	4	68	47	32	1679	91	13445	205	...	5410	396	2878	58	477
0	186	130	310	1022	80	2270	1440	5506	16887	4603	...	2578	12142	4947	3340	5203
1	55	6	174	43	110	944	165	2564	501	3765	...	747	7649	193	1470	132
2	138	294	29	1288	15	1361	3754	2562	15795	254	...	8404	11261	5249	86	1695
3	272	23	373	655	122	1755	24	5509	3397	2400	...	295	4128	231	519	1523
4	5	164	3	130	11	51	1026	64	4217	60	...	1133	67	3153	20	102
5	116	23	505	330	13	1996	82	2375	2834	1868	...	102	2557	270	1355	160
6	2	4	1	4	23	1	14	18	44	34	...	42	8	20	0	0
7	31	18	11	273	3	167	128	665	1338	59	...	119	566	249	12	166
8	47	1	88	7	19	149	4	370	33	384	...	23	159	0	23	6
9	1	61	1	20	3	20	442	18	1024	4	...	122	13	129	1	8
10	267	0	56	23	28	58	0	517	163	295	...	3	78	0	4	5

```python
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?

```python
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
```

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

```python
big_df.tail()
```

	MIDI Pitch	Duration	Onset	Rel. MIDI Pitch	Rel. Duration	Rel. Onset	Song ID	Avg. MID
450591	71	0.25	28.50	0.691456	0.008197	0.934426	8513	68
450592	69	0.25	28.75	0.098779	0.008197	0.942623	8513	68
450593	73	0.25	29.00	1.284133	0.008197	0.950820	8513	68
450594	71	1.00	30.00	0.691456	0.032787	0.983607	8513	68
450595	69	0.50	30.50	0.098779	0.016393	1.000000	8513	68

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

```
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])
```

```python
transitions_df = pd.DataFrame(0, index=idx, columns=idx)
transitions_df
```

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

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

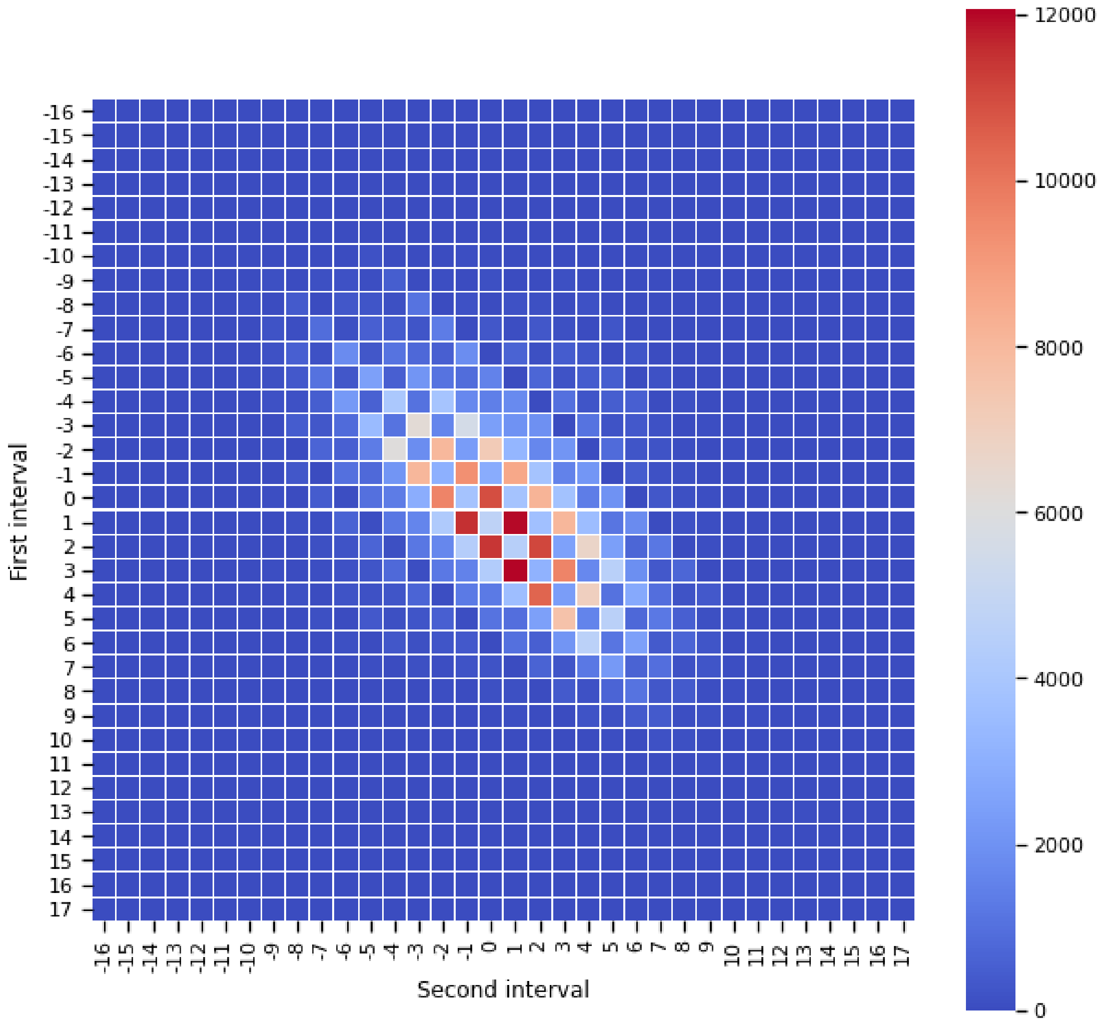
```
%%time

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
```

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

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

```
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()
```



14.6. n-grams

- n-gram viewer [https://www.peachnote.com/#!nt=singleNoteAffine&npq=62+0+1+2] :cite:`Viro2011`
- theory on n-grams

15. 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).

```python
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.

```python
conn = sqlite3.connect("data/wjazzd.db")
```

```python
conn
```

```
<sqlite3.Connection at 0x201747b7990>
```

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

```python
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:

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

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

```
solos_meta.shape
```

```
(456, 17)
```

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

```
solos_meta.sample(5)
```

	melid	trackid	compid	recordid	performer	title	titleaddon	solopart	i
429	430	260	236	140	Wayne Shorter	Eighty-One		1	t
440	441	335	295	180	Woody Shaw	Rosewood		1	t
240	241	152	137	76	Johnny Hodges	Bunny		1	a
326	327	174	158	87	Miles Davis	Walkin'		1	t
224	225	199	177	103	John Coltrane	Impressions	1963	1	t

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

```
solos.head()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	…	f0_
0	1	1	10.343492	65.0	0.138776	4	1	0	1	1	…	
1	2	1	10.637642	63.0	0.171247	4	4	0	2	1	…	
2	3	1	10.843719	58.0	0.081270	4	4	0	2	4	…	
3	4	1	10.948209	61.0	0.235102	4	1	0	3	1	…	
4	5	1	11.232653	63.0	0.130612	4	1	0	4	1	…	

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

```
solos.tail()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	f0_mod
200804	200805	456	63.135057	57.0	0.168345	4	2	53	4	2	...	
200805	200806	456	63.303401	55.0	0.087075	4	3	54	1	1	...	
200806	200807	456	63.390476	57.0	0.191565	4	3	54	1	2	...	slide
200807	200808	456	63.640091	59.0	0.406349	4	1	54	2	1	...	
200808	200809	456	64.058050	52.0	1.433832	4	2	54	3	2	...	vibrato

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

```
solos.shape
```

```
(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.

```
solos.columns
```

```
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.

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

```
{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',
 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',
 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',
 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',
 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',
 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',
 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',
 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',
 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.

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

```
solos.head()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	lou
0	1	1	10.343492	65.0	0.138776	4	1	0	1	1	...	0.1
1	2	1	10.637642	63.0	0.171247	4	4	0	2	1	...	0.3
2	3	1	10.843719	58.0	0.081270	4	4	0	2	4	...	0.0
3	4	1	10.948209	61.0	0.235102	4	1	0	3	1	...	0.5
4	5	1	11.232653	63.0	0.130612	4	1	0	4	1	...	0.5

```
solos.tail()
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	..
200804	200805	456	63.135057	57.0	0.168345	4	2	53	4	2	..
200805	200806	456	63.303401	55.0	0.087075	4	3	54	1	1	..
200806	200807	456	63.390476	57.0	0.191565	4	3	54	1	2	..
200807	200808	456	63.640091	59.0	0.406349	4	1	54	2	1	..
200808	200809	456	64.058050	52.0	1.433832	4	2	54	3	2	..

15.1. Melodic arc?

Does the melodic arc also appear in the Jazz solos?

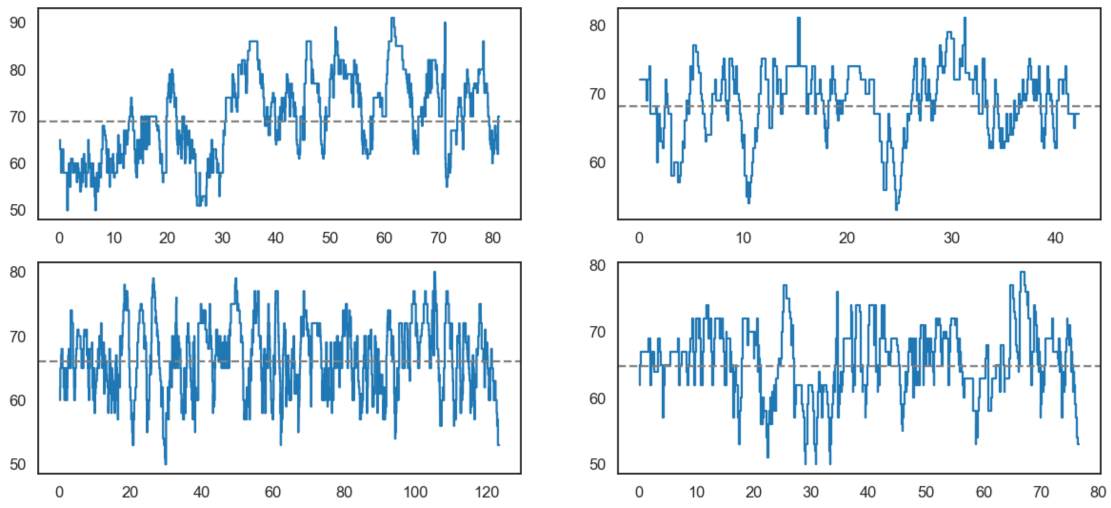
```
def notelist(melid):

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

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

```
notelist(1)
```

	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

```
def plot_melodic_profile(notelist, ax=None, c=None, mean=False, Z=False, sections=False, stand

    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="--")
```

```
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)
```

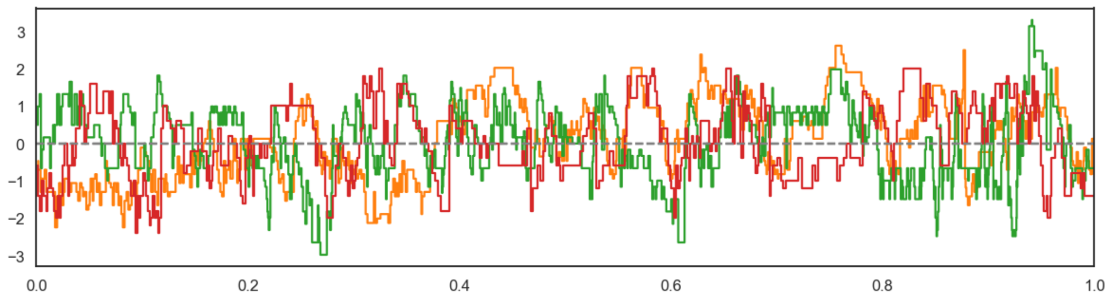


```
def standardize(notelist):
    """
    Takes a notelist as input and returns a standardized version.
    """

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

    return notelist
```

```
standardize(notelist(1))
```

	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

```
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()
```



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

```
solos
```

	eventid	melid	onset	pitch	duration	period	division	bar	beat	tatum	...	loud_ma
0	1	1	10.343492	65.0	0.138776	4	1	0	1	1	...	0.126209

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

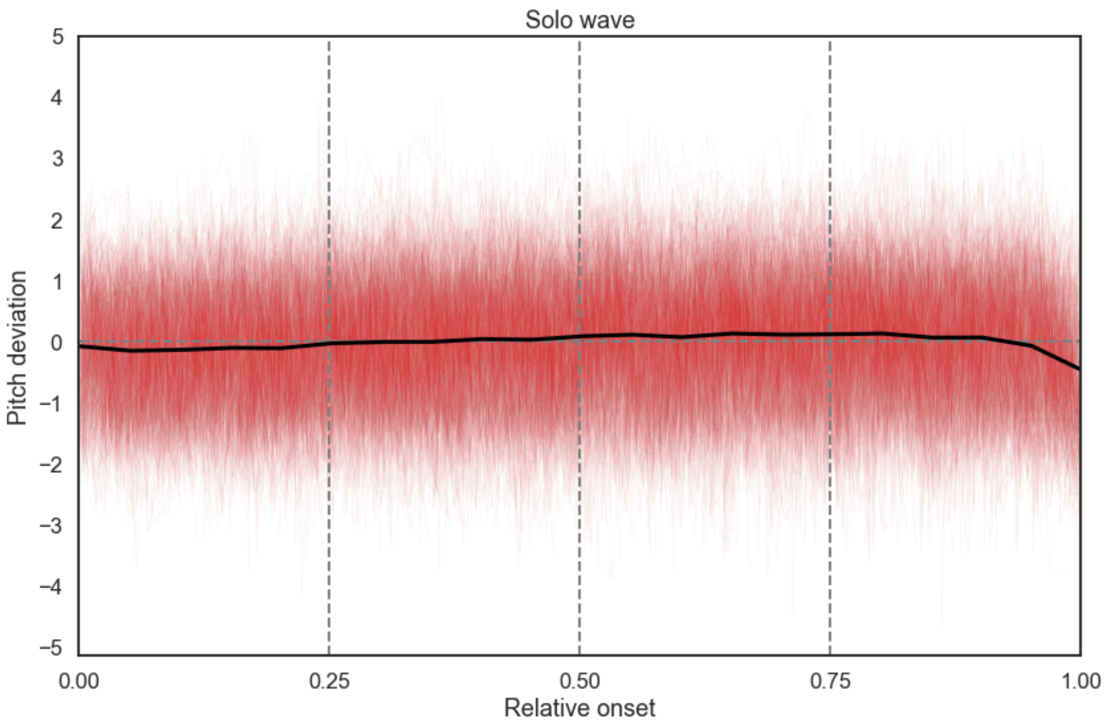
```
big_df
```

	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

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

```
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)
```

```python
%%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"]
    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
```

15.2. 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.

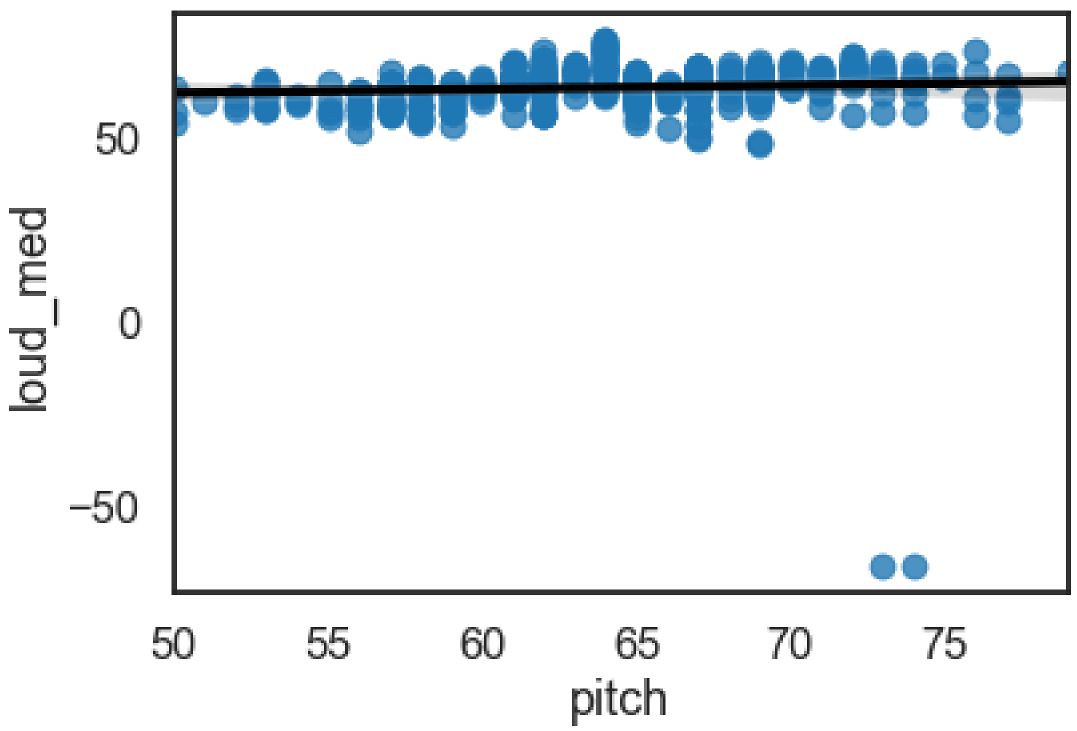
```
example_solo = solos[ solos["melid"] == 233 ][["pitch", "loud_med"]]
```

```
example_solo
```

	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

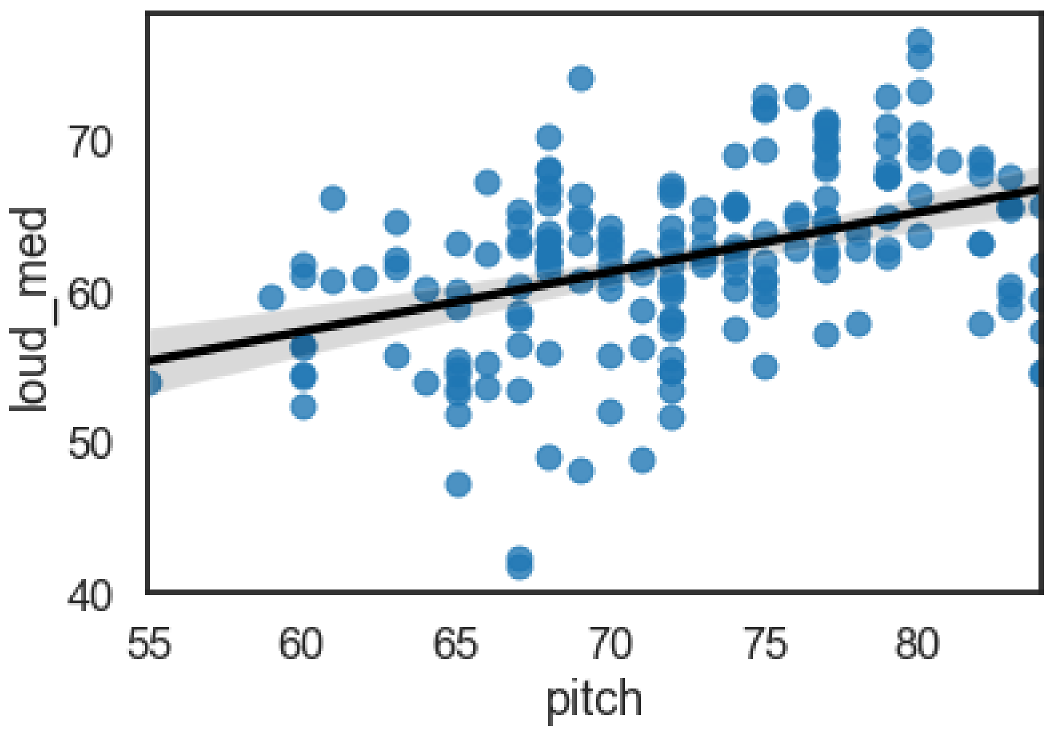
We can get a visual impression of whether there might be a direct relation between the two features by plotting it and drawing a regression line. For this, the `regplot()` function of the `seaborn` library is well-suited.

```
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!

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

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

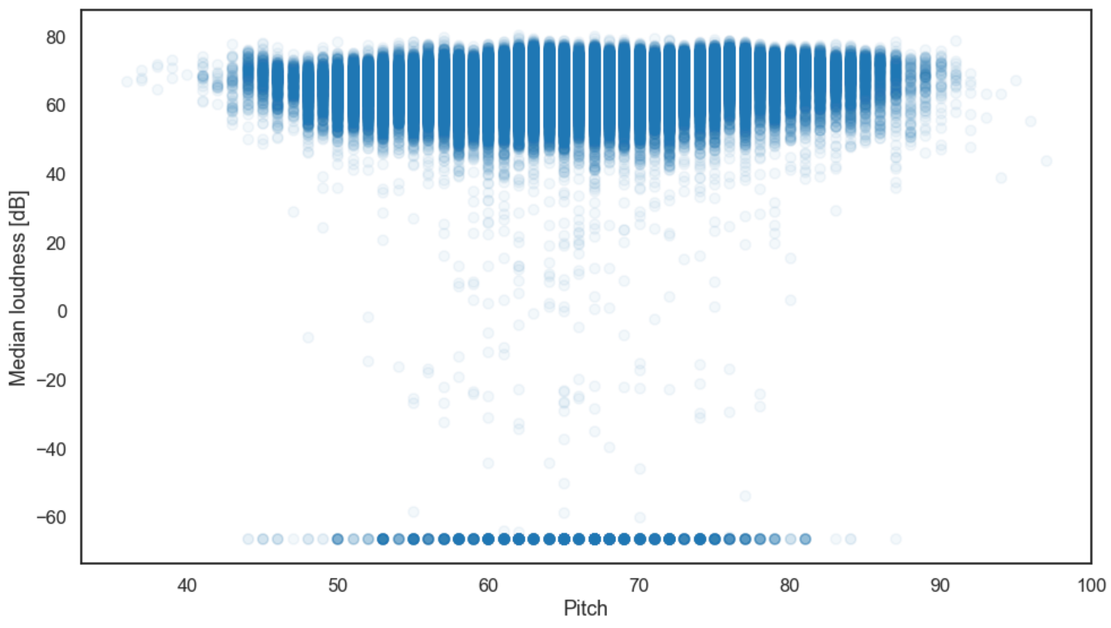


In this case, there is a positive trend. The higher the pitch, the louder the performer plays. Since we have now two different examples - in one case no relation, in the other case a positive correlation - we should now look at whether there is a trend emerging from all solos taken together.

15.3. 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.

```
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?

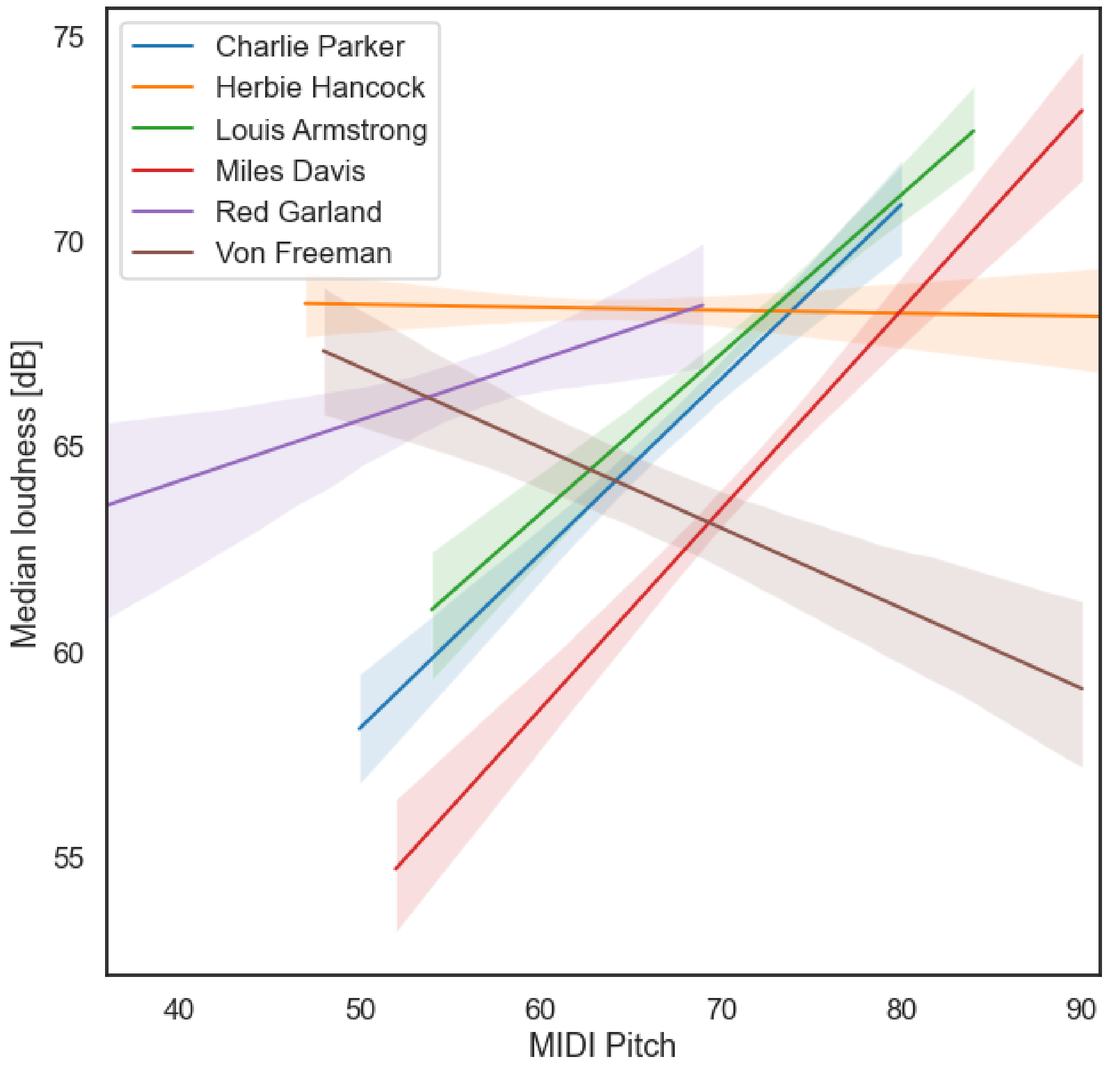
15.4. Comparing performers

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.

```
selected_performers = ["Charlie Parker", "Miles Davis", "Louis Armstrong", "Herbie 
```

```
grouped_df = solos.groupby("performer")
```

```
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
        )

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?

16. Data-Driven Music History

```
import pandas as pd # for working with tabular data
pd.set_option('display.max_columns', 500)
import matplotlib.pyplot as plt # for plotting
plt.style.use("fivethirtyeight") # select specific plotting style
import seaborn as sns; sns.set_context("talk")
import numpy as np
```

16.1. Research Questions

- General: How can we study historical changes quantitatively?
- Specific: What can we say about the history of tonality based on a dataset of musical pieces?

16.2. A bit of theory

```
note_names = list("FCGDAEB") # diatonic note names in fifths ordering
note_names
```

```
['F', 'C', 'G', 'D', 'A', 'E', 'B']
```

```
accidentals = ["bb", "b", "", "#", "##"] # up to two accidentals is suffient here
accidentals
```

```
['bb', 'b', '', '#', '##']
```

```
lof = [ n + a for a in accidentals for n in note_names ] # lof = "Line of Fifths"
print(lof)
```

```
['Fbb', 'Cbb', 'Gbb', 'Dbb', 'Abb', 'Ebb', 'Bbb', 'Fb', 'Cb', 'Gb', 'Db', 'Ab', 'Eb', 'Bb', 'F
```

```
len(lof) # how long is this line-of-fifths segment?
```

```
35
```

We call the elements on the line of fifths **tonal pitch-classes**

16.3. Data

16.3.1. A (kind of) large corpus: TP3C

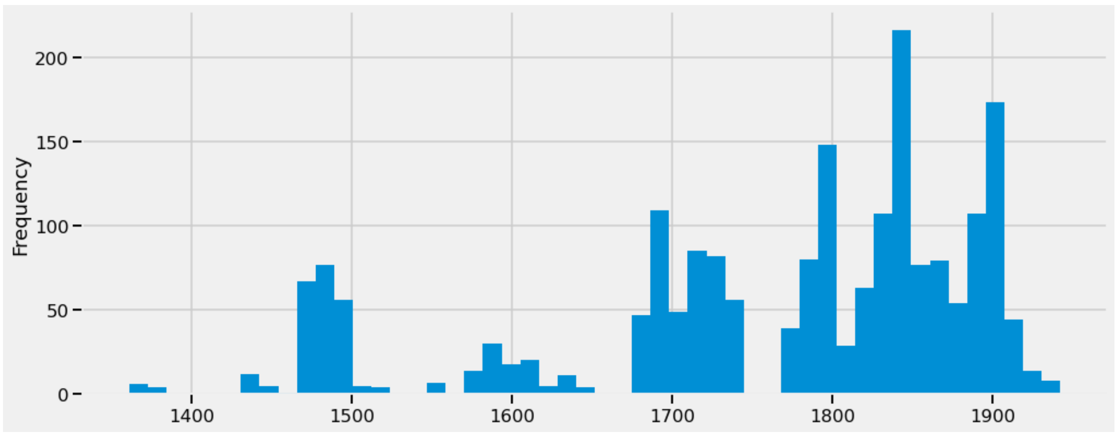
Here, we use a dataset that was specifically compiled for this kind of analysis, the **Tonal pitch-class counts corpus (TP3C)** (Moss, Neuwirth, Rohrmeier, 2020)

- 2,012 pieces
- 75 composers
- approx. spans 600 years of music history
- does not contain complete pieces but only counts of tonal pitch-classes

```
import pandas as pd # to work with tabular data

url = "https://raw.githubusercontent.com/DCMLab/TP3C/master/tp3c.tsv"
data = pd.read_table(url)

data.sample(10)
```

	composer	composer_first	work_group	work_catalogue	opus	no
1742	Corelli	Arcangelo	12 Trio Sonatas	Op.	3	4
468	Bach	Johann Sebastian	Inventions and Sinfonias	BWV	779	NaN
1630	Chopin	Frédéric	Mazurkas	Op.	33	2
1604	Joplin	Scott	Ragtimes	NaN	NaN	NaN
137	Bach	Johann Sebastian	Wohltemperiertes Klavier II	BWV	888	2
1231	Schubert	Franz	Die schöne Müllerin	D. 795	NaN	18
530	Brahms	Johannes	8 Klavierstücke	Op.	76	4
1311	Schumann	Robert	Dichterliebe	Op.	48	3
338	Bach	Johann Sebastian	Wohltemperiertes Klavier I	BWV	863	1
713	Fauré	Gabriel	NaN	Op.	6	2

```
data["display_year"].plot(kind="hist", bins=50, figsize=(15,6)); # historical overv
```



- it can be seen that there are large gaps and that some historical periods are underrepresented
- however, it is not so obvious how to fix that
- do we want a uniform distribution over time?
- do we want a "historically accurate" distribution?
- do we want to remove geographical/gender/class/instrument/etc. biases?
- on one hand, balanced datasets are likely not to reflect historical realities
- on the other hand, such datasets rather represent the "canon", that is a contemporary selection of "valuable" compositions that may differ greatly from what was considered relevant at the time

–> There is no unique objective answer to these questions. It is important to be aware of these limitations and take them into account when interpreting the results

For this workshop we ignore all the metadata about the pieces (titles, composer names etc.) but only focus on their tonal material. Therefore, we don't need all the columns of the table.

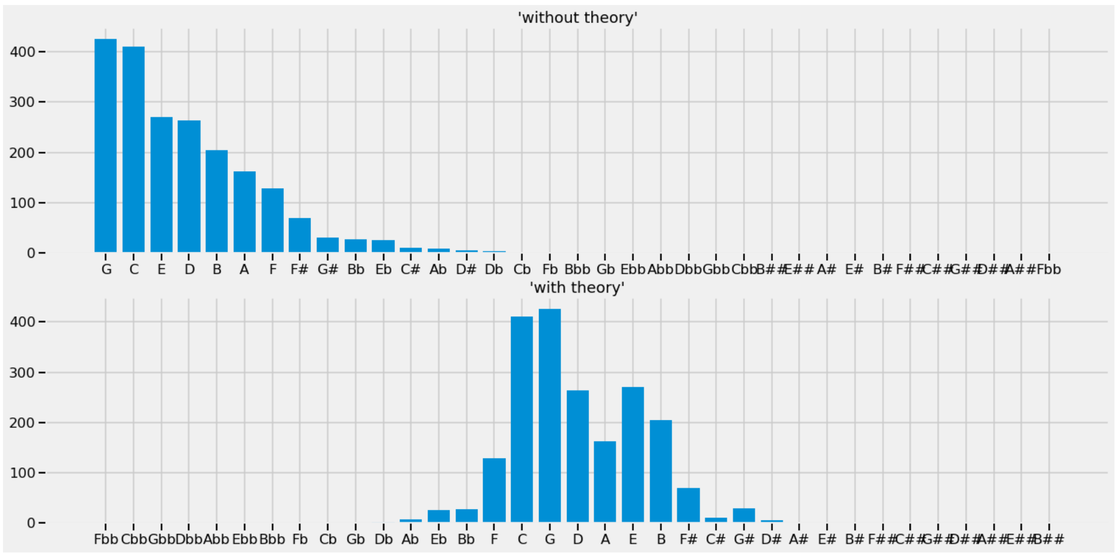
```
tpc_counts = data.loc[:, lof] # select all rows (":") and the lof columns
tpc_counts.sample(20)
```

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	Ab	Eb	Bb	F	C	G
155	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1073	0	0	0	0	0	0	0	0	11	23	83	189	219	114	229	225	323
1975	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1178	0	0	0	0	0	0	0	0	0	0	0	0	11	84	101	77	111
402	0	0	0	0	0	0	0	0	1	2	1	2	2	9	13	100	214
98	0	0	0	0	2	0	4	2	6	41	118	144	114	164	241	340	142
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	52	193	249
344	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	2	6

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	Ab	Eb	Bb	F
1784	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
407	0	0	0	0	0	0	0	0	0	0	0	3	0	11	148
1992	0	0	0	0	0	0	0	0	3	4	27	138	315	155	253
543	0	0	0	0	0	0	0	0	0	17	3	0	28	61	36
1046	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1032	0	0	0	0	0	1	5	4	18	35	47	132	95	205	173
737	0	0	0	0	19	22	16	41	103	101	33	48	251	183	97
186	0	0	0	0	0	0	0	0	0	0	4	2	16	97	176
1653	0	0	0	0	0	0	0	0	0	0	1	2	0	16	123
1115	0	0	0	0	0	0	0	0	0	10	8	6	41	64	42
1886	0	0	0	0	0	0	0	0	0	0	6	149	233	174	230
343	0	0	0	0	0	0	0	0	0	0	0	1	19	53	102

```
piece = tpc_counts.iloc[10]

fig, axes = plt.subplots(2, 1, figsize=(20,10))

axes[0].bar(piece.sort_values(ascending=False).index, piece.sort_values(ascending=F
axes[0].set_title("'without theory'")

axes[1].bar(piece.index, piece)
axes[1].set_title("'with theory'")

# plt.savefig("img/random_piece.png")
# plt.show()
```

```
Text(0.5, 1.0, "'with theory'")
```



Let us have an overview of the note counts in these pieces!

If we would just look at the raw counts of the tonal pitch-classe, we could not learn much from it. Using a theoretical model (the line of fifths) shows that the notes in pieces are usually come from few adjacent keys (you don't say!).

We probably have very long pieces (sonatas) and very short pieces (songs) in the dataset. Since we don't want length (or the absolute number of notes in a piece) to have an effect, we rather consider tonal pitch-class distributions instead counts, by normalizing all pieces to sum to one.

```
tpc_dists = tpc_counts.div(tpc_counts.sum(axis=1), axis=0)
tpc_dists.sample(20)
```

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db
606	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1844	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
251	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
737	0.0	0.0	0.0	0.0	0.017288	0.020018	0.014559	0.037307	0.093722	0.091902	0.030027
10	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.001466
822	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
202	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.003268	0.011438
850	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1609	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
555	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
940	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.005703	0.002852	0.004753	0.046578
1165	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb
1438	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1548	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1265	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.005924	0.000000
1169	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.011472
1368	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1741	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
1302	0.0	0.0	0.0	0.0	0.000000	0.000000	0.000000	0.000000	0.000000	0.002725
80	0.0	0.0	0.0	0.0	0.001965	0.002947	0.000000	0.008841	0.030452	0.051081

For further numerical analysis, we extract the data from this table and assign it to a variable `X`.

```
# extract values of table to matrix
X = tpc_dists.values

X.shape # shows (#rows, #columns) of X
```

```
(2012, 35)
```

Now, `X` is a 2012 × 35 matrix where the rows represent the pieces and the columns (also called "features" or "dimensions") represent the relative frequency of tonal pitch-classes.

Thinking in 35 dimensions is quite difficult for most people. Without trying to imagine what this would look like, what can we already say about this data?

Since each piece is a point in this 35-D space and pieces are represented as vectors, pieces that have similar tonal pitch-class distributions must be close in this space (whatever this looks like).

What groups of pieces that cluster together? Maybe pieces of the same composer are similar to each other? Maybe pieces from a similar time? Maybe pieces for the same instruments?

If we find clusters, these would still be in 35-D and thus difficult to interpret. Luckily, there are a range of so-called *dimensionality reduction* methods that transform the data into lower-dimensional spaces so that we actually can look at them.

A very common dimensionality reduction method is **Principal Components Analysis (PCA)**.

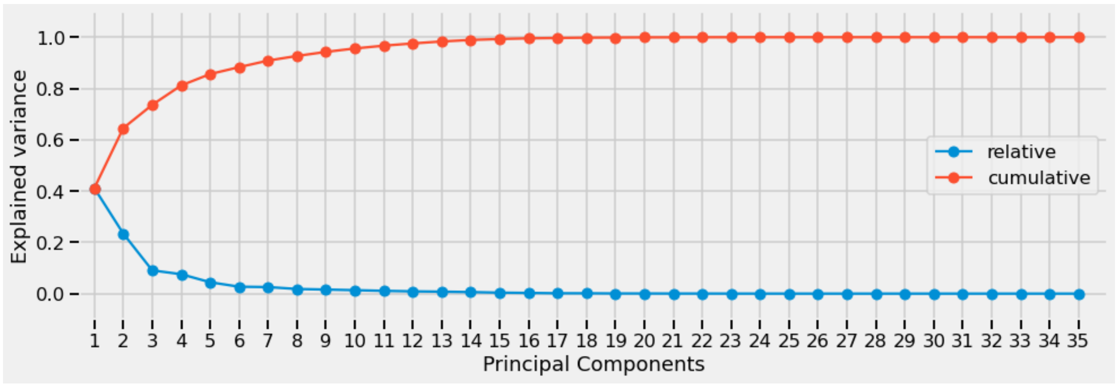
The basic idea of PCA is:

- find dimensions in the data that maximize the variance in this direction

- these dimensions have to be orthogonal to each other (mutually independent)
- these dimensions are called the *principal components*
- each principal component is associated with how much of the data variance it explains

```
import numpy as np # for numerical computations
import sklearn
from sklearn.decomposition import PCA # for dimensionality reduction

pca = sklearn.decomposition.PCA(n_components=35) # initialize PCA with 35 dimensions
pca.fit(X) # apply it to the data
variance = pca.explained_variance_ratio_ # assign explained variance to variable
```

```
fig, ax = plt.subplots(figsize=(14,5))
x = np.arange(35)
ax.plot(x, variance, label="relative", marker="o")
ax.plot(x, variance.cumsum(), label="cumulative", marker="o")
ax.set_xlim(-0.5, 35)
ax.set_ylim(-0.1, 1.1)
ax.set_xlabel("Principal Components")
ax.set_ylabel("Explained variance")
plt.xticks(np.arange(len(lof)), np.arange(len(lof)) + 1) # because Pyhon starts counting at 0

plt.legend(loc="center right")
plt.tight_layout()
# plt.savefig("img/explained_variance.png")
# plt.show()
```



```
variance[:5]
```

```
array([0.41144591, 0.23410347, 0.09063507, 0.07574242, 0.04436989])
```

The first principal component explains 41.1% of the variance of the data, the second explains 23.4% and the third 9%. Together, this amounts to 73.6%.

Almost three quarters of the variance in the dataset is retained by reducing the dimensionality from 35 to 3 dimensions (8.6%)! If we reduce the data to two dimensions, we still can explain $\approx$ 65% of the variance.

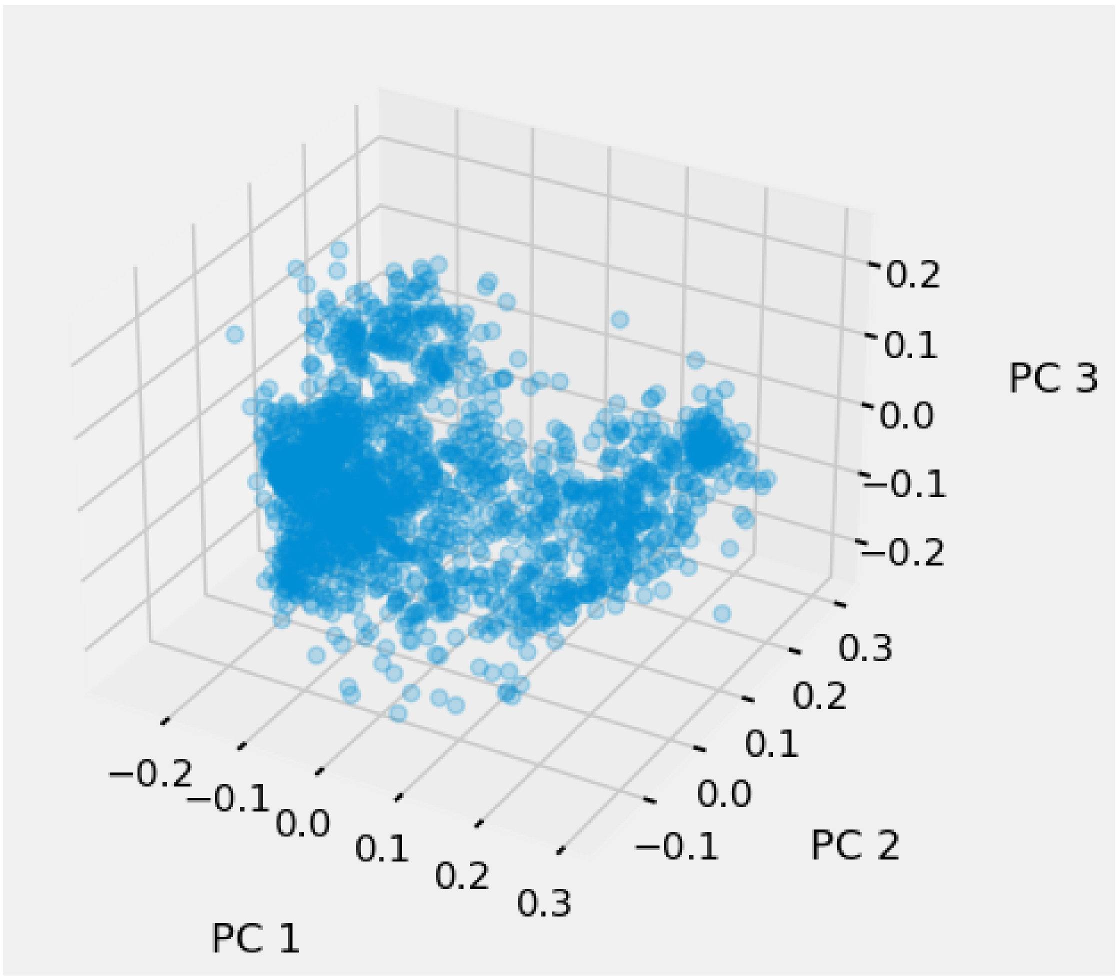
This is great because it means that we can look at the data in 2 or 3 dimensions without loosing too much information.

16.4. Recovering the line of fifths from data

```
pca3d = PCA(n_components=3)
pca3d.fit(X)

X_ = pca3d.transform(X)
X_.shape
```

```
(2012, 3)
```

```
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(6,6))

ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_[:,0], X_[:,1], X_[:,2], s=50, alpha=.25) # c=cs,
ax.set_xlabel("PC 1", labelpad=30)
ax.set_ylabel("PC 2", labelpad=30)
ax.set_zlabel("PC 3", labelpad=30)

plt.tight_layout()
# plt.savefig("img/3d_scatter.png")
# plt.show()
```

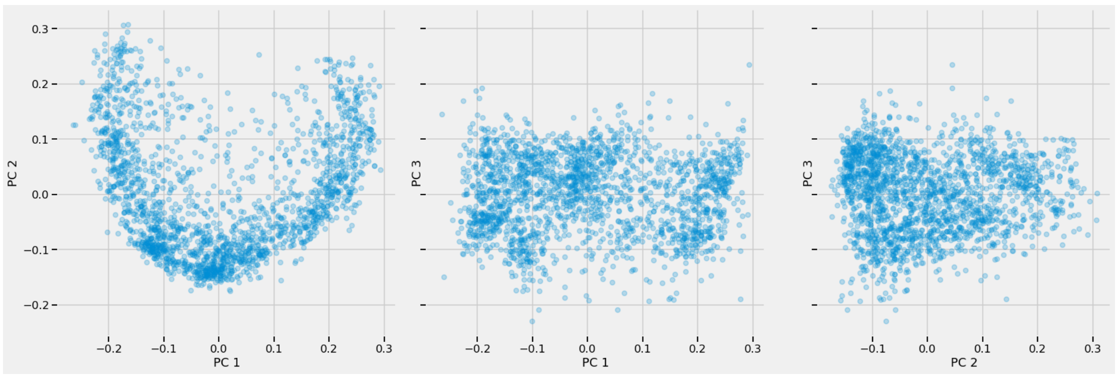


Each piece in this plot is represented by a point in 3-D space. But remember that this location represents ~75% of the information contained in the full tonal pitch-class distribution. In 35-D space each dimension corresponded to the relative frequency of a tonal pitch-class in a piece.

- What do these three dimensions signify?
- How can we interpret them?

Fortunately, we can inspect them individually and try to interpret what we see.

```
from itertools import combinations

fig, axes = plt.subplots(1,3, sharey=True, figsize=(24,8))

for k, (i, j) in enumerate(combinations(range(3), 2)):

    axes[k].scatter(X_[:,i], X_[:,j], s=50, alpha=.25, edgecolor=None)
    axes[k].set_xlabel(f"PC {i+1}")
    axes[k].set_ylabel(f"PC {j+1}")
    axes[k].set_aspect("equal")

plt.tight_layout()
# plt.savefig("img/3d_dimension_pairs.png")
# plt.show()
```



Clearly, looking at two principal components at a time shows that there is some latent structure in the data. How can we understand it better?

One way to see whether the pieces are clustered together systematically be coloring them according to some criterion.

As always, many different options are available. For the present purpose we will use the most simple summary of the piece: its most frequent note (which is the *mode* of its pitch-class distribution in statistical terms) and call this note its **tonal center**.

This will also allow to map the tonal pitch-classes on the line of fifths to colors.

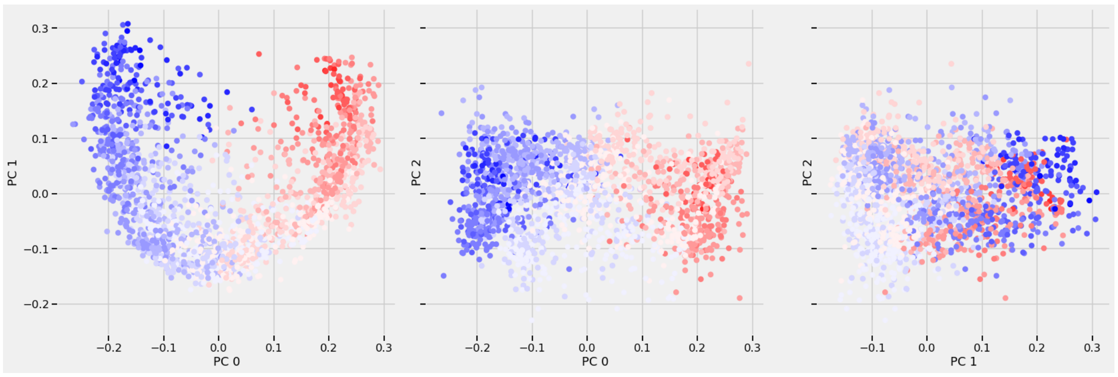
```
tpc_dists["tonal_center"] = tpc_dists.apply(lambda piece: np.argmax(piece[lof].valu
tpc_dists.sample(10)
```

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	A
1990	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.002752	0.004587	0.015138	0.0
364	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.001026	0.006028	0.020521	0.0
1857	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.0
1045	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.0
1246	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.01232	0.020534	0.008214	0.131417	0.1
214	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.0
167	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.0
570	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.0

	Fbb	Cbb	Gbb	Dbb	Abb	Ebb	Bbb	Fb	Cb	Gb	Db	Ab	Eb
586	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.001506	0.006274	0.04
684	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.00000	0.000000	0.000000	0.000000	0.000000	0.00

```python
from matplotlib import cm
from matplotlib.colors import Normalize

#normalize item number values to colormap
norm = Normalize(vmin=-15, vmax=20)

# cs = [ cm.seismic(norm(c)) for c in data["tonal_center"]]
cs = [ cm.seismic(norm(c)) for c in tpc_dists["tonal_center"]]
```

```python
from itertools import combinations

fig, axes = plt.subplots(1,3, sharey=True, figsize=(24,8))

for k, (i, j) in enumerate(combinations(range(3), 2)):

    axes[k].scatter(X_[:,i], X_[:,j], s=50, c=[ np.abs(c) for c in cs], edgecolor=None)
    axes[k].set_xlabel(f"PC {i}")
    axes[k].set_ylabel(f"PC {j}")
    axes[k].set_aspect("equal")

plt.tight_layout()
# plt.savefig("img/3d_dimension_pairs_colored.png")
# plt.show()
```



16.5. Historical development of tonality

The line of fifths is an important underlying structure for pitch-class distributions in tonal compositions

But we have treated all pieces in our dataset as synchronic and have not yet taken their historical location into account.

Let's assume the pitch-class content of a piece spreads on the line of fifths from F to A♯. This means, its range on the line of fifths is $10 - (-1) = 11$. The piece covers eleven consecutive fifths on the lof.

We can generalize this calculation and write a function that calculates the range for each piece in the dataset.

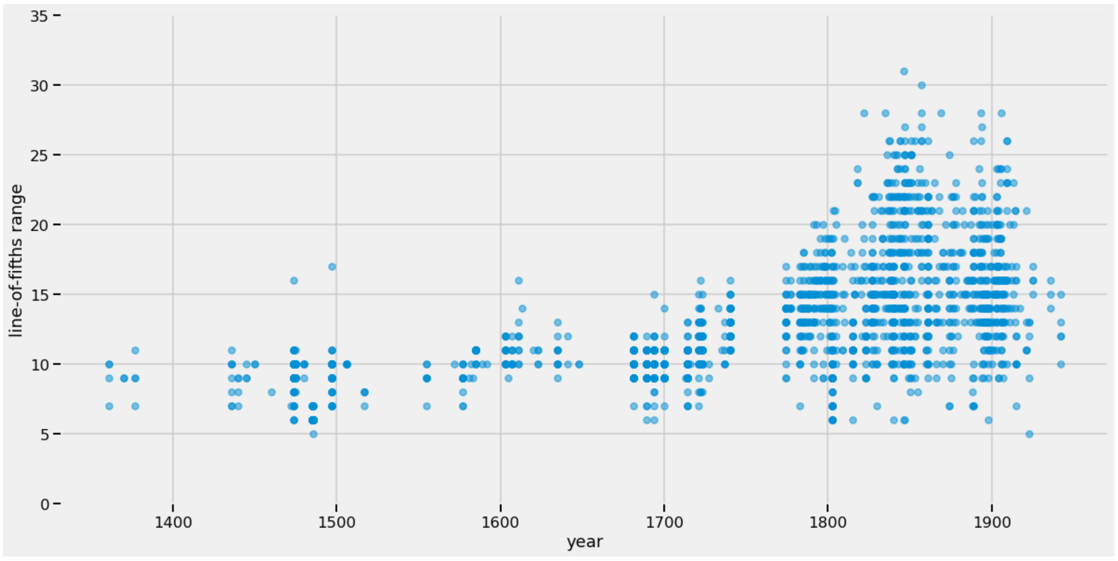
```
def lof_range(piece):
    l = [i for i, v in enumerate(piece) if v!=0]
    return max(l) - min(l)
```

```
data["lof_range"] = data.loc[:, lof].apply(lof_range, axis=1) # create a new column
data.sample(20)
```

	composer	composer_first	work_group	work_catalogue
992	Agricola	Alexander	Missa Malheur me bat	NaN
1772	Corelli	Arcangelo	12 Trio Sonatas	Op.
1199	Scarlatti	Domenico	Sonata	K
1361	Scriabin	Alexander	Préludes	Op.
1490	Tchaikovsky	Pyotr	The Seasons	Op.
387	Chopin	Frédéric	Préludes	Op.
290	Alkan	Charles Valentin	Préludes	Op.
478	Bach	Johann Sebastian	Inventions and Sinfonias	BWV
1947	Mozart	Wolfgang Amadeus	Sonaten	KV
601	Couperin	François	Troisième livre de pièces de Clavecin	NaN
974	Ockeghem	JeanDe	Missa Fors Seulement	NaN
1430	Victoria	TomasLuisde	NaN	NaN
528	Brahms	Johannes	8 Klavierstücke	Op.
1765	Corelli	Arcangelo	12 Trio Sonatas	Op.
1300	Schumann	Clara	Sechs Lieder	Op.
204	Liszt	Franz	12 Transcendental Etudes	S.
1858	Grieg	Edvard	Lyrical Pieces	Op.
468	Bach	Johann Sebastian	Inventions and Sinfonias	BWV
439	Victoria	TomasLuisde	NaN	NaN
441	Alkan	Charles Valentin	NaN	Op.

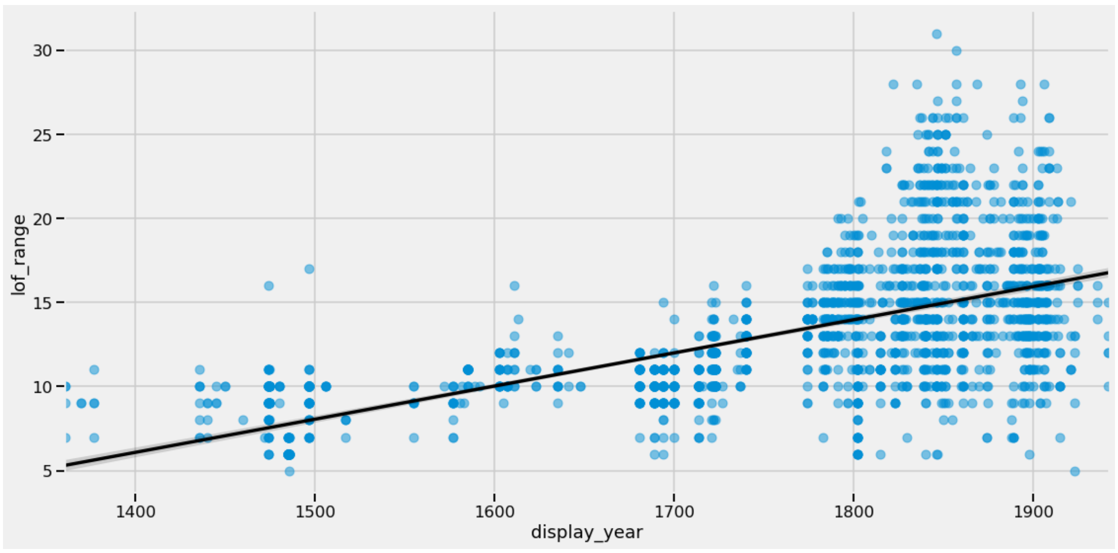
This allows us now to take the `display_year` (composition or publication) and `lof_range` (range on the line of fifths) features to observe historical changes.

```
fig, ax = plt.subplots(figsize=(18,9))
ax.scatter(data["display_year"].values, data["lof_range"].values, alpha=.5, s=50)
ax.set_ylim(0,35)
ax.set_xlabel("year")
ax.set_ylabel("line-of-fifths range");
# plt.savefig("img/hist_scatter.png");
```



We could try to fit a line to this data to see whether there is a trend (kinda obvious here).

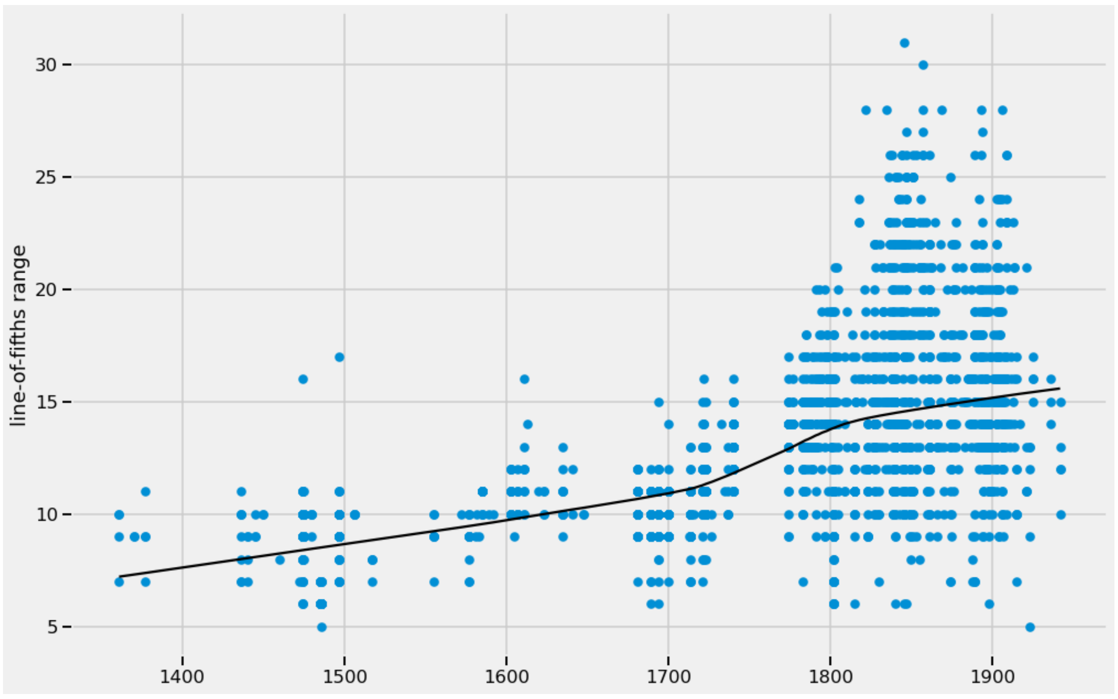
```
g = sns.lmplot(
    data=data,
    x="display_year",
    y="lof_range",
    line_kws={"color":"k"},
    scatter_kws={"alpha":.5},
#     lowess=True,
    height=8,
    aspect=2
);
# g.savefig("img/hist_scatter_line.png");
```



But actually, this is not the best idea. Why should any historical process be linear? More complex models might make more sense.

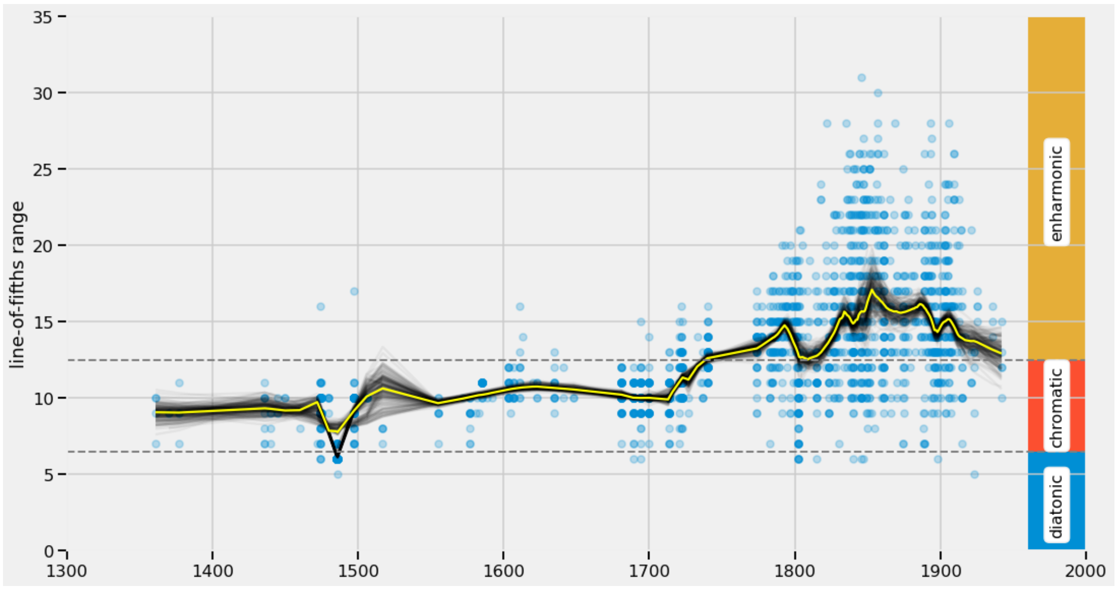
A more versatile technique is *Locally Weighted Scatterplot Smoothing* (LOWESS) that locally fits a polynomial. Using this method, we see that a non-linear process is displayed.

```python
from statsmodels.nonparametric.smoothers_lowess import lowess

x = data.display_year
y = data.lof_range
l = lowess(y,x)

fig, ax = plt.subplots(figsize=(15,10))

ax.scatter(x,y, s=50)
ax.plot(l[:,0], l[:,1], c="k")
ax.set_ylabel("line-of-fifths range");
# plt.savefig("img/hist_scatter_lowess.png")
# plt.show()
```



16.6. If there is time: some more advanced stuff

```
B = 200
delta = 1/10

fig, ax = plt.subplots(figsize=(16,9))

x = data.display_year
y = data.lof_range
l = lowess(y,x, frac=delta)

ax.scatter(x,y, s=50, alpha=.25)

for _ in range(B):
    resampled = data.sample(data.shape[0], replace=True)

    xx = resampled.display_year
    yy = resampled.lof_range
    ll = lowess(yy,xx, frac=delta)

    ax.plot(ll[:,0], ll[:,1], c="k", alpha=.05)
ax.plot(l[:,0], l[:,1], c="yellow")

## REGIONS
from matplotlib.patches import Rectangle

text_kws = {
    "rotation" : 90,
    "fontsize" : 16,
    "bbox" : dict(
        facecolor="white",
        boxstyle="round"
    ),
    "horizontalalignment" : "center",
    "verticalalignment" : "center"
}

rect_props = {
    "width" : 40,
    "zorder" : -1,
    "alpha" : 1.
}

stylecolors = plt.rcParams["axes.prop_cycle"].by_key()["color"]

ax.text(1980, 3, "diatonic", **text_kws)
ax.axhline(6.5, c="gray", linestyle="--", lw=2) # dia / chrom.
ax.add_patch(Rectangle((1960,0), height=6.5, facecolor=stylecolors[0], **rect_props

ax.text(1980, 9.5, "chromatic", **text_kws)
ax.axhline(12.5, c="gray", linestyle="--", lw=2) # chr. / enh.
ax.add_patch(Rectangle((1960,6.5), height=6, facecolor=stylecolors[1], **rect_props

ax.text(1980, 23.5, "enharmonic", **text_kws)
ax.add_patch(Rectangle((1960,12.5), height=28, facecolor=stylecolors[2], **rect_pro

ax.set_ylim(0,35)
ax.set_xlim(1300,2000)

ax.set_ylabel("line-of-fifths range");
# plt.savefig("img/final.png", dpi=300)
# plt.show()
```

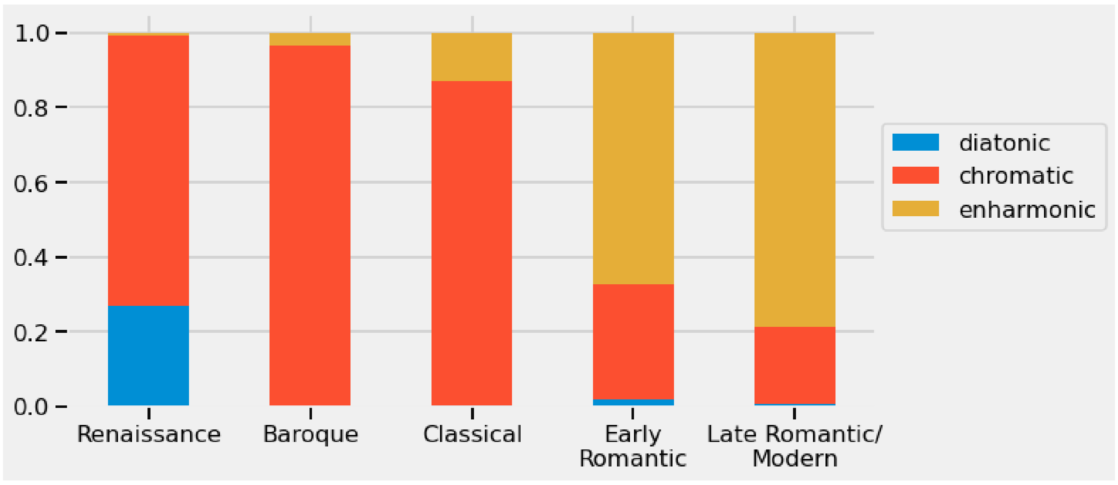


Usung bootstrap sampling we achieve an estimation of the local varience of the data and thus of the diversity in the note usage of the musical pieces.

We also can distinguish three regions in terms of line-of-fifth range: diatonic, chromatic, and enharmonic.

Grouping the data together in these three regions, we see a clear change from diatonic and chromatic to chromatic and enharmonic pieces over the course of history.

```
epochs = {
    "Renaissance" : [1300, 1549],
    "Baroque" : [1550, 1649],
    "Classical" : [1650, 1749],
    "Early\nRomantic" : [1750, 1819],
    "Late Romantic/\nModern" : [1820, 2000]
}

strata = [
    "diatonic",
    "chromatic",
    "enharmonic"
]

widths = data[["display_year", "lof_range"]].sort_values(by="display_year").reset_index(drop=T

df = pd.concat(
    [
        widths[
            (widths.display_year >= epochs[e][0]) & (widths.display_year <= epochs[
        ]["lof_range"].value_counts(normalize=True).sort_index().groupby(
            lambda x: strata[0] if x <= 6 else strata[1] if x <= 12 else strata[2]
        ).sum() for e in epochs
    ], axis=1, sort=True
)

df.columns = epochs.keys()
df = df.reindex(strata)
df.T.plot(kind="bar", stacked=True, figsize=(12,5))
# plt.title("Epochs")
plt.legend(bbox_to_anchor=(1.3,0.75))
plt.gca().set_xticklabels(epochs.keys(), rotation="horizontal")
plt.tight_layout()
# plt.savefig("img/epochs_regions.png")
plt.show()
```



- Renaissance: largest diatonic proportion overall but mostly chromatic
- Baroque: alost completely chromatic
- Classical: enharmonic proportion increases -> more distant modulations
- This trend continues through the Romantic eras

16.7. Summary

1. We have analyzed a very specific aspect of Western classical music.
2. We have used a large(-ish) corpus to answer our research question.
3. We have operationalized musical pieces as vectors that represent distributions of tonal pitch-classes.

4. We have used the dimensionality-reduction technique Principal Component Analysis (PCA) in order to visually inspect the distribution of the data in 2 and 3 dimensions.
5. We have used music-theoretical domain knowledge to find meaningful structure in this space.
6. We have seen that pieces are largely distributed along the line of fifths.
7. We have used Locally Weighted Scatterplot Smoothing (LOWESS) to estimate the variance in this historical process.
8. We have seen that, historically, composers explore ever larger regions on this line and that the variance also increases.

Part V.

CRITICAL DIGITAL MUSICOLOGY

17. Copyright

Goal

Know a few famous copyright infringement cases and why data analysis is important here.

- Plagiarism cases and copyright

18. Representation and representativeness

Goal

Understand the difference between representativeness and representation. Obtain a critical understanding of biases relevant for data selection.

- Representation and the canon
- Representing means modeling means abstraction (what is “music” in “music encoding”?)
- biases: how to recognize them, how to deal with them, and when biases are a good thing.
- FAIR and CARE

19. Discussion

Goal

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.