FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Name Surname

Thesis title

Name of the department

Supervisor of the bachelor thesis: Supervisor's Name
Study programme: study programme
Study branch: study branch

Prague YEAR

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date ....

Author's signature

Dedication.

Title: Thesis title

Author: Name Surname

Department: Name of the department

Supervisor: Supervisor's Name, department

Abstract: Abstract.

Keywords: key words

Contents

Introduction

Imagine, for a moment, that you are transported in time to 10th-century Europe. It is Sunday, and the weekly mass is just starting. Look around yourself. The tall ceilings of the impressive church were built so to bring you closer to God. The stained glass windows let through just enough light so that it is not completely dark. Can you smell the incense? Everything in the environment around you reminds you that you are taking part in a sacred ritual. Listen to the priest; when he recites the prayers, his speech does not flow in a natural way. Instead, the entire text is intoned on a single note, except for the slightly inflected ends of clauses. Sometimes, the choir replaces the priest, singing more elaborate monophonic melodies. The words they are singing are Latin, but even if you do not understand them, you know what their purpose is: to celebrate the deity. Their voices echo in the stone church, creating an otherwordly experience.

What you are hearing is called Gregorian chant, one of the oldest preserved types of music. It has been central to the cultural development of most of Europe, as it is a part of the Roman-catholic tradition.

Gregorian chant is a widely researched topic of study, partly thanks to the substantial amount of available sources. However, traditional musicology is limited by the individual capacity; it is infeasible to study extremely large numbers of chants by hand. This is where Digital Humanities come into play. Digitalization of large corpora of Gregorian chant enables researchers to draw conclusions supported by much more data than a human could ever examine.

Nevertheless, contemporary Gregorian chant research still lacks many tools that would facilitate the study of rhythm (which was not recorded) or the internal differentiation of chant. Another important problem is chant's historical and geographical development. Throughout the centuries, the tradition spread accross Europe, while always changing a small part of it. This has led to many variations of the same chants existing at the same time. Knowing how exactly the changes came into place could shed light on many open questions in history.

The aim of this thesis is to create a tool that musicologists could use to answer the questions about chants' origin. We do this by employing well-known algorithms for multiple sequence alignment from bioinformatics. Once the idea is proposed, it is almost immediately obvious that biological and musical sequences share features which warrant using the same methods on both. As opposed to other sequence comparison techniques, such as edit distance, MSA algorithms do not need any assumption about the length of chants or other characteristics, which is why they are suitable for the study of chant.

The tool provides the possibility to align a set of chants in multiple modes, each of which has slightly different uses. An important contribution of the tool is that it makes the discovery contrafacta (chants with the same melody but different lyrics) and transpositions (chants with the same melody but shifted by an interval) as easy as it has never been.

It is important to note that the purppose of this thesis is not to conduct experiments. It is merely a tool for musicologists who possess the knowledge needed to select the appropriate data so that some important discoveries can be made.

1. Related work

With the advent of computers, the field of Music Information Retrieval (MIR) emerged. Research in this interdisciplinary field focuses on extracting information from musical notation using computer science methods, such as signal processing or machine learning. Its applications vary widely, from recommender systems, to automatic audio transcription, to music generation. MIR encompasses all different kinds of music, regardless of their location, age, or function. Researchers have developed a multitude of software tools that facilitate music analysis, irrespective of what type of music it is. One of such toolkits is *music21* [Cuthbert and Ariza, 2010], a Python package able to encode musical notation as Python objects and perform analysis on large datasets.

The study of plainchant using computational methods has not been done extensively. The main research tool for musicologists in this field is the Cantus Index [Lacoste and Koláček]. It is an online index of chants from several different chant databases, providing researchers with a common API for all of them. The entries in Cantus Index only contain four data fields: full-text, genre, feast (not required), and Cantus ID, which is automatically assigned to newly added chants. The tool is also able to search for melodies in the original source and even provides search-by-melody functionality. There are ten databases indexed in the catalogue, the largest of which is the Cantus database [Lacoste et al.].

Cornelissen et al. [2020a] developed *chant21*, a Python package able to convert two standard melodic notations, *volpiano* and *gabc* to a *music21* object, therefore making it easier to study Gregorian chant computationally. The data they used were scraped from Cantus database [Lacoste et al.] and GregoBase [Berten and contributors] and released as CantusCorpus and GregoBaseCorpus, respectively. Finally, they performed two case studies using the package. In the first one, they confirmed the melodic arch hypothesis [Huron, 1996], which had previously only been studied manually. Second, they analyzed the relation between differentiæ and antiphon openings [Shaw, 2018] and found that it differs accross modes.

Some of the computational research into plainchant has been centered on mode classification. Huron and Veltman [2006] used pitch class profiles to classify modes. They created a pitch-class distribution for each of the eight modes, and used these classes to classify previously unseen data. Cornelissen et al. [2020b] compared three approaches to mode classification: classical approach, which classifies chants based on the final pitch, range, and the initial pitch; profile approach, which was largely inspired by Huron and Veltman [2006]; and distributional approach, which focuses on the melodic aspect of mode. The authors chose various segmentations and representations of chants and used a tf-idf vector model to classify mode. The study found that we can accurately classify mode even when we discard all absolute pitch information, the melody contour contains enough information on its own.

A considerable amount of research has been done into the evaluation of melodic similarity, albeit not for Gregorian chant specifically. Wickland [2017] provides an overview of the methods. He mentions edit distance, Markov chains, and geometric measurements as the most widely used ones. Park et al. [2019] used an adapted edit distance metric to calculate the similarity of two melodic sequences by first calculating the similarity for all segments of each of the sequences and then scaling them by a weight function depending on the segment length, which yielded them what they call a multi-scale similarity stack. The overall similarity was obtained by averaging its values. Then they used the MSS stack to create a visualization that takes on the shape of a trapezoid that shows which segments of two sequences are the most similar.

Bountouridis et al. [2017] argue that methods originally developed for bioinformatics have a great potential to be applied to music. They offer analogies for bioinformatics concepts found in musicology. For example, they liken DNA and proteins to melodic sequences, homologues (proteins that have the same ancestor) to song covers, evolution to oral transmission, etc. They claim that despite the similarities, MRI has not leveraged the full potential of bioinformatics methods. In their article, they focus on modelling melodic similarity using multiple-sequence alignment (MSA) algorithms, therefore not relying on heuristics, as opposed to previous works. Their results revealed that the MAFFT algorithm yields the best alignemnt, which can be attributed to the algorithm using gap-free segments as anchor points, therefore partitioning melodies into more meaningful segments than other algorithms.

The general algorithm for calculating pairwise sequence alignment is the Needleman-Wunsch algorithm [Needleman and Wunsch, 1970]. It uses dynamic programming to break down the problem into smaller problems. Given two sequences, it starts aligning them from the beginning. At each point, the algorithm checks whether the two sequences match in the current position, and if not, whether it will leave the elements mismatched or insert a space. In essence, all possible alignments are computed and scored and the best one is chosen. The algorithm always yields an optimal alignment, therefore it is used when the quality of the alignment is important. However, because of its time complexity, it is unsuitable for many applications.

Unlike pairwise sequence alignment, multiple sequence alignment has been shown to be NP-complete [Wang and Jiang, 2009]. As such, there is no practical way of computing an optimal MSA and we must instead rely on heuristics to obtain a sufficiently good alignment.

Notredame [2007] provides an overview of modern multiple sequence alignment algorithms. According to him, the most frequently used algorithms use the progressive approach, where a guide tree is estimated from unaligned sequences and then pairwise alignment algorithms are used to find the MSA following the tree. He notes that the scoring methods of the pairwise algorithm are essential. There are two main groups of scoring methods: matrix-based algorithms, where a substitution matrix is used to determine the cost of replacing one symbol with another, and the consistency-based methods, which use a collection of global and local alignments to calculate a position-specific substitution matrix. The author claims that the best methods yield indistinguishable results, except for remote homologs with less than 25% identity.

T-Coffee [Notredame et al., 2000] uses the progressive approach described above. It was the first algorithm that used a preprocessed collection of alignments to create a library that helps create the guide tree. The library is generated using both global and local pairwise alignments. Thanks to this approach, T-Coffee minimizes the errors made in the first stages of building the MSA, which is a shortcoming of many previous algorithms, as these errors tend to persist. They combined precomputed local and global alignments and create a function that assigns a weight to each pairwise alignment depending on how consistent the pair of residues is with the residue pairs from all other alignments. This process leads to a significant improvement of the results.

MAFFT [Katoh et al., 2002] further improves on other methods by using Fast Fourier transform to identify homologues fast. In addition, the authors propose a simplified scoring system that reduces CPU time while maintaining its accuracy. The authors' results showed a performance 100 times better than that of T-Coffee.

Despite the fact that MSA algorithms have already been used for music analysis, this is the first work that focuses specifically on applying the methods on Gregorian chant. Gregorian chant is specific by its monophonic nature, which means that there is just one sequence to analyze. Each chant has also undergone many changes over the centuries, therefore there are many variants of the same chants that can be researched. These characteristics make Gregorian chant an ideal subject for MSA analysis and application of related bioinformatics methods.

2. Gregorian chant

This chapter is adapted after Hiley [2009].

Gregorian chant is the music associated with the Roman catholic tradition. It is sung in churches, as well as in convents. Its current form is, as far as we can tell, very similar to its original form from the first millenium.

Gregorian chant is one of the earliest forms of music preserved in written form, and the largest preserved body of medieval music. The earliest preserved fragments of written notes date back to the 9th century, although texts from as early as the eighth century have been found.. Its name, Gregorian, references Pope Gregory the Great, however, his relation to the chant is not entirely clear.

It is not the only type of chant. In the early centuries after Christianity spread across Western and Eastern Roman Empire, new forms of worship started being developed. The most obvious differences were between the West and the East, which had multiple cultural centers such as Constantinople, Jerusalem, or Alexandria. However, liturgies varied in the West as well, from Rome to Milan to the Iberian peninsula to Gaul. Each center developed their own tradition, including their own type of chant. Roman-catholic tradition being so prevalent in Europe can be attributed to Charlemagne's attempt to unify European kingdoms.

Gregorian chant is an integral part of the Roman church, and has been so for centuries. It is the monophonic music (i.e. single-voice) sung during liturgies. Here, liturgy means chant sung during Christian worship. Unlike in the Eastern churches, where the term is reserved for the Eucharist, liturgy includes both the Mass and the Divine Office in the Roman church.

As with other kinds of music, there exist several genres of Gregorian chant. A *genre* is a group of chants that share some characteristics, such as complexity or content. Different genres play different roles in the liturgy. Some are sung while the priest is walking to the altar, others convey the core message of a mass, others have just a decorative purpose.

Not all genres are sung during every kind of worship. In the following section, we describe the Mass and the Divine Office and mention some genres that are specific for each.

2.1 Mass and Divine Office

Mass is the service most familiar to most believers. It can be divided into several parts, all leading up to the most important one, the act of communion. This act commemorates the Last Supper, Jesus's last meal with his disciples before his execution. During the communion, bread, representing Christ's body, and wine, representing his blood, are given out. During the course of the Mass, multiple different chants are sung. *Introitus*, meaning 'entrance', is sung at the beginning of the service while the priest and his assistants are walking to the altar. *Tractus* is a chant that is sung during Lent, i.e. the period between Ash Wednesday and the Saturday before Easter. Outside of this period, *alleluia* replaces *tractus* and is followed by *sequentia* on the most important feast days.

The other part of the liturgy, besides the Mass, is the Divine Office, also called Liturgy of the Hours or canonical hours. It is the set of chants sung during services at different times of the day, for example *Vespers* in the evening or *Lauds* in the morning. The office consists largely of singing psalms, of which there are 150, all sung on different days and hours. Psalms were usually preceded and followed by antiphons, which differ not only by the day of the week, but also depending on the place. Each day of the week had an allocated set of antiphons, hymns and responsories, while responsories were assigned to the different Sundays. Additionally, important feasts had their own set of chants to be sung.

The liturgical year contains several feasts. Some feasts are fixed on a specific date. Feasts associated with a specific saint are an example of those. For example, John the Baptist is celebrated on the 24th of June, Michael on the 29th of September, and All Saints on the 1st of November. Other fixed-date feasts include Christmas Day (25 December), Epiphany (6 January), and others. Such feasts can fall on any day of the week, and if they fall on a Sunday, they will take precedence over it. On the other hand, there are also feasts that are fixed to a specific day of the week, the most important of them being Easter Sunday, the day when Christ rose from the dead. Each feast has a different set of chants that can be completely original.

2.2 Variation between chants

The individual chants differ in several criteria, the first one being the *mode*. Mode is the system of pitch organization, somewhat similar to modern-day scales. All chants use the seven tones of a diatonic scale, meaning a scale with five whole steps and two half steps.

Melodies are classified into one of eight modes according to their last note, called *finalis*, and their range. Most chants end on one of the notes *D*, *E*, *F*, or *G*. These four notes determine four pairs of modes. The melodies were further classified depending on whether they moved mostly in the range above the *finalis*, in which case it would be classified as the *authentic* mode of the pair, or in the range around the *finalis*, which means it is classified as the *plagal* mode. Some types of chant tend to occur mostly in a specific mode.

Another criterion is the complexity of chants, that is, how elaborate the melody is. On the one side of the spectrum, there is a one-to-one syllable to note correspondence. Antiphons and some hymns are genres close to this text-setting. The other extreme is melismatic style. Melisma is a long vocalization of a single syllable, therefore melismatic melodies are more ornate. Some of the more melismatic genres are the gradual, tract and offertory.

We have already mentioned several different genres of chant. Antiphons are chants sung to frame a psalm during the Office hours. They are relatively short and simple, somtimes consisting of only two phrases. Responsories are sung during the Night Office. Each has a main section and a verse. They are melodically very rich and are on of the most impressive forms of chant. The number of chants sung during the Mass is lower, as there will usually only be one introit, one gradual, etc. during a service.

It is important to note that text and melody do not form unique pairs. Instead, each text can be sung to multiple different melodies and multiple texts can be sung to one melody. Chants with the same melody but different lyrics are called *contrafacta*.

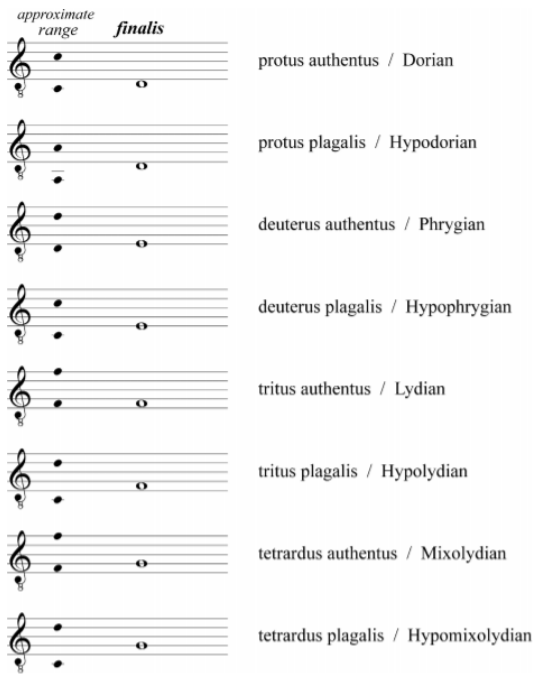


Figure 2.1: List of modes, their ranges and finalis. [Hiley, 2009, p. 44]

2.3 Chant notation

It is clear that the annual cycle is very complex and the amount of chants is abundant. Therefore, it is not surprising that while the chants during the services themselves were sung from memory, they were written down in books. Each church and each convent had their own manuscripts, which is the reason of their abundance.

At first, melodies were written without a staff. The writers simply marked the direction in which the pitch was moving, serving barely as memory aids, not for learning. Later, better accuracy was required, therefore melodies started being written with their exact pitch, resembling the modern-day musical notation. However, written melodies never contained information about the rhythm. An example of a chant melody as found in a manuscript is shown in Figure 2.2.

Figure 2.2: Example of a chant in a manuscript. [Lacoste et al., id 007553]

In the middle ages, tools to tune singers' voice to a specific pitch were not readily available. This led to the singing “by feeling”. As a result, some churches or convents sang the same melody, but shifted by a certain interval, and they were inscribed in manuscripts with this shift. Such melodies (those that differ by the same interval in all positions) are called *transpositions*.

2.4 Chant research

3. Data

Our main source of data is the Cantus database [Lacoste et al.], one of the databases indexed in the Cantus Index. The database serves as a digital archive of chants, each entry containing information about its source, liturgical occasion, mode, and others. Work on the project started in the late 1980s, and to date, around 500,000 individual chants from approximately 150 manuscripts have been indexed. Each entry is transcribed manually and undergoes a thorough examination before publishing [Lacoste, 2012].

We are using a scraped version of the Cantus database released as CantusCorpus [Cornelissen et al., 2020a]. Unlike the Cantus database which is continuously being updated and is therefore unsuitable for computational study, the corpus is versioned, therefore each version always contains the same data. We are using version 0.2 released in July 2020 which contains 497,071 entries. However, a majority of the data is not suitable for this application, as they do not contain both melodic and textual data in full, therefore we are only using a subset of size around 13,000. The corpus is available for download in CSV format.

3.1 CSV

CSV is one of the most common formats for tabular data. The abbreviation stands for *comma-separated values*. As the name suggests, the format uses commas to separate columns (although other separators, such as a semicolon, can be used as well to allow for simpler parsing in case that the data frequently contains commas that would otherwise need to be escaped), while the individual rows are separated by a line break. The data is stored as plaintext, which makes it easily readable. Parsing CSV files becomes more complicated when the data contains column and row separators inside fields; in that case quotation marks or esape sign has to be used. There exist many well-designed parsers, one such parser is the Python module simply called *csv*[1]. This application uses the module *pandas*[2] to parse CSV files, which in turn uses the *csv* module.

3.2 Database fields

The following table represents the data fields in the database.

Data field	Description
id	automatically generated id in the database
corpus_id	human-readable id identifying the chant in the CantusCorpus
incipit	incipit (the first few words) of chant
cantus_id	id identifying the chant in the Cantus Index
mode	mode of the chant
finalis	the final note of the chant
differentia	the melodic ending of psalms

[1]https://docs.python.org/3/library/csv.html
[2]https://pandas.pydata.org/

siglum	manuscript in which the chant is found
position	liturgical role of the chant
folio	page of the manuscript where the chant is found
sequence	order in which the chant is found in the folio
marginalia	clarification about the location of the chant
cao_concordances	references to older literature
feast_id	feast of the year during which the chant was sung
genre_id	genre of the chant, e.g. *antiphon*, *responsory*
office_id	office of the day during which the chant was sung, e.g. *Laudes*, *Vespers*
source_id	id of the manuscript in which the chant is found
melody_id	id of melody by which it can be found in the Cantus Index
drupal_path	URL of the chant on the Cantus database website
full_text	full text in a standardized spelling
full_text_manuscript	full text in the manuscript spelling
volpiano	transcription of the melody in volpiano format
notes	indexing notes

Table 3.1: List of database fields

3.3 User-defined data

The application enables user to upload their own dataset. The following table specifies the fields in the database. Validation of the file is not implemented; it is left up to the user to upload a valid file. The meaning of the individual fields is as described in the previous section unless said otherwise.

Column name	Type	Can be empty	Notes
none or Unnamed: 1	any	yes	the column will be dropped
id	string	yes	equivalent to *corpus_id* in the database
incipit	string	no	
cantus_id	string	yes	should be a valid Cantus ID
mode	string	yes	list of modes is in attachment
finalis	string	yes	
differentia	string	yes	
siglum	string	no	
position	string	yes	
folio	string	yes	
sequence	string	yes	
marginalia	string	yes	
cao_concordances	string	yes	
feast_id	string	yes	list of feasts is in attachment
genre_id	string	yes	list of genres is in attachment
office_id	string	yes	list of offices is in attachment

source_id	string	yes	
melody_id	string	yes	
drupal_path	string	yes	should be a valid URL
full_text	string	no	
full_text_manuscript	string	yes	
volpiano	string	no	has to be in Volpiano format
notes	string	yes	

Table 3.2: Fields in the user-uploaded CSV

3.4 Melody encoding

The melodies in the volpiano fields are encoded as strings of alphanumeric characters and dashes. These can be rendered as musical notation using the Volpiano font. Each character represents either a pitch, empty space, or other musical characters, such as a clef.

Volpiano was developed as a research tool optimized for databases and word processors. There are strict rules concerning the transcription, which leads to all Volpiano-encoded melodies having a standardized format. For example, the Volpiano protocol dictates that each transcription begins with a treble clef followed by three spaces, that words are separated by three spaces and syllables by two, and others. [3]

3.5 Data cleaning

As mentioned earlier, not the entire dataset is usable. Since we are analysing melodies, it is necessary that each data point contains information about the chant's melody.

As one of the application's main features is melody alignment, it is necessary that we only use those data points whose *volpiano* field is not empty. Moreover, some visualizations compare text length to melody length, therefore we require the data points to have both. Additionally, text and melody should be able to be aligned, i.e. they contain the same number of words and syllables.

Removing the data that does not adhere to the conditions, we are left with 13,397 usable data points.

[3] https://cantus.uwaterloo.ca/sites/default/files/documents/2.%20Volpiano%20Protocols.pdf

```
,id ,incipit ,cantus_id ,mode,finalis ,differentia ,siglum ,
    position ,folio ,sequence ,marginalia ,cao_concordances
    ,feast_id ,genre_id ,office_id ,source_id ,melody_id ,
    drupal_path ,full_text ,full_text_manuscript ,volpiano
    ,notes
621,chant_000622 ,A Christo de caelo vocatus
    ,001188,8,,1,F–Pn lat. 12044,3.,053v,5.0,,,
    feast_0287 ,genre_a ,office_m ,source_014 ,,http://
    cantus.uwaterloo.ca/chant/399542/,A Christo de
    caelo vocatus et in terram prostratus ex
    persecutore effectus est vas electionis ,A xpisto de
     caelo vocatus et in terram prostratus ex
    persequutore effectus est vas electionis ,1−−−g——g–
    kk—h——g——h—g——f—gh—g——g——g——hgf—g——gh—
    f—f7——g——f—h—j—kl—kj–klk——h—kj—hg——g——h
    ——gf—gh—h—g—g−−−4,
635,chant_000636 ,A Christo de caelo vocatus ,007123a
    ,1,,,US–CHNbcbl 097,01,035r ,2.0,,,feast_1321 ,
    genre_v ,office_m ,source_069 ,,http://cantus.
    uwaterloo.ca/chant/665425/,A Christo de caelo
    vocatus,,1−−h——h—hghgf——g——g—g——g—gh—g−−−,
645,chant_000646 ,A Christo de caelo vocatus ,007123a
    ,1,,,D–KA Aug. LX,01,125r ,2.0,,,feast_1321 ,genre_v ,
    office_m ,source_414 ,,http://cantus.uwaterloo.ca/
    chant/617583/,A Christo de caelo vocatus et in
    terram prostratus ex persecutore effectus est vas
    electionis ,A xpisto de celo vocatus et in terram
    prostratus ex persecutore effectus est vas
    electionis | ~Vere,1−−−h–h——hg—hg–gf7——g——g—g
    ——g—gh—g——g——g——hG—gh——gh—hjh—h——hf——gh
    —h—h—hk—h——h—hk—h——h——h——hg–hg—hf—fghg
    —hkh–hgfe—fgh–gf7−−−3,
667,chant_000668 ,A Christo de caelo vocatus ,007123a
    ,1,,,A–KN 1018,01,071v ,3.0,,,feast_1321 ,genre_v ,
    office_m ,source_265 ,,http://cantus.uwaterloo.ca/
    chant/293103/,A Christo de caelo vocatus et in
    terram prostratus ex persecutore effectus est vas
    electionis ,A christo de celo vocatus et in terram
    prostratus ex persecutore efectus est vas
    eleccionis ,1−−−dh——h—hg–hgg——g——g—g——g—gh—g
    ——g——g——hF—gh7——gh—hijh—h——h——gH—h—h—hk
    —h——h—hk—h——h——hghgfed–efg——gh—fe7—ef—
    fghgh—gf−−−3,
923,chant_000924 ,A deo praelectus∗,600006,8,,,NL–Uu
    406,,196r ,8.0,,,feast_1814 ,genre_r ,office_v2 ,
    source_573 ,,http://cantus.uwaterloo.ca/chant
    /497010/,A deo praelectus∗,A deo preelectus,1−−−g
    ——g—ffg——gge−−−3,
```

Figure 3.1: Example of a valid CSV file

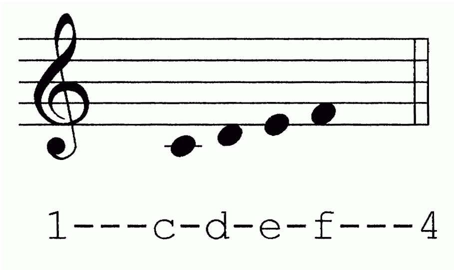


Figure 3.2: Example of volpiano as stored in the database and how it is rendered as musical notation. [Lacoste, 2012, Figure 2]

4. Multiple sequence alignment

Sequence alignment, in general, is a task whose purpose is to arrange two or more sequences with a common alphabet to identify similar and different regions within them. A set of sequences being *aligned* in this case can be understood as each being extended by spaces in such a way that if they are arranged in a matrix, each sequence occupying one row and each column containing one character, it can be seen what had to happen for one sequence to change into another on a character-by-character basis: insertion (or deletion), character substitution, or nothing if the characters in the given position are identical. A good sequence alignment algorithm is one that does not perform unnecessary insertions (deletions) or substitutions.

The problem of multiple sequence alignment is most studied in bioinformatics. Since DNA was first sequenced in the 1970s, there has been a need to compare various genomes to determine similarity. As organisms mutate and evolve, their DNA or RNA changes. Aligning their genomes reveals similar and different regions, which facilitates the tracking of these mutations and makes it possible to determine the order in which they happened.

However, the applications of sequence alignment are not limited to biology. Every task that makes use of determinig the similarity of some sequences, where the emphasis is put on finding regions where they do not diverge, can make use of the existing methods.

Melody alignment of Gregorian chant can be considered as such. As the tradition spread across Europe, each place changed some of the existing melodies by a little, thereby creating new melodies that can change further as they travel through time and space. This is akin to the mutations in DNA caused by environmental factors. Finding well-conserved regions in many instances of chant provides great insight into which parts of a melody are unlikely to change, and, on the other hand, which ones tend to vary a lot. It can also reveal the ancestors of a melody and the path which it traveled to transform into its final form. This is in line with the focus of philology shifting not to merely reconstructing an earliest layer of a text (with the unspoken assumption that this is the "real" text), but to map the entire tradition of text transmission and evolution, taking the later layers to be as valid within their cultural environment as the older layers.

In this chapter, we will first give the definition of the problem of sequence alignment. We will mention some important considerations, as well as theoretical limitations. Then we will provide an overview of the methods developed for bioinformatics that attempt to solve the problem. Finally, we will show how we applied the existing methods and technologies on Gregorian chant melodies.

4.1 The problem of sequence alignment

Assume that we have an alphabet $\mathcal{A}$ and a character σ such that $\sigma \notin \mathcal{A}$. Then let us have a set of sequences $S = \{s_1, s_2, ..., s_k\}$ with $s_i \in \mathcal{A}^{l_i}$. The output of a sequence alignment algorithm is the set of aligned sequences $A = \{a_1, a_2, ..., a_k\}$, where $a_k \in (\mathcal{A} \cup \{\sigma\})^L$, $L \geq l_i \, \forall i \in \{1, 2, ..., k\}$. Each original sequence s_i can be obtained from the aligned sequence a_i by removing all σ.

Given two aligned sequences a_i and a_j and an index $p \leq L$, we define the following operations:

- *Identity*: $(a_i)_k = (a_j)_k$
- *Insertion*: $(a_i)_k = \sigma \wedge (a_j)_k \in \mathcal{A}$
- *Deletion*: $(a_i)_k \in \mathcal{A} \wedge (a_j)_k = \sigma$
- *Substitution*: $(a_i)_k \in \mathcal{A} \wedge (a_j)_k \in \mathcal{A} \wedge (a_i)_k \neq (a_j)_k$

Each of the operations has an associated cost. The cost of substitution can further vary depending on which characters are being substituted. We can then define the overall cost of the alignment A in different ways, e.g. as the sum of costs over all triples (i, j, p) $\forall i, j \in \{1, 2, ..., k\}$ $\forall p \leq L$ or as the sum of costs for unordered pairs $\{i, j\}$ and indices p, in which case insertion and deletion are considered the same operation. The goal of a sequence alignment algorithm is to minimize the cost. There are other, more complicated ways of defining the cost function, and the performance of an algorithm is highly dependent on which one it uses.

4.1.1 Pairwise and multiple sequence alignment

Depending on the number of sequences to align, we distinguish between pairwise alignment for pairs of sequences and multiple sequence alignment for more than two. Despite the similarity in their outcomes, the two problems are fundamentally different from a computational perspective.

Pairwise alignment is relatively easy to solve. The Needleman-Wunsch algorithm, which is a dynamic programming algorithm, can find an optimal solution in the asymptotic time of $\mathcal{O}(mn)$, where m and n are the respective lengths of the sequences. This means that it is possible to find an optimal alignment even for longer sequences.

Needleman-Wunsch algorithm can be extended to more than two sequences. However, with each additional sequence, its complexity increases, and it quickly becomes impractical or even practically impossible to align multiple sequences this way. In fact, it has been proven that multiple sequence alignment is an NP-complete problem [Wang and Jiang, 2009]. It is therefore necessary to use various heuristics to generate alignments. Current algorithms do not aim at finding the optimal alignment; instead, they try to produce one that is good enough.

4.1.2 Local and global sequence alignment

There is a distinction to be made between local and global sequence alignment.

The problem description above is the definition of global alignment. Aligning sequences globally means aligning the entire sequences end-to-end. (This does not mean, however, that there cannot be gaps at the beginning or at the end of the generated alignment.) All characters from all sequences must be present in the final alignment. Global alignment is used to compare relatively similar sequences, such as protein homologues or versions of the same chant sung at different points in time.

On the other hand, the goal of local alignment is to find similar regions in divergent sequences, while the rest of the sequences is disregarded. The output of local alignment algorithms contains only a substring of both sequences. Local alignment is suitable for finding conserved patterns.

Both methods are useful in their own way. Local alignment provides a slightly different insight than global alignment, however, they can be combined to extract more information. In fact, the best current multiple sequence alignment algorithms use local alignment for pairs of sequences to generate a better overall global alignment. [Notredame, 2007]

4.2 Sequence alignment methods

The methods used to find sequence alignments depend on how many sequences there are and whether they should be aligned globally or locally. Dynamic programming can used for finding pairwise alignment, both local and global. For many sequences, other methods have been developed. They do not compute the optimal alignment, however, by using appropriate heuristics, their output is good enough.

4.2.1 Pairwise alignment: dynamic programming

Dynamic programming techniques are useful for pairwise alignment. The Needleman-Wunsch algorithm [Needleman and Wunsch, 1970] computes the global alignment of two sequences. A variation of the algorithm, the Smith-Waterman algorithm [Smith and Waterman, 1981], computes the local alignment of two sequences.

Needleman-Wunsch algorithm

The idea of the algorithm is to start with two empty sequences and subsequently add characters from either or both of the given sequences so as to obtain an optimal alignment in each step. Namely, suppose that we have two sequence prefixes A and B that have already been aligned optimally and their alignment gives a score of s. Furthermore, suppose that the next characters in the sequences are a and b, respectively. There are three possibilities:

- We append a and b to the respective prefixes. By doing so, we obtain the aligned sequence prefixes Aa and Bb.

```
Aa
Bb
```

- We append a to A and a gap to B. This way, we get the aligned prefixes Aa and B.

```
Aa
B-
```

- We append a gap to A and b to B. Now we have aligned the prefixes A and Bb.

```
A-
Bb
```

Each of the possibilities adds a value to the score s depending on what characters were added. To get the optimal alignment, we choose the one that yields the highest score. We then proceed to the next character, having two optimally aligned prefixes A' and B'. This leads to a recursive algorithm that can be formulated using dynamic programming.

Let us have two input sequences, A and B of lengths m and n with a common alphabet $\mathcal{A}$ and the gap character σ. Let us define the scoring function s as

$$s(a,b) = \begin{cases} 1 & \text{if } a = b \\ -1 & \text{if } a = \sigma \vee b = \sigma \\ -1 & \text{if } a \neq b \end{cases}$$

The algorithm initializes a matrix M of size $(m+1)*(n+1)$. The rows and columns represent the characters of A and B, respectively, except for the first row and the first column, which represent the beginning of a sequence or an empty sequence. That is to say, the row $M_{i,*}$ represents the character A_{i-1} for $i \geq 2$ and analogically, the column $M_{*,j}$ represents the character B_{j-1} for $j \geq 2$.

The algorithm iterates over the rows and columns of the matrix. In each step, it calculates the value of a cell $M_{i,j}$, provided that each of the cells $M_{i-1,j}$, $M_{i,j-1}$ and $M_{i-1,j-1}$ have been filled out, as

$$M_{i,j} = max \begin{cases} M_{i-1,j-1} + s(A_{i-1}, B_{j-1}) \\ M_{i-1,j} + s(A_{i-1}, \sigma) \\ M_{i,j-1} + s(\sigma, B_{j-1}) \end{cases}$$

In other words, the algorithm either aligns the two characters in positions $i-1$ and $j-1$, or it inserts a gap into sequence B, or it inserts a gap into sequence A, and chooses the version which yields the highest score.

The first row and column are apparently special cases, as there is no previous row or column. Therefore, for each cell in the first row or column, there is only one possible choice of score, which is equivalent to inserting a space.

After filling out the entire matrix, the algorithm then finds the optimal alignment by backtracking in the matrix. It starts in the bottom right cell of the matrix, which represents both sequences being aligned. In each iteration, it looks at the cells above, to the left and to the top-left of the current positions and chooses the highest score. If it moves top or left, it means that a gap was inserted. Moving diagonally represents a match or mismatch. By tracing its way back to the top left corner, the algorithm finds the alignment that yielded the highest score.

Let us show the algorithm on an example. Consider two sequences of nucleotide residues:

```
GATTA
GCATG
```

The matrix is initialized without anything filled out.

```
      G  C  A  T  G

G
A
T
T
A
```

We start filling out the matrix in the top left corner. Using the basic scoring scheme (+1 for a match, -1 for everything else), we insert a 0 to the beginning, and, since there is no top cell for the first row and no left cell for the first column, we add -1 to each subsequent cell, representing a gap insertion.

```
      G  C  A  T  G
   0 -1 -2 -3 -4 -5
G -1
A -2
T -3
T -4
A -5
```

The next cell to fill out is the one in the first `G` column and the first `G` row. We have three options:

- Move from the top, represents gap insertion for a score of -1. The final score would be $(-1) + (-1) = -2$.
- Move diagonally, represents match, giving a score of +1. The score in this case would be $0 + 1 = 1$.
- Move from the left, i.e. gap insertion. The score would be -2 as in the first case.

We choose the highest score, which in this case is a diagonal move representing a match. It follows intuition: we are now aligning `G` and `G`, matching them seems logical.

```
      G  C  A  T  G
   0 -1 -2 -3 -4 -5
G -1  1
A -2
T -3
T -4
A -5
```

Now let us look at the cell in the `C` column and the `G` row. Moving from the top yields -3, moving from the left yields 0 and moving diagonally yields -2, as it is a mismatch in this case. The highest of these scores is 0.

```
       G  C  A  T  G
    0 -1 -2 -3 -4 -5
 G -1  1  0
 A -2
 T -3
 T -4
 A -5
```

We continue filling out the table this way until it is complete.

```
       G  C  A  T  G
    0 -1 -2 -3 -4 -5
 G -1  1  0 -1 -2 -3
 A -2  0  0  1  0 -1
 T -3 -1 -1  0  2  1
 T -4 -2 -2 -1  1  1
 A -5 -3 -3 -1  0  0
```

As we have now calculated the scores for all prefixes, we can use backtracking to find the optimal alignment for the two sequences. Starting in the bottom right cell, we choose the cell (top, left or top-left) with the highest score. If multiple cells have the same score, we can choose either. The different alignments the cells represent are all optimal. The cell that we choose gives us the optimal alignment of the sequences before the last character was added. Depending on the direction by which we moved, we know whether this last character was a character of the sequence or a gap.

For example, consider the bottom right corner of the matrix.

```
    T  G
 T  1  1
 A  0  0
```

Starting in the bottom right, we can choose either the top or the top-left cell. Choosing the top-left one, i.e. moving diagonally, means that the last characters in the alignment were `A` and `G` for the respective sequences. We can find the alignment of the prefixes `GATT` and `GCAT` by backtracking from the top-left cell. By contrast, choosing the top cell means inserting a gap in the second sequence, and by backtracking from there we can find the alignment of `GATT` and `GCATG`.

Figure 4.1 shows a path that represents the alignment

```
G A - T T A
G C A T - G
```

$$
\begin{array}{ccccccc}
 & & G & C & A & T & G \\
 & 0 & -1 & -2 & -3 & -4 & -5 \\
G & -1 & 1 & 0 & -1 & -2 & -3 \\
A & -2 & 0 & 0 & 1 & 0 & -1 \\
T & -3 & -1 & -1 & 0 & 2 & 1 \\
T & -4 & -2 & -2 & -1 & 1 & 1 \\
A & -5 & -3 & 3 & -1 & 0 & 0
\end{array}
$$

Figure 4.1: A path representing an optimal alignment.

Smith-Waterman algorithm

The algorithm is similar to Needleman-Wunsch algorithm. As its purpose is to find an optimal local alignment, it does not penalize long regions of mismatches or gaps. Its purpose is to find regions with the most matches. The only difference from the Needleman-Wunsch algorithm is in how new matrix cells are filled out. Namely, when willing out a new cell, we use the formula

$$
M_{i,j} = max \begin{cases} 0 \\ M_{i-1,j-1} + s(A_{i-1}, B_{j-1}) \\ M_{i-1,j} + s(A_{i-1}, \sigma) \\ M_{i,j-1} + s(\sigma, B_{j-1}) \end{cases}
$$

In other words, all negative values in what would be the matrix from Needleman-Wunsch algorithm are replaced by 0.

4.2.2 Multiple sequence alignment: progressive methods

As has already been mentioned, it is essentially impossible to use dynamic programming to compute the alignment of more than two sequences. Therefore, other methods have been developed. The most successful ones appear to be the so-called progressive methods. In general, they use some heuristics to estimate a guide tree, which is a phylogenetic tree determining how close the sequences are to each other, and then they compute the actual multiple alignment following the order of this tree.

One of such methods is the Tree-based Consistency Objective Function for alignment Evaluation (T-Coffee) algorithm [Notredame et al., 2000]. Its most important contribution is its extended library generated from both local and global pairwise alignments of all pairs of input sequences. This library enables the algorithm to make fewer mistakes in the initial stages of the guide tree, as these errors propagate throughout the entire tree.

Another method is MAFFT (the name presumably coming from the acronyms for multiple alignment and fast Fourier transform) [Katoh et al., 2002]. The authors focused on finding alignments that are not only optimal, but also biologically correct. They developed a way of rapid identification of homologous regions between two sequences using FFT, and then used the better pairwise alignments to create a better multiple alignment.

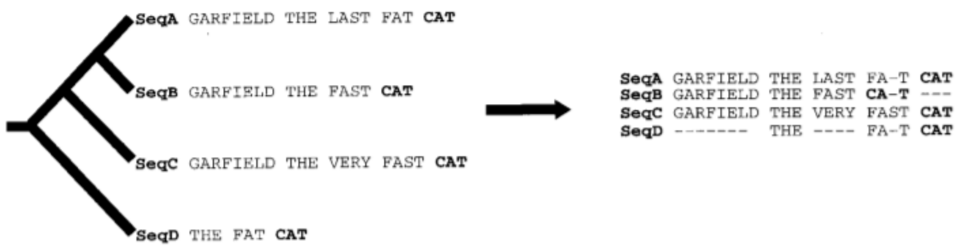


Figure 4.2: Misalignment of the word CAT using other progressive methods. [Notredame et al., 2000, Figure 2(a)]

T-Coffee

The T-Coffee algorithm consists of various stages. The first one is to compute pairwise alignments for all pairs of input sequences. Two primary libraries are generated, one for global and one for local alignments. Each can contain more than one alignment for each pair.

As some alignments tend to be more correct than others, weighting is then performed. The authors chose sequence identity of two aligned sequences as the weight of each of the aligned residues in the pair. For example, consider the sequences A `GARFIELD THE LAST FAT CAT` and B `GARFIELD THE FAST CAT`. If they are aligned as

```
GARFIELD THE LAST FAT CAT
GARFIELD THE FAST CAT ---
```

then their sequence identity is 88%, as there are two non-equal characters aligned (i.e. there is no gap penalty). Therefore, the weight of each residue pair $W(A(x), B(y))$, where $A(x)$ denotes the character x from sequence A, and analogically for $B(y)$, is equal to 88.

Figure 4.3: Primary library created from 4 sequences. [Notredame et al., 2000, Figure 2(b)]

The two primary libraries are then combined into one by combining all identical residue pairs into one entry and summing their weights, while pairs that are only present once are added with their original weight. Residue pairs that are not present in any alignment have an implicit weight of 0.

Although the information present in the primary library is sufficient to obtain a multiple alignment, it is computationally hard to do so. Instead, the authors chose to generate what they call an extended library using the weights in the primary library.

Library extension is performed by comparing each aligned residue pair with all the others. Consider a residue pair $(A(x), B(y))$ and a sequence C. The initial weight of the pair is then increased by $min(W(A(x), C(z)), W(C(z), B(y)))$, i.e. the minimum weight associated to the alignment of some residue $C(z)$ with both $A(x)$ and $B(y)$. This is done for all residues from all sequences. In practice, most of the weights will be 0, therefore the actual algorithm computes the weights more efficiently. Library extension in effect computes how consistent a residue-pair alignment is.

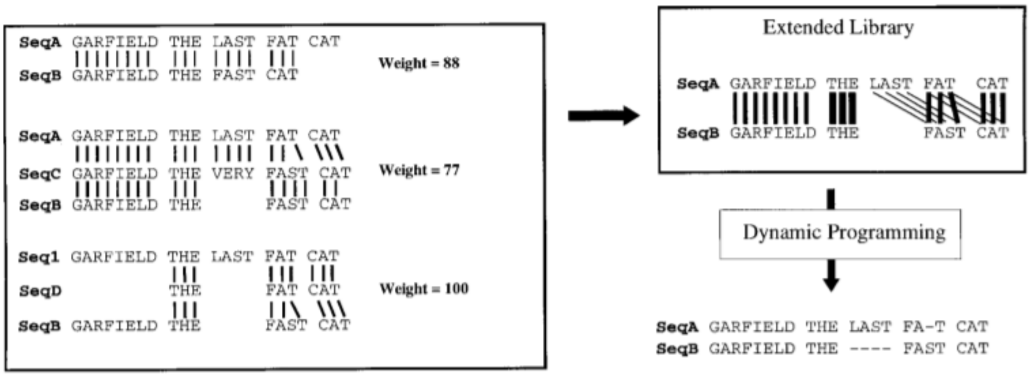


Figure 4.4: Extended library weights for two sequences and their alignment re-computed using these weights. [Notredame et al., 2000, Figure 2(c)]

Having obtained the consistency information from the extended library, it is now possible to create the guide tree and the final multiple alignment. Using a distance matrix between all the sequences, the tree is computed as follows. First, we align the closest two sequences using dynamic programming and the weights from the extended library. In each of the following steps, we either add a sequence to an already computed alignment, or we align the next closest sequences. We repeat this step until the alignment of all sequences is complete.

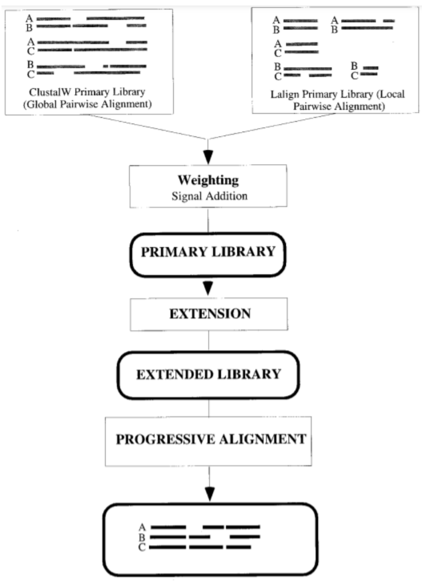


Figure 4.5: T-Coffee layout. [Notredame et al., 2000, Figure 1]

MAFFT

The MAFFT method is similar to T-Coffee in that it constructs a library of alignments which it then uses to create the final multiple alignment. The authors developed MAFFT to work in multiple modes, one of which is the progressive method as described above; the other one is the iterative refinement method, which allows for alterations of the multiple alignment obtained from the progressive method.

Pairwise alignment in MAFFT uses the fact that certain amino acids have more similar physico-chemical properties than others. Substitutions tend to preserve the overall structrure of a protein, therefore substitutions of similar amino acids are more frequent than those of different ones. The two properties the authors use are amino acid volume and polarity.

Let us define the correlation of the volume component between two amino acid sequences with the positional lag of k as

$$c_v(k) = \sum_{1 \leq n \leq N, 1 \leq n+k \leq M} \hat{v}_1(n)\hat{v}_2(n+k)$$

where N and M are the lengths of the sequences and $\hat{v}(a) = [v(a) - \bar{v}]/\sigma_v$ is the normalized volume value with $\bar{v}$ denoting the average volume of all the amino acids and σ_v their standard deviation.

Since the sequences tend to be equal in length, computing $c_v(k)$ using naive methods takes time proportional to N^2. However, applying Fast Fourier transform to the calculation reduces the time to $\mathcal{O}(NlogN)$.

We define the polarity component correlation $c_p(k)$ analogically.

The correlation between two amino acids is then expressed as

$$c(k) = c_v(k) + c_p(k)$$

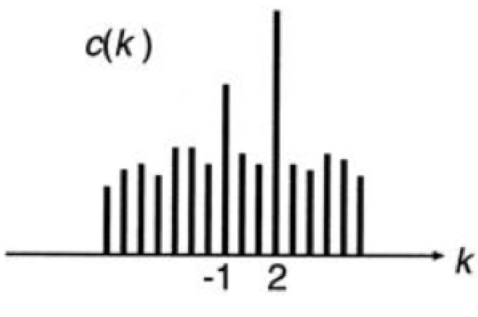


Figure 4.6: Plot of the correlation function $c(k)$. [Katoh et al., 2002, Figure 1A]

If we plot the function $c(k)$, there will be some peaks corresponding to the homologous regions of the two sequences, if there are any (Figure 4.6). However, the FFT analysis only gives us the positional lag of the regions, not their positions. To find the exact positions, we use a sliding window (of size 30 in the article) and calculate the degree of local homologies for the 20 highest peaks in the $c(k)$

function. If a segment exceeding a given homology threshold is identified, we label it as a homologous region.

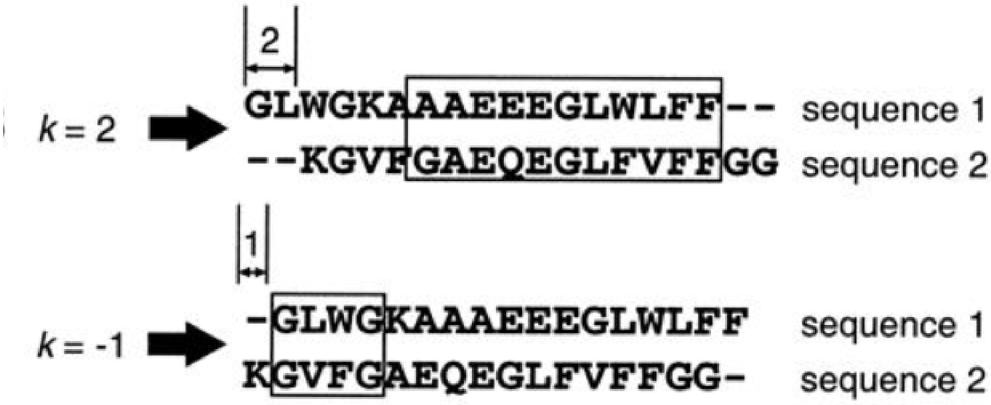


Figure 4.7: Finding homologous regions with different positional lags using a sliding window. [Katoh et al., 2002, Figure 1B]

After finding homologous segments between two sequences, their alignment is obtained by constructing a homology matrix $S \in R^{n*n}$, where n is the number of homologous segments. The cell S_{ij} is assigned a value depending on whether the i-th homologous segments of the first sequence corresponds to the j-th homologous segment of the second sequence. If so, the cell gets a value corresponding to the score of this segment; otherwise it has a value of 0. The optimal arrangment of homologous segments is then obtained using dynamic programming.

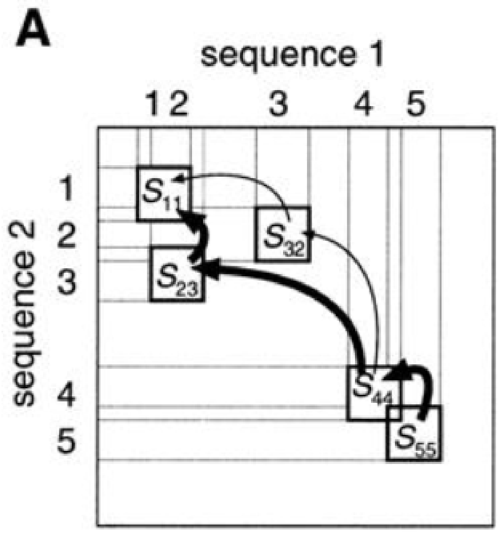


Figure 4.8: Dynamic programming applied on segment arrangement. [Katoh et al., 2002, Figure 2A]

Having arranged homologous segments, the algorithm then computes pairwise alignments for each pair of sequences. As in T-Coffee, it uses the alignments to create the primary and extended libraries, which are in turn used to estimate the guide tree and the final alignment. If MAFFT is configured to perform the progressive method, this alignment is the final one. Otherwise, if iterative refinement is allowed, MAFFT uses the weights from the library to adjust the alignment to get a more optimal one. However, iterative refinement is very slow from a relatively low number of sequences, therefore it is not suitable for a large dataset.

4.3 Sequence alignment for chant melodies

As we have discussed before, melodies of Gregorian chant are in many ways similar to biological sequences. The alphabet is different (instead of 20 amino acids in proteins or 4 nucleotide bases in DNA we have around 40 different characters used in melody representation) and the chants are usually shorter in length. The evolution of both biological and melodic sequences is guided by environmental factors such as migration to other places. Some segments tend to undergo a lot of modifications, while other remain relatively unchanged. Studying the alignment of chant melodies can reveal a lot of information about their relationship.

In the following sections, we describe three approaches to melody alignment used in our application: the naive text-based alignment, .

The first one is a naive approach. It merely aligns all chants to have the same number of words and each word to have the same number of syllables, filling the extra positions with gaps, if needed.

The other two approaches use MAFFT to perform a proper sequence alignment. They differ in what we are aligning. One of them aligns notes and other symbols as they are; each representing a certain pitch. The other one is more sophisticated: it does not use the absolute value of the pitches, but rather the intervals between the notes. This way, we can see segments that have merely been shifted by a certain interval.

4.3.1 Word-based alignment

The word-based approach serves as a basic tool to show the relationship between the text setting and melody. The algorithm simply extends all sequences to have the number of words equal to the maximum out of all sequences, and each word at a position i to have the number of syllables equal to the maximum out of all words at that position. The syllables are then filled with characters until there are any to use and the remaining ones are left empty.

Algorithm 1: Naive approach to alignment

input : n volpiano-encoded melodies and their texts
output: aligned melodies combined with their texts

$num\_words \longleftarrow max(length(sequence))$ for all sequences;
initialize n empty lists of length num_words;
for *word at index i* **do**
    $num\_syllables \longleftarrow max(length(word))$ for all words at position i;
    set the i-th position of each list to be of length $num\_syllables$;
    for *syllable at index j* **do**
        $num\_characters \longleftarrow max(length(syllable))$ for all syllables at position j of words at position i;
        set the j-th position of the i-th sublist of each sequence to be of length $num\_characters$;
    end
end
for *each melody* **do**
    for *i-th word of melody* **do**
        for *j-th syllable of word* **do**
            fill the j-th position of the i-th sublist of the list corresponding to the melody with the characters corresponding to the current syllable;
        end
    end
    combine the text with the melody;
end
return aligned melodies combined with their texts;

Figure 4.9: Chants of different genres aligned using the naive approach. The vertical lines represent word boundaries.

4.3.2 Multiple alignment using absolute pitches

For this task, we are using the MAFFT software[1]. MAFFT primarily works for sequences of amino acids and nucleotide bases, but it also has a limited support for non-biological sequences.

As the software uses the character `-` for marking gaps, it requires that the input sequences do not contain any, or they will be removed. Since melodies encoded as Volpiano use the character to mark ends of words and ends of syllables,

[1]https://mafft.cbrc.jp/alignment/software/

we need to perform some preprocessing. Namely, we replace all contiguous sequences `---` with the end-of-word marking symbol `~` and all contiguous sequences `--` with the end-of-syllable marker `|`. All remaining `-`s are removed, if there are any.

The sequences are then passed to MAFFT with the option `--text`, which indicates that the sequences are not biological. Additionally, we also pass the option `--reorder`, which returns the aligned sequences in order of similarity. We made this choice so as to facilitate the identification of related melodies.

After MAFFT performs the alignment, we retrieve the sequences and attempt to combine them with their text. We do this by splitting both the text and the melody into syllables and mapping them onto each other (Algorithm 2). However, this might not be possible. Due to errors in the original encoded melody (i.e. a missing or an extra `-`), we may have altered the structure of the chant in preprocessing. In case of such event, we remove the affected sequence from consideration and align the remaining sequences again.

Algorithm 2: Aligning melody and lyric

```
input  : volpiano: melody divided into words and those into syllables,
         and text: lyric divided analogically
output: volpiano and text aligned

set aligned_text_and_melody to be an empty list;
for word at index i do
    open new word in aligned_text_and_melody;
    if i-th word of text has more syllables than i-th word of melody then
        merge the extra syllables into the last one;
    else if i-th word of text has fewer syllables than i-th word of melody
     then
        extend the i-th word of text by empty syllables to match the
         volpiano word
    for syllable at index j of i-th word do
        combine text and melody of the j-th syllable of the i-th word and
         add it to the current word;
    end
    close current word;
end
return aligned_text_and_melody;
```

Once all sequences are successfully aligned without errors in text and melody combination, we can display them to the user. In the application, we use the Volpiano font, so as to maintain consistency. However, the sequences shown are not properly encoded following Volpiano protocols. First of all, we choose different representations of end of a word (here we use a bar line) and end of a syllable (a single space) so that the number of characters in each sequence remains equal and the alignment is visible. Second of all, after the alignment, MAFFT will have inserted `-`s to represent gaps. In proper Volpiano, these characters represent gaps between words, syllables, and neumes. However, here they just mean empty space.

Algorithm 3: Multiple alignment using absolute pitches

input : a set of volpiano-encoded melodies with their respective lyrics
output: aligned melodies combined with their lyrics

```
preprocess all melodies to a MAFFT-friendly format;
while melodies have not been aligned without error do
    run MAFFT on all melodies;
    for aligned melody do
        combine melody with its text;
        if melody and text cannot be combined then
            remove melody from list;
            remember to run the while loop again;
        end
    end
end
return list of aligned melodies combined with their texts;
```

Figure 4.10: Chants aligned using MAFFT on absolute pitches.

4.3.3 Multiple alignment using intervals

The general outline of the algorithm is the same as Algorithm 3. However, as we are not aligning absolute pitches, but rather intervals, we need to compute these intervals.

There are 18 possible pitches, which means 37 possible intervals (17 positive, 17 negative, and one without the change of pitch). We will represent the no-change interval with the character `a`, the positive intervals with the lowercase characters `b-t` and the negative intervals with the uppercase characters `B-T`. The characters `i` and `I` are not used for reasons explained later.

The interval representation is constructed from a Volpiano-encoded melody by replacing the note symbols with the appropriate interval symbol. The i-th note will be replaced with the symbol representing the interval $(i-1, i)$. As the first note has no predecessor, it is left as it is. The non-note symbols also remain unchanged.

Algorithm 4: Converting volpiano-encoded melody into interval representation

input : volpiano-encoded melody
output: interval representation of the input melody

```
interval_representation ⟵ "";
for character c in volpiano do
    if c is a non-note character then
        append c to interval_representation;
    else if c is a note-representing character and it is the first such one
      then
        append c to interval_representation;
        last_seen_note ⟵ c;
    else
        interval ⟵ (last_seen_note, c);
        i ⟵ symbol for interval;
        append i to interval_representation;
        last_seen_note ⟵ c;
end
return interval_representation;
```

We use the interval representations as inputs to MAFFT. That means that what MAFFT returns are the aligned sequences of intervals. To display them to the user properly, we need to decode the sequences.

Algorithm 5: Converting interval representation back to volpiano

output: interval representation of a melody
input : volpiano-encoded equivalent of the input melody

```
volpiano ⟵ "";
for character c in interval representation do
    if c is a non-interval character then
        append c to volpiano;
    else if c is a note-representing character and it is the first such one
      then
        append c to volpiano;
        last_seen_note ⟵ c;
    else if c is an interval-representing character and we have already
      seen the first note then
        note ⟵ last_seen_note+ interval represented by i;
        append note to volpiano;
        last_seen_note ⟵ note;
end
return volpiano;
```

The reason why we are not using `i` and `I` as symbols for an interval is that as we are looking at whether a character is a note or an interval or neither, if they represented an interval, we would choose the appropriate branch in Algorithm 5. However, marking an interval is not the only way how the character could have got there: `i` and `I` represent non-note elements in the Volpiano protocol. Therefore, we could mistakenly shift the melody by an interval, as there is no way of knowing which case it is.

The complete Algorithm 6, similar to Algorithm 3:

Algorithm 6: Multiple alignment using intervals

```
input : a set of volpiano-encoded melodies with their respective lyrics
output: aligned melodies combined with their lyrics

convert all melodies to interval representations;
preprocess all interval representations to a MAFFT-friendly format;
while melodies have not been aligned without error do
    run MAFFT on all interval representations;
    for aligned interval representation do
        convert interval representation to volpiano-encoded melody;
        combine melody with its text;
        if melody and text cannot be combined then
            remove melody from list;
            remember to run the while loop again;
        end
    end
end
return list of aligned melodies combined with their texts;
```

Figure 4.11: Chants aligned using MAFFT on intervals.

4.4 Conservation profile

By obtaining an alignment of a set of chants, we can obtain an interesting characteristic which is called the *conservation profile*. We define the conservation value of an element a at position i as

$$con_i(a) = \frac{\sum_{C:C_i=a} 1}{\sum_C 1}$$

where C denotes a chant and C_i the element at position i in the aligned chant C. In other words, the conservation value gives the ratio of elements a at position i out of all chants.

Conservation profile of aligned sequences depends on what approach we choose. Consider the last two chants shown in Figure 4.11. Each note is different, however, a person well-versed in musical theory immediately notices that they are only shifted by 5 semitones (except for the first note). If we consider only the intervals, they are almost the same melodies. However, if we consider the absolute pitches, they are totally different. Figure 4.12 and Figure 4.13 demonstrate these differences by displaying the chants' conservation profile.

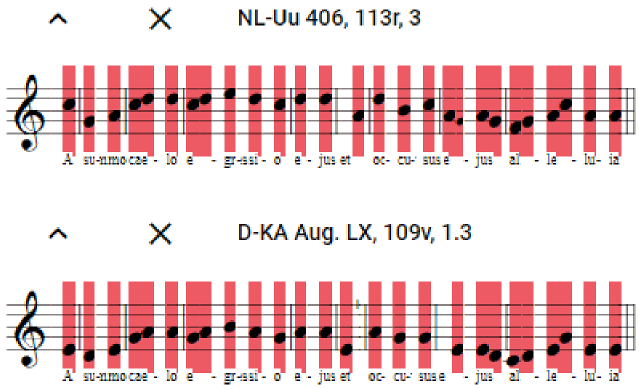


Figure 4.12: Conservation profile on chants aligned by pitch.

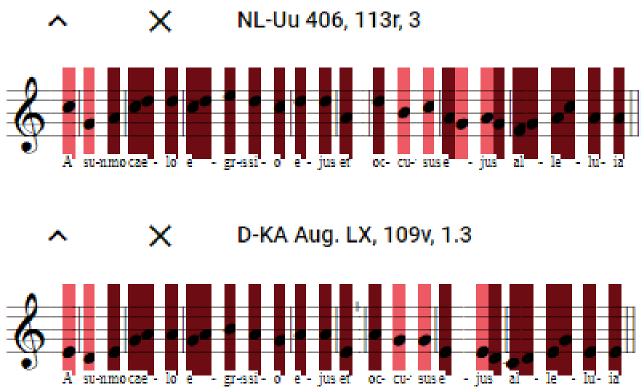


Figure 4.13: Conservation profile on chants aligned by intervals.

5. User documentation

In this chapter, we present an overview of the functionality of the application. The first section sketches out the overall functionality of the application. The following sections describe each feature in more detail.

5.1 Site map

5.2 Landing page

The landing page displays a list of chants from a specified source, which is by default set to the provided data source *CantusCorpus v0.2*. As we assume that the number of all selected chants can be large, we do not display all of them at once. Instead, the user can choose the number of chants displayed at once and paginate through the list to get to later ones.

The panel on the right side enables data source selection. To change data source (more than one can be selected at once), click the checkbox next to the data source name and click *save*. To hide or unhide the data source panel, click on the black *Data sources* button. The panel is available in the entire application.

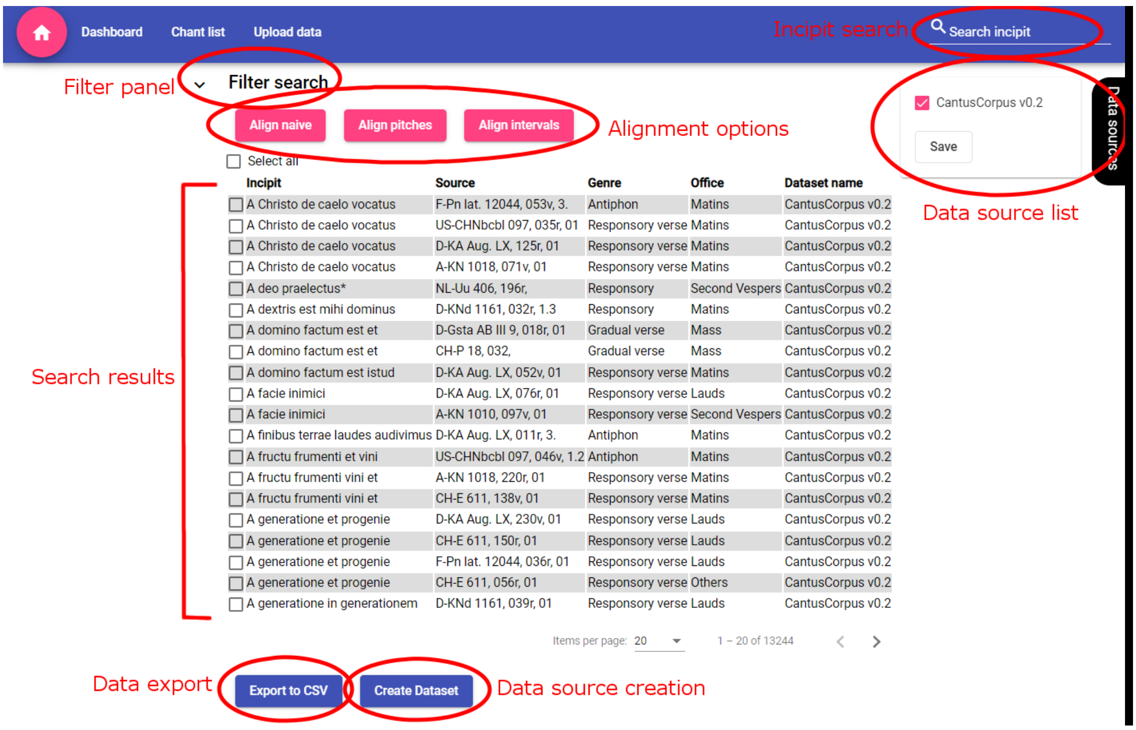


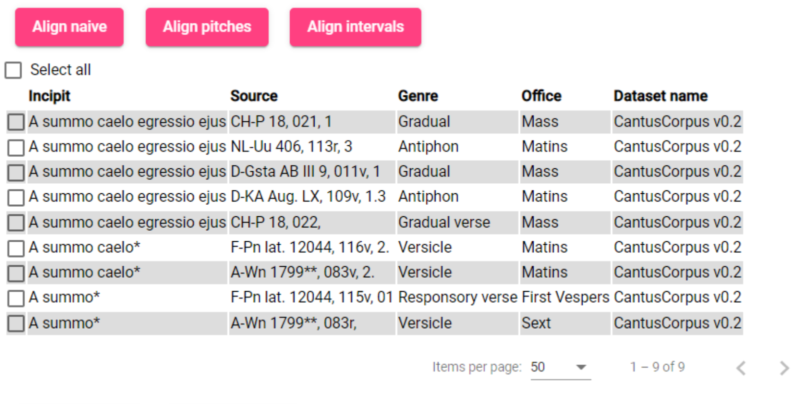
Figure 5.1: Landing page and its features.

5.2.1 List of available chants

5.2.2 Incipit search

The landing page provides the possibility to filter chants from the current selection by their incipit (the first few words of the chant). To do so, enter the desired string into the field in the top-right corner marked as *Search incipit* and press Enter. A set of chants that contain the query as a substring (not necessarily as a prefix) of their incipit will be displayed. Be aware that a misspelled query will not return the correct results.

Figure 5.2: Entering search query into the *Search incipit* box.



Incipit	Source	Genre	Office	Dataset name
A summo caelo egressio ejus	CH-P 18, 021, 1	Gradual	Mass	CantusCorpus v0.2
A summo caelo egressio ejus	NL-Uu 406, 113r, 3	Antiphon	Matins	CantusCorpus v0.2
A summo caelo egressio ejus	D-Gsta AB III 9, 011v, 1	Gradual	Mass	CantusCorpus v0.2
A summo caelo egressio ejus	D-KA Aug. LX, 109v, 1.3	Antiphon	Matins	CantusCorpus v0.2
A summo caelo egressio ejus	CH-P 18, 022,	Gradual verse	Mass	CantusCorpus v0.2
A summo caelo*	F-Pn lat. 12044, 116v, 2.	Versicle	Matins	CantusCorpus v0.2
A summo caelo*	A-Wn 1799**, 083v, 2.	Versicle	Matins	CantusCorpus v0.2
A summo*	F-Pn lat. 12044, 115v, 01	Responsory verse	First Vespers	CantusCorpus v0.2
A summo*	A-Wn 1799**, 083r,	Versicle	Sext	CantusCorpus v0.2

Figure 5.3: The result of the query. Notice that each incipit has the query as a substring.

5.2.3 Search filter

It is possible to filter the search results by their genre and office. The filter panel can be opened by clicking the down arrow next to the *Filter search* heading. By default, all genres and all offices are selected. To select or unselect some of them, click the checkbox next to their abbreviation. To select or unselect all genres or offices, click the checkbox next to *All* in the respective section.

The filter panel does not use the full names of the genres and offices, as in some cases, the name can be very long. Instead, it uses their abbreviations as used in Cantus Index. Attachments XXX and YYY provide a list of genres, resp. offices and their abbreviations.

5.2.4 Chant detail

By clicking on a chant's incipit in the search results list, the user is redirected to a page which displays the chant's melody with the text in full, as well as other

Figure 5.4: Search filter panel.

information of interest. It shows the chant's Cantus ID, by which it can be found in the Cantus Database. It also shows a URL which points to the corresponding entry on the Cantus website. These fields are valid for all chants from the default *CantusCorpus v0.2* dataset, but there is no guarantee of their correctness in user-uploaded data.

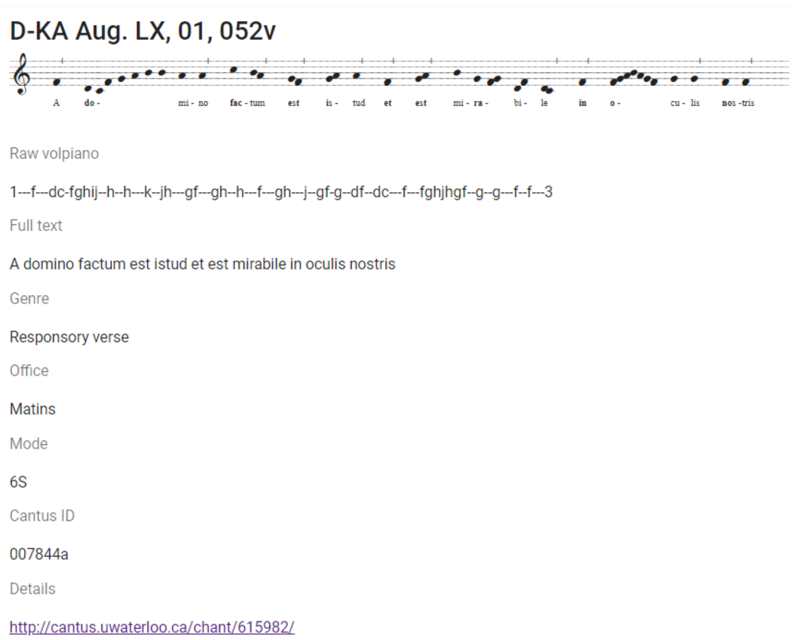


Figure 5.5: Page with the detail of the selected chant.

5.2.5 Data export

The user may wish to export the results of their search into a CSV file. In that case, select the desired chants by checking the checkbox next to their incipit (or *Select all*) and click the *Export to CSV* button at the bottom of the page. A file with the name *dataset.csv* will be automatically downloaded.

```
id,corpus_id,incipit,cantus_id,mode,finalis,differentia,siglum,position,folio,sequence,marginalia,cao_concordances,feast_id,genre_id,office_id,source_id,melody_id,
drupal_path,full_text,full_text_manuscript,volpiano,notes,dataset_name,dataset_idx
0,chant_000622,A Christo de caelo vocatus,001188,8,,1,F-Pn lat.
12044,3.,053v,5.0,,,feast_0287,genre_a,office_m,source_014,,http://cantus.uwaterloo.ca/chant/399542/,A Christo de caelo vocatus et in terram prostratus ex
persecutore effectus est vas electionis,A xpisto de caelo vocatus et in terram prostratus ex persequutore effectus est vas electionis,1---g---g-kk--h---g---h--
g---f--gh--g---g---g---hgf--g---gh--f--f7---g---f--h--j--kl--kj-klk---h--kj--hg---g---h---gf--gh--h--g--g---4,,CantusCorpus v0.2,0
1,chant_000636,A Christo de caelo vocatus,007123a,1,,,US-CHNbcbl
097,01,035r,2.0,,,feast_1321,genre_v,office_m,source_069,,http://cantus.uwaterloo.ca/chant/665425/,A Christo de caelo vocatus,,1--h---h--hghgf---g---g--g---g--
gh--g---,,CantusCorpus v0.2,0
2,chant_000646,A Christo de caelo vocatus,007123a,1,,,D-KA Aug. LX,01,125r,2.0,,,feast_1321,genre_v,office_m,source_414,,http://cantus.uwaterloo.ca/chant/617583/,A
Christo de caelo vocatus et in terram prostratus ex persecutore effectus est vas electionis,A xpisto de celo vocatus et in terram prostratus ex persecutore
effectus est vas electionis | ~Vere,1---h-h---hg--hg-gf7---g---g--g---g--gh--g---g---g---hG--gh---gh--hjh--h---hf---gh--h--h--hk--h---h--hk--h---h---h---hg-hg--
hf--fghg--hkh-hgfe--fgh-gf7---3,,CantusCorpus v0.2,0
3,chant_000668,A Christo de caelo vocatus,007123a,1,,,A-KN 1018,01,071v,3.0,,,feast_1321,genre_v,office_m,source_265,,http://cantus.uwaterloo.ca/chant/293103/,A
Christo de caelo vocatus et in terram prostratus ex persecutore effectus est vas electionis,A christo de celo vocatus et in terram prostratus ex persecutore
efectus est vas eleccionis,1---dh---h--hg-hgg---g---g--g---g--gh--g---g---g---hF--gh7---gh--hijh--h---h---gH--h--h--hk--h---h--hk--h---h---hghgfed-efg---gh--fe7--
ef--fghgh--gf---3,,CantusCorpus v0.2,0
4,chant_000924,A deo praelectus*,600006,8,,,NL-Uu 406,,196r,8.0,,,feast_1814,genre_r,office_v2,source_573,,http://cantus.uwaterloo.ca/chant/497010/,A deo
praelectus*,A deo preelectus,1---g---g--ffg---gge---3,,CantusCorpus v0.2,0
```

Figure 5.6: First few lines of the exported CSV file.

5.3 Data source creation

The application is not limited to the data present by default. The user can either create a data source from their search results, or upload completely new data.

5.3.1 Create from search

To create a data source from the search results, check the checkboxes next to the desired chants and click the *Create Dataset* button at the bottom of the page. The user will be prompted for the data source name. If no name is entered, the name of the data source will be *undefined*. After confirming the name, the data source list will be automatically updated with the newly-created one.

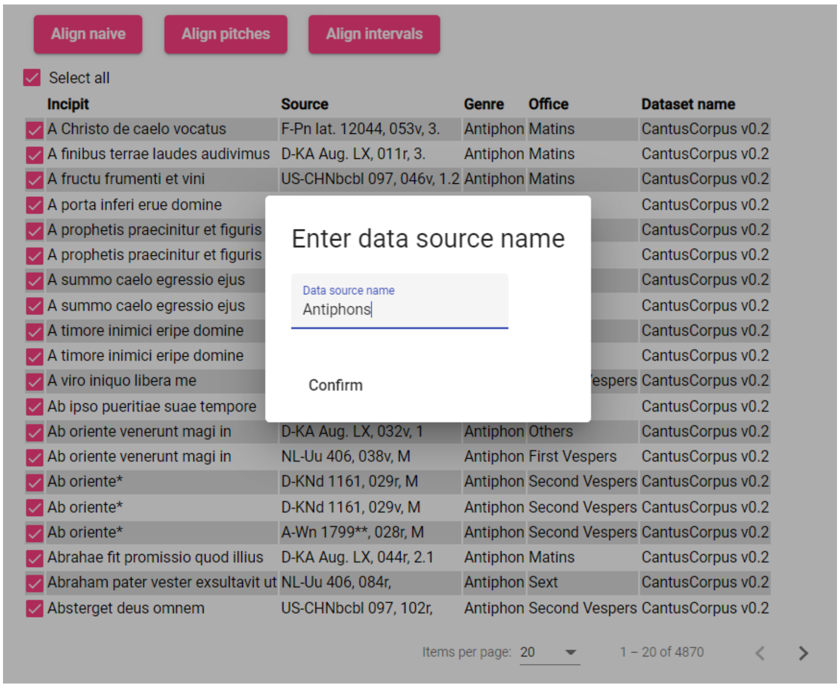


Figure 5.7: Prompt for entering the name of the data source appears after clicking the *Create Dataset* button.

5.3.2 User data upload

To upload new data, navigate to the *Upload data* page in the top panel. Select the file to upload and enter the data source name. The file should be in CSV

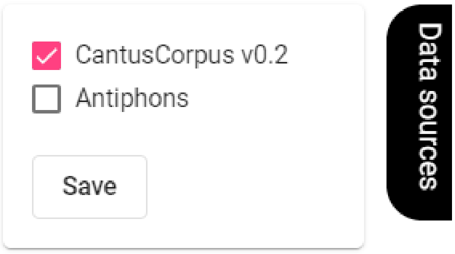


Figure 5.8: The newly created data source automatically appears in the data source list.

format and its contents are specified in section XXX. Click *Upload* to upload the file. After the file is processed, the data source list will be automatically updated. The exported file created in section XX is suitable for upload.

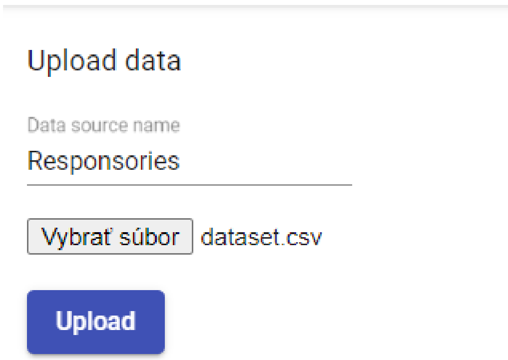


Figure 5.9: The upload form with the file *dataset.csv* selected for upload. The new data source will have the name *Responsories*.

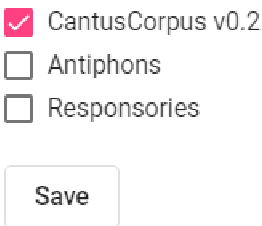


Figure 5.10: The uploaded data source automatically appears in the list.

5.4 Alignment

It is possible to select a set of chants and have them be aligned using one of the methods described in section XXX. The alignment option is selected by clicking on one of the buttons above the chant list after selecting the chants to be aligned. After doing so, the user is redirected to a page where the alignment result is shown. In some cases, the alignment make take up to a minute to complete.

The alignment result is shown as a list of melodies. The melodies are either arranged in the same order as they were displayed in the chant list (in the case of naive alignment) or in the order of similarity.

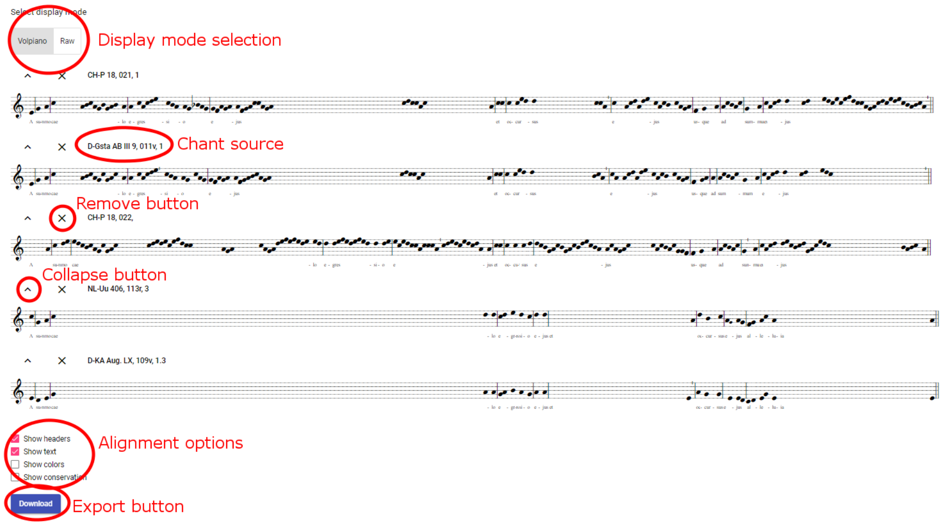


Figure 5.11: Result of the alignment.

To show more information about each melody, the user can click on the chant source button by which each chant is identified. The same information as when navigating to the chant's page is displayed. To hide the detail, click on the chant source button again.

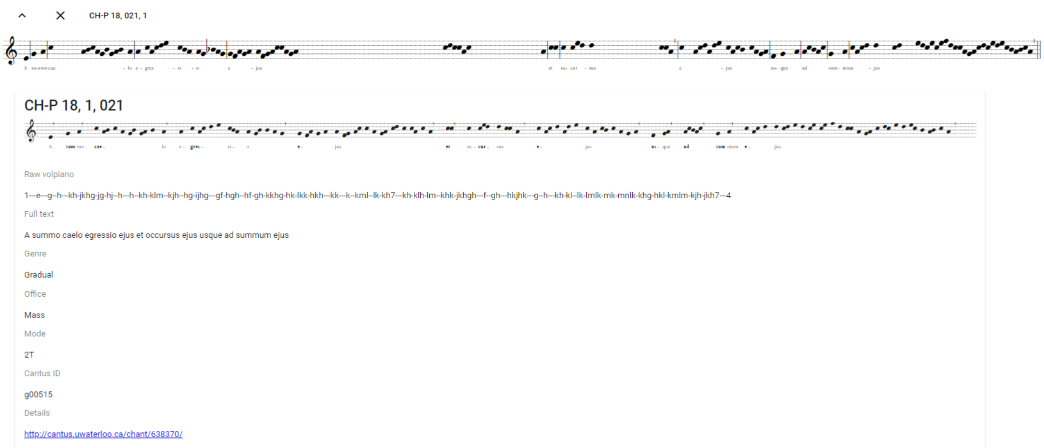


Figure 5.12: Aligned melody with the chant information displayed.

If the user wishes to rearrange the displayed melodies, simply click anywhere close to a melody and drag it to another position.

5.4.1 Collapsing and deleting alignment

After the alignment is shown, the user can choose to collapse the alignment (i.e. hide the notes) by clicking on the *Collapse* button in each melody's header or to remove it completely by clicking on the *Remove button*. The chant can be uncollapsed by clicking on the same button. However, removing the chant entirely is permanent. To get the removed chant back, the page needs to be reloaded.

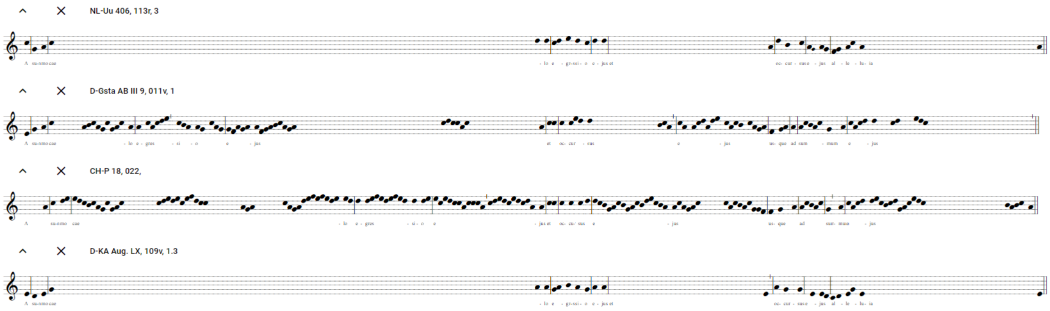


Figure 5.13: Alignment after dragging and dropping the last melody. Compare to Figure 5.11.

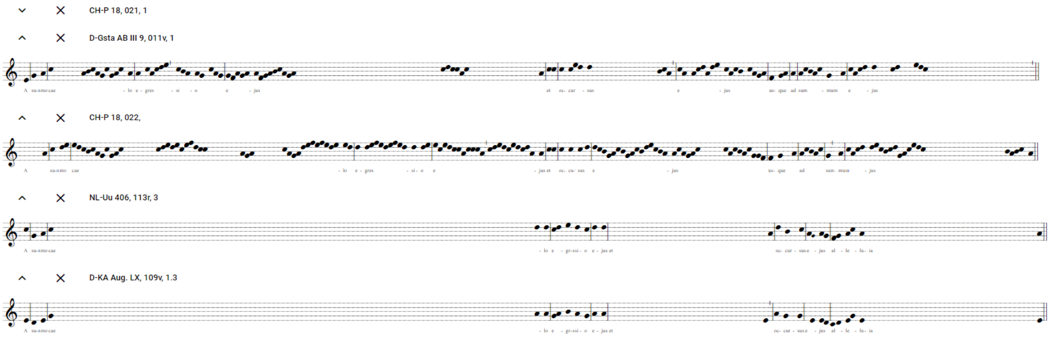


Figure 5.14: The first chant is collapsed.

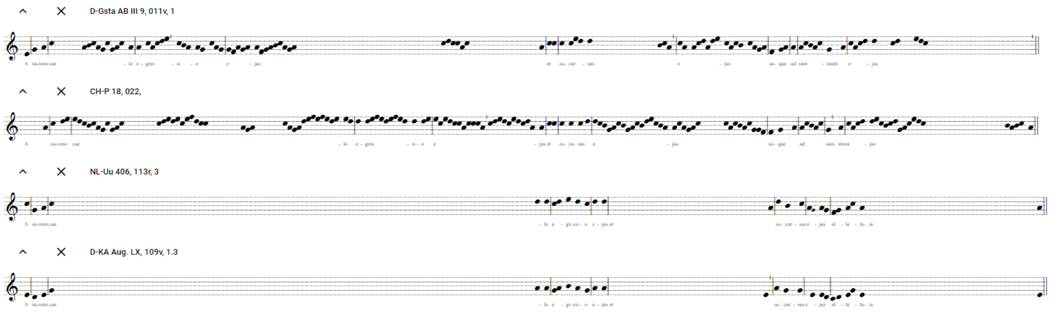


Figure 5.15: The first chant is removed. Compare to Figure 5.11.

5.4.2 Alignment display options

Once the chants have been aligned, there exist several options to better show how they are aligned.

As the Volpiano font does not have characters of equal width, the notes that are actually aligned might not be shown as aligned. To see the actual alignment, the user can change the display mode above the aligned melodies to *Raw*, which shows the melodies as the encoded string in a monopaced font.

To see the aligned sequences better, it is possible to remove the headers and the text independently by unclicking the corresponding boxes in the alignment options.

It is possible to see how similar certain regions are. The first option is to color each note based on its absolute pitch. In the case of interval alignment, this

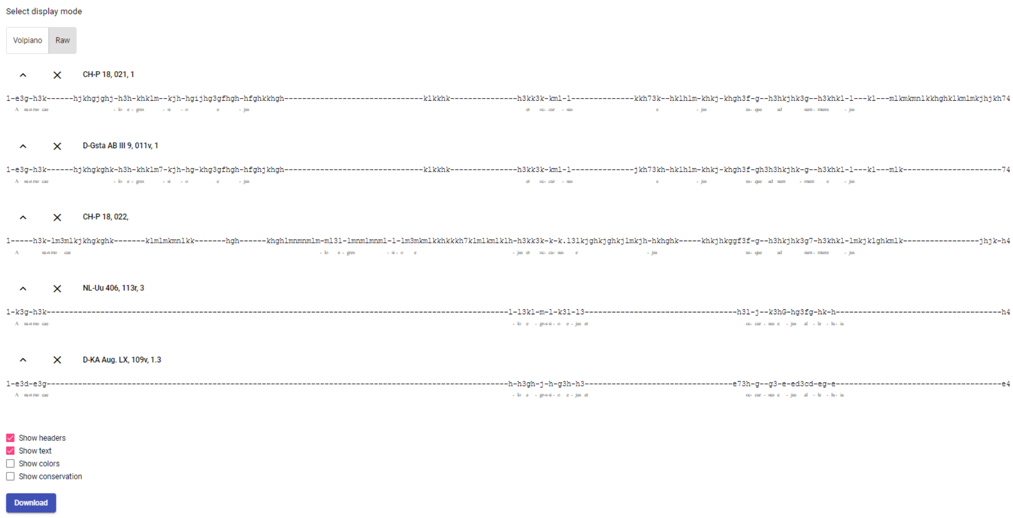


Figure 5.16: Aligned melodies shown as the raw string.

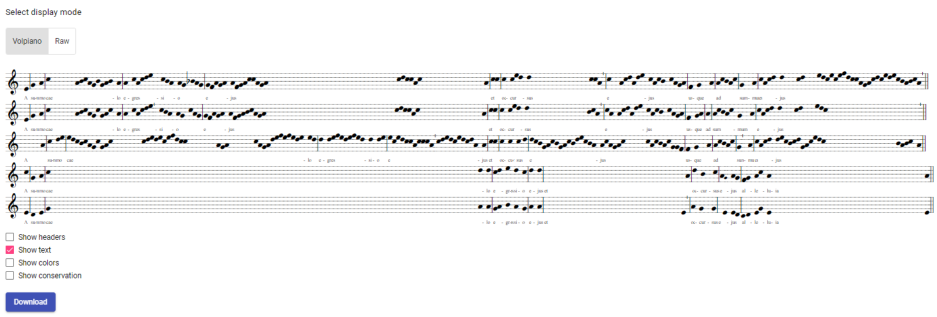


Figure 5.17: The aligned melodies shown without headers.

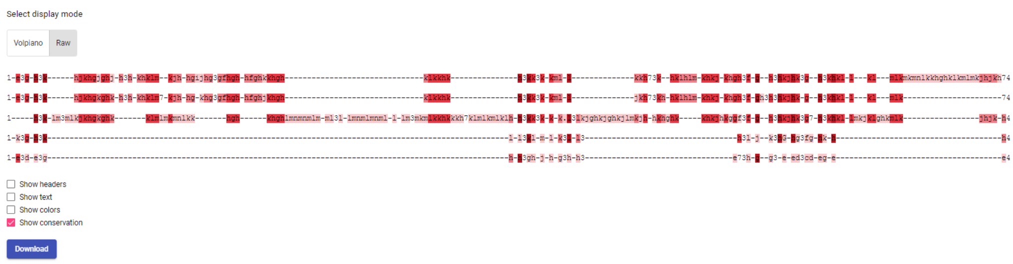


Figure 5.18: The aligned melodies shown in *Raw* mode without headers and text.

option may show different colors even though the intervals are the same.

The other option is to show the conservation profile, i.e. the proportion of the other chants sharing the same note or interval at the given position. This option will show conservation profile corresponding to the intervals in the interval alignment option. Darker colors mean the position is more conserved, while lighter colors mean less conserved. Using this option, it is possible to see how similar entire regions are.

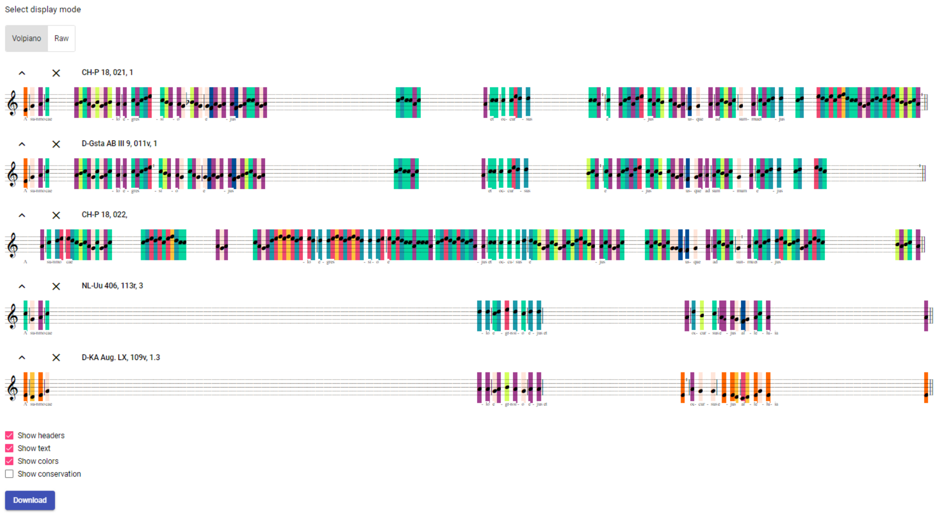


Figure 5.19: Aligned melodies with each note colored a different color.

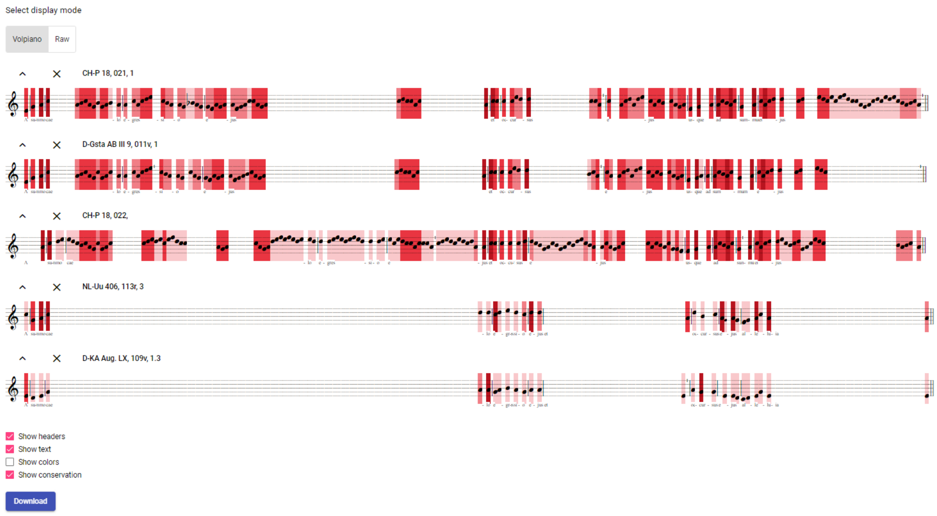


Figure 5.20: Aligned melodies with their conservation profile shown.

5.4.3 Alignment export

To export the aligned melodies, click the *Download* button at the bottom of the page. The format of the file will be suitable to pass directly into MAFFT again. Each melody's identifier is the string stored in the *drupal_path* field.

The melodies in the exported file will be either encoded as symbols corresponding to Volpiano characters in the case of naive and pitch alignment, or as symbols corresponding to intervals in the case of interval alignment.

Only the unremoved melodies will be present in the file. Their order will be the same as the current order on the page.

5.5 Dashboard

5.6 Known issues

6. Development documentation

The application consists of three principal parts that ensure its proper functioning: the database, back end and front end. Their interactions are outlined in Figure 6.1.

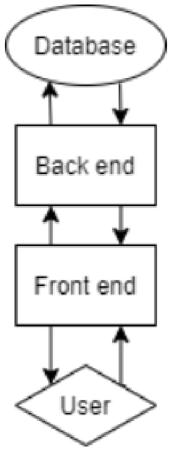


Figure 6.1: A high-level overview of the application's architecture.

The database stores the data used by the application. The data it contains do not impact the functionality of the application. It can be added to or removed from, as long as the added data follows the schema. It is even possible to have multiple databases and use the one that is currently needed.

The back end is the part of the application that takes care of the internal logic. It pulls data from the database and adds new ones. It performs calculations that would be too compute-intensive for the front end. The back end is interacted with via its API: anyone can make an appropriate http request to a specified URL and the back end returns the corresponding response.

The front end is the user-facing interface. Its main function is to display data returned by the back end. It can also perform some lighter calculations. The user interacts with the front end via a web browser.

In the following sections, we will describe the proper functioning of each part in more detail.

6.1 Data

In this application, we use SQLite[1] as our database technology. It is a lightweight and self-contained database engine, which makes it easy to work with. As our use case is not data-heavy, we chose this technology since we prefer flexibility over performance.

Our database consists of a single table `chant`. The fields of one entry are described in Section 3.2. Its schema is shown in Figure 6.2.

The last two fields, `dataset_name` and `dataset_idx` determine to which data source the row belongs. Data source is the main grouping criterion for data

[1] https://www.sqlite.org/

```
CREATE TABLE IF NOT EXISTS "chant" (
"id" INTEGER,
  "corpus_id" TEXT,
  "incipit" TEXT,
  "cantus_id" TEXT,
  "mode" TEXT,
  "finalis" TEXT,
  "differentia" TEXT,
  "siglum" TEXT,
  "position" TEXT,
  "folio" TEXT,
  "sequence" REAL,
  "marginalia" TEXT,
  "cao_concordances" REAL,
  "feast_id" TEXT,
  "genre_id" TEXT,
  "office_id" TEXT,
  "source_id" TEXT,
  "melody_id" TEXT,
  "drupal_path" TEXT,
  "full_text" TEXT,
  "full_text_manuscript" TEXT,
  "volpiano" TEXT,
  "notes" TEXT,
  "dataset_name" TEXT,
  "dataset_idx" INTEGER
);
CREATE INDEX "ix chant id"ON "chant" ("id");
```

Figure 6.2: Schema of the *chant* table.

in this application. The user may select multiple data sources for comparison. The expected amount of data in each data source is in the order of thousands of entries. Because of this, we chose to store the data in a single database, as opposed to e.g. having one database for each data source.

6.2 Back end

For the back end, we use the Django framework.[2] Django is a Python web framework enabling developers to create web applications quickly, having abstracted away the most tedious of web development aspects.

Django programs consist of one or more logical units called *apps*. An *app* is a self-contained program providing some related functionality. Thanks to Django's modularity, an *app* can be plugged in into many projects without the need to always rewrite it. Our application only uses one *app*, `melodies`, which provides REST API to other programs.

The core functionality of the application is contained in the submodule `core`. The submodule is used by the *app* `melodies`.

[2]https://www.djangoproject.com/

6.2.1 Core

`core` is the Python submodule that contains essential functionality ensuring the proper behavior of the application, e.g. the implementation of alignment, MAFFT integration, and others. Figure 6.3 outlines the classes provided in the submodule and how they interact. The classes' functionality is described below.

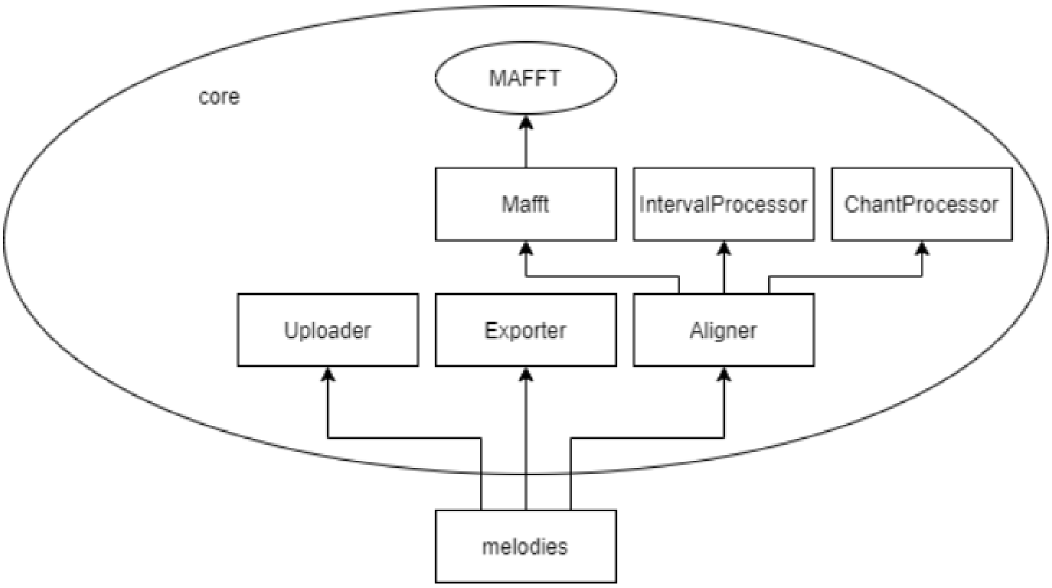


Figure 6.3: Outline of the classes contained in *core* and their interaction. An arrow from A to B indicates that A depends on B.

Exporter. The class contains one method, `export_to_csv`. It takes as an argument a list of chant IDs contained in the database. It returns a response with a file with the corresponding database entries formatted as CSV.

Uploader. The class defines one method, `upload_csv`. Its arguments are a *pandas*[3] dataframe, containing data columns as described in Section 3.2, and a dataset name. The method inserts the data into the database, ensuring that no duplication of IDs occurs.

ChantProcessor. The class contains functionality for working with Latin texts and with melodies encoded as Volpiano. The method `get_syllables_from_text` takes a string of Latin text and returns the text syllabified: the returned value is a list, each of whose elements represents a word of the text, and each word is a list of strings representing the syllables in that word. To syllabify the text, we use the module `cltk`,[4] which facilitates working with classical languages, including Latin.

Some parts of the application require the encoded melody to contain no `-` characters. Instead, they assume that ends of words are marked with a tilde (`~`) and ends of syllables with a pipe (`|`). The method that converts proper Volpiano-encoded melody to this processed format is the method `insert_separator_chars`. The equivalent of syllabifying text for melodies is `get_syllables_from_volpiano`. It takes a melody encoded as a processed Volpiano string (with `~` and `|` instead

[3]https://pandas.pydata.org/
[4]http://cltk.org/

of `-`.). It assumes that the melody contains a clef at the first position and a bar line at the last position, which is the proper way of encoding melodies according to the Volpiano protocols.[5] The method returns a list whose elements represent words, and each word is a list of strings representing syllables (equivalent to the syllabifying method for text). Examples of using these functions are shown in Figure 6.4 and Figure 6.5.

```
>>> text = "A Christo de caelo vocatus et in terram prostratus ex persecutore effectus est vas electionis"
>>> syllabified_text = ChantProcessor.get_syllables_from_text(text)
>>> syllabified_text
[['A'], ['Ch', 'ris', 'to'], ['de'], ['cae', 'lo'], ['vo', 'ca', 'tus'], ['et'], ['in'], ['ter', 'ram'], ['pros', 'tra',
'tus'], ['ex'], ['per', 'se', 'cu', 'to', 're'], ['ef', 'fec', 'tus'], ['est'], ['vas'], ['e', 'lec', 'ti', 'o', 'nis']]
```

Figure 6.4: Example of dividing Latin text into syllables using *ChantProcessor*.

```
>>> volpiano = "1---g---g-kk--h---g---h--g---f--gh--g---g---g---hgf--g---gh--f--f7---g---f--h--j--kl--kj-klk---h--kj--hg-
--g---h---gf--gh--h--g--g---4"
>>> processed_volpiano = ChantProcessor.insert_separator_chars(volpiano)
>>> processed_volpiano
'1~g~gkk|h~g~h|g~f|gh|g~g~g~hgf|g~gh|f|f7~g~f|h|j|kl|kjklk~h|kj|hg~g~h~gf|gh|h|g|g~4'
>>> syllabified_volpiano = ChantProcessor.get_syllables_from_volpiano(processed_volpiano)
>>> syllabified_volpiano
[['g'], ['gkk', 'h'], ['g'], ['h', 'g'], ['f', 'gh', 'g'], ['g'], ['g'], ['hgf', 'g'], ['gh', 'f', 'f7'], ['g'], ['f', 'h
', 'j', 'kl', 'kjklk'], ['h', 'kj', 'hg'], ['g'], ['h'], ['gf', 'gh', 'h', 'g', 'g']]
```

Figure 6.5: Example of dividing a melody encoded as Volpiano into syllables using *ChantProcessor*.

The method `get_stressed_syllables` takes a Latin string as input and returns a list whose elements represent words, where word is a list of 0s and 1s, a 0 representing an unstresed syllable and a 1 representing a stressed one. To calculate the stressed syllables, we use the module `cltk`. However, its stress recognition is not completely accurate, therefore this method may also return incorrect results. The function is shown in use in Figure 6.6.

```
>>> text = "A Christo de caelo vocatus et in terram prostratus ex persecutore effectus est vas electionis"
>>> stressed_text = ChantProcessor.get_stressed_syllables(text)
>>> stressed_text
[[0], [1, 0], [1], [1, 0], [1, 0, 0], [1], [1], [1, 0], [0, 1, 0], [1], [0, 0, 1, 0, 0], [0, 1, 0], [1], [1], [0, 0, 1, 0
, 0]]
```

Figure 6.6: Example of finding the stressed syllables of a Latin text using *Chant-Processor*.

IntervalProcessor. The class provides tools to work with the interval representation of a melody. It contains two methods, `transform_volpiano_to_intervals` and `transform_intervals_to_volpiano`. The first one takes a string representing a melody (it can be encoded either as dictated by the Volpiano protocol, or processed to have its words separated by `~`s and syllables by `|`s) and returns the same melody represented as intervals. The latter is its reverse: it takes a string representing a melody encoded by intervals and returns the Volpiano representation. Figure 6.7 shows how these functions are used. Notice that after transforming the interval representation of the original melody back to Volpiano encoding, we obtain the original melody.

Mafft. This class provides the interface for working with the MAFFT software.[6] The software must be installed on the computer that runs the application.

[5] https://cantus.uwaterloo.ca/sites/default/files/documents/2.%20Volpiano%20Protocols.pdf
[6] https://mafft.cbrc.jp/alignment/software/

```
>>> volpiano = "1---g---g-kk--h---g---h--g---f--gh--g---g---g---hgf--g---gh--f--f7---g---f--h--j--kl--kj-klk---h--kj--hg-
--g---h---gf--gh--h--g--g---4"
>>> volpiano_to_intervals = IntervalProcessor.transform_volpiano_to_intervals(volpiano)
>>> volpiano_to_intervals
'1---g---a-da--C---B---b--B---B--bb--B---a---a---bBB--b---ab--C--a7---b---B--c--b--bb--BB-bbB---C--cB--BB---a---b---BB--b
b--a--B--a---4'
>>> intervals_to_volpiano = IntervalProcessor.transform_intervals_to_volpiano(volpiano_to_intervals)
>>> intervals_to_volpiano
'1---g---g-kk--h---g---h--g---f--gh--g---g---g---hgf--g---gh--f--f7---g---f--h--j--kl--kj-klk---h--kj--hg---g---h---gf--g
h--h--g--g---4'
```

Figure 6.7: Transforming a Volpiano-encoded melody into an interval representation and back.

On Windows systems, it is assumed that it is installed in a WSL instance.[7] On other systems, it has to be installed natively.

An instance of the class encapsulates the action of aligning one set of data. It needs to have a specified input file, and, optionally, an output file, which can be set via the methods `set_input` and `set_output`. If the user wishes to run the alignment with other options as specified in MAFFT's documentation, it can be done using the method `add_option`. The method `add_volpiano` can optionally be used if the user does not want to provide their own file (however, its location must still be defined) or wishes to add another sequence to align. MAFFT is only run by calling `run_process`. The result of the alignment is retrieved using the methods `get_aligned_sequences` and `get_sequence_order`, which return the aligned sequences and their headers, respectively, in the order of similarity.

Aligner. The class defines the methods to compute the three types of alignment, as described in Section 4.3: `alignment_syllables`, which performs the word-based alignment; `alignment_pitches`, computing the alignment of pitches; and `alignment_intervals`, which return the alignment of intervals. All of them take as an argument just the list of IDs of the chants to be aligned and return a dictionary easy to work with in HTML templates.

6.2.2 REST API

The `melodies` app provides API that offers other programs a way of interacting with it. It is done in the form of web requests to a specific URL. The requests can be either GET or POST requests. GET requests do not contain any additional data; the response returned always depends only on the state of the application. On the contrary, POST requests contain data that the application takes into account when constructing the response.

Table 6.1 summarizes the API provided by the application. The API is described into more detail below.

Endpoint	Method	POST fields	Description
api/chants/	POST	`dataSources`, `incipit`, `genres`, `offices`	Returns a list of chants filtered by their data source, incipit, genre and office
api/chants/<pk>	GET		Returns the chant with the ID <pk>

[7] https://docs.microsoft.com/en-us/windows/wsl/

api/chants/sources	GET		Returns a list of all data sources currently present in the database with their name and index
api/chants/upload/	POST	`file`, `name`	Uploads `file` to the database as a new data source and gives it the specified `name`
api/chants/export/	POST	`idsToExport`	Returns a CSV file containing the chants with the specified IDs
api/chants/create-dataset/	POST	`idsToExport`, `name`	Creates a new data source from the chants with the specified IDs and gives it the specified `name`
api/chants/align/	POST	`idsToAlign`, `mode`	Takes chants with the specified IDs and aligns them according to `mode`

Table 6.1: API of back end.

api/chants/ The POST request must contain the fields as specified in Table 6.1. `dataSources` is a list of indices of allowed data sources as stored in the database. `incipit` is a string that must be present as a substring in the *incipit* field in each of the returned entries. `genres` and `offices` are lists of strings, where each string is an identifier of a genre or an office, as seen in Attachment XX. Each returned entry must have its genre be an element of `genres` and its office be an element of `offices`. The response is a list of all entries in the database satisfying these constraints.

api/chants/<pk> The application looks for a chant with the ID `pk` in the database. It returns an error if no such entry exists. If it does, it returns a JSON object with the following keys: *db_source*, which is the entry as stored in the database, *json_volpiano*, which is the chant processed in such a way that its melody and text are easy to display in an HTML template, and *stresses*, a list of stressed syllables corresponding to the chant's text, if it can be computed.

api/chants/sources The response is a JSON object with one key, `sources`. Its value is the list of pairs representing the data sources currently present in the database. The first value of each pair is a data source index and the second one is its name.

api/chants/upload/ The POST request must contain a CSV file as specified in Section XX in the field `file` and a string `name`. The data in the file will be uploaded directly to the database as a new data source with the given name. Some values may be changed before inserting the data in the database, namely the columns *id*, *dataset_idx*, and *dataset_name*, if the file contains them. The response is a JSON object with the keys *name*, the name of the new data source, and *index*, the index assigned to it.

api/chants/export/ The POST request body must contain the field `idsToExport`. It is a list of integers representing the IDs of the chants to be exported. If an ID does not correspond to any chant in the database, the ID will be excluded. The response contains a CSV file with its entries being the corresponding chants, formatted as outlined in Section XX.

api/chants/create-dataset/ The POST request must contain a list of chant IDs in the field `idsToExport` and a string in the field `name`. The chants that correspond to the IDs will be duplicated in the database, given new IDs and assigned to a new data source with the name as specified in the field `name`. The response is the same JSON object as in the *upload* API, containing the keys *name* and *index*, the name and index of the newly created data source.

api/chants/align/ The POST request contains two fields. `idsToAlign` is the list of chant IDs we want to align. `mode` is the mode of alignment we chose. Its values can be *syllables*, which is the word-based alignment, *full*, the alignment on pitches, or *intervals*, the alignment on intervals. If the value is not one of those, the request will return an error. The response is a JSON object with the following keys: *chants* is an object representing the aligned melodies along with their texts, easily renderable by an HTML template. *errors* contains a list of chant IDs of the chants that could not be aligned for various reasons. *successes* contains an object that describes the chants that have been successfully aligned: *sources* are their original sources (i.e. siglum, folio and position), *ids* the IDs as stored in the database, *volpianos* the melodies as given by the alignment algorithm, and *urls* the values in the *drupal_path* field of the database.

6.3 Front end

For the front end, our application uses the Angular framework.[8] It is a component-based front-end framework. An Angular application consists of components, which are small, relatively self-contained units exposing some functionality of the application (both appearance and behavior), and services, providing means of sharing and manipulating data, e.g. the interaction with the back end. Both components and services should only contain a small, logically contained set of functionality. This makes the application safe and scalable.

This application contains several components and services that interact with each other. Figure 6.8 shows a diagram of the front end architecture. The elements in rectangles are components. Orange background, *AppComponent*, represents the component encapsulating the entire application. Blue background represents components that are always present on the page. An dotted arrow from component A to component B indicates that component A has component B in its HTML template. The ellipses represent services. An arrow from a component or service A to service B means that element A uses service B.

The following sections describe each service and component in more detail.

6.3.1 Services

IncipitService. The service stores the value of the incipit search query which is used when pulling data from the database. It provides two methods: `setIncipit` which changes the current value of the search query, and `incipit` which returns a *BehaviorSubject*[9] storing the value.

[8]https://angular.io/

[9]https://www.learnrxjs.io/learn-rxjs/subjects/behaviorsubject

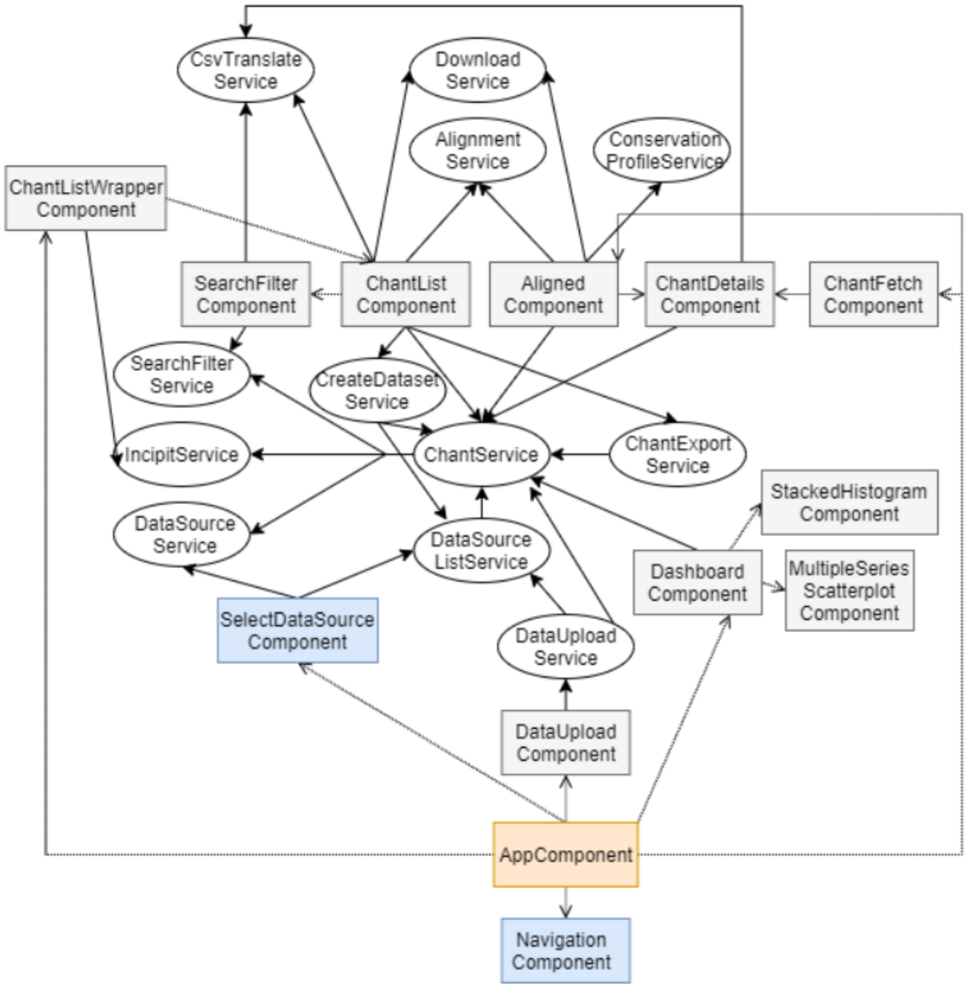


Figure 6.8: The architecture of the front end.

SearchFilterService. The service stores an object containing a list of allowed genres and a list of allowed offices which will be used for filtering when retrieving data from the database. It provides two methods, `setFilterSettings`, which changes the current value, and `getFilterSettings`, returning a *BehaviorSubject* with the value.

DataSourceService. The service stores a list of indices of data sources we are currently using. The stored value persists throughout browser sessions. The service has two methods, `setSourceList`, which changes the value, and `getSourceList`, which returns a *BehaviorSubject* with the list.

ChantService. This service is the main interface for working with the back end. It uses back end's API as described in Section 6.2.2 to obtain the desired data. The method `getChant`, taking a single number as an argument, sends a request to the back end to retrieve the chant with the ID equal to the argument. The method `loadData` uses the current values stored int *IncipitService*, *SearchFilterService*, and *DataSourceService* to obtain all chants adhering to the constraints; the method `getList` provides the object where these results are stored. The method `getAlignment` takes a list of chant IDs and an alignment mode and makes a request to the back end to calculate the corresponding alignment. `getDataSources` obtains all data sources currently present in the database. `exportChants` takes as an argument a list of chant IDs and the corresponding API call returns a CSV file with their database entries. `createDataset` creates a new data source from the chant IDs and name provided to it. All arguments to these functions are passed as a part of a *FormData*[10] object.

DataSourceListService. The service stores a list of the data sources in the database as of the time of the last call to the back end's API. The list is accessed by calling `getAllSources`. The method `refreshSources` retrieves the current list of data sources from back end.

CreateDatasetService. The only method `createDataset` takes as an argument a list of chant IDs and a data source name and ensures that it is sent to back end to be inserted into the database. Additionally, it calls *DataSourceListService*'s `refreshSources` to obtain the most recent list of data sources.

DataUploadService. The service's only method `uploadData` takes two arguments, a CSV file and a data source name. It passes these to *ChantService* to be inserted into the database and assures that the current data source list is correct by a call to *DataSourceListService*.

ChantExportService. The service provides one method, `exportChants`. Its argument is a list of chant IDs to be exported. The method returns a CSV file with these chants.

CsvTranslateService. The service provides means to obtain a genre's or an office's full description given its identifier as stored in the database. The descriptions are sourced from CSV files provided in CantusCorpus. They can be accessed via the methods `getGenre` and `getOffice`, both taking the identifier of the genre or office as an argument. The service also contains the method `getAllValues`, which takes as an argument either the string *genres* or *offices*, and returns an object containing all values of the given type.

DownloadService. The service contains a single method, `download`. It takes two arguments: `blob`, which is the content of a file to be downloaded, and `filename`, the name of the file. Calling the method triggers the download of the file.

AlignmentService. The service stores the IDs of chants we want aligned and the mode in which they should be aligned. Both of these values persist throughout browser sessions. The IDs can be changed or obtained by calling the setter and getter on `idsToAlign`. The mode can be set via the method `setMode`, which returns 0 if the provided mode is correct and 1 otherwise; and retrieved with the method `getMode`.

ConservationProfileService. Given a list of aligned melodies, as computed by the back end, the service's method `calculateConservationProfile` calculates the conservation value for each position in each melody. It returns it in a form easy to work with in HTML templates.

6.3.2 Components

AppComponent. This component is the one always present on the page, encapsulating other components. It defines the global styles of the entire page. The component displays *SelectDataSourceComponent* and *NavigationComponent*, which

[10]https://developer.mozilla.org/en-US/docs/Web/API/FormData

means they are always visible on the page. It further displays components depending on the current URL, as indicated by the `router-outlet` tag.

NavigationComponent. The component provides easy navigation to the app's main pages. It also enables the user to enter a query for searching among the chants' incipits.

SelectDataSourceComponent. The component exposes the interface to change the current selection of data sources.

ChantLlistWrapperComponent. The component encapsulates *ChantListComponent*. It reads the incipit search query from the current URL and passes it to *IncipitService*.

ChantListComponent. The component displays a list of chants from the current search results. It also provides the possibility to filter search results, to export a selection of chants, and to create a new data source.

SearchFilterComponent. The component enables the user to select which genres and offices should be included in search results.

AlignedComponent. The component displays results of alignment. It also provides the possibility to export the results to a file.

ChantFetchComponent. The component encapsulates *ChantDetailsComponent*. It reads chant ID from URL and passes it to *ChantDetailsComponent*.

ChantDetailsComponent. The component displays the relevant information about a selected chant.

DashboardComponent. The component displays several visualization of data from the current search selection.

StackedHistogramComponent. The component contains a histogram of data passed to it. We use the library *d3js*[11] for the plot creation.

MultipleSeriesScatterplotComponent. The component shows a scatter plot of data passed to it. Again, we use the library *d3js*.

DataUploadComponent. The component displays the interface for uploading a file to the database.

[11] https://d3js.org/

Conclusion

Bibliography

Olivier Berten and contributors. Gregobase: A database of gregorian scores. URL https://gregobase.selapa.net/.

Dimitrios Bountouridis, Daniel G. Brown, Frans Wiering, and Remco C. Veltkamp. Melodic similarity and applications using biologically-inspired techniques. *Applied Sciences*, 7(12), 2017. ISSN 2076-3417. doi: 10.3390/app7121242. URL https://www.mdpi.com/2076-3417/7/12/1242.

Bas Cornelissen, Willem Zuidema, and John Ashley Burgoyne. Studying large plainchant corpora using chant21. In *7th International Conference on Digital Libraries for Musicology*, DLfM 2020, page 40–44, New York, NY, USA, 2020a. Association for Computing Machinery. ISBN 9781450387606. doi: 10.1145/3424911.3425514. URL https://doi.org/10.1145/3424911.3425514.

Bas Cornelissen, Willen Zuidema, and John Ashley Burgoyne. Mode classification and natural units in plainchant. 2020b. URL https://program.ismir2020.org/poster_232.html.

Michael Scott Cuthbert and Christopher Ariza. music21: A toolkit for computer-aided musicology and symbolic music data. pages 637–642, 2010.

David Hiley. *Gregorian Chant*. Cambridge Introductions to Music. Cambridge University Press, 2009. doi: 10.1017/CBO9780511807848.

David Huron. The melodic arch in western folksongs. *Computing in Musicology*, pages 3–23, 1996.

David Huron and Joshua Veltman. A cognitive approach to medieval mode: Evidence for an historical antecedent to the major/minor system. *Empirical Musicology Review*, 1(1):33–55, 2006. URL https://doi.org/10.18061/1811/24072.

Kazutaka Katoh, Kazuharu Misawa, Kei-ichi Kuma, and Takashi Miyata. Mafft: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic Acids Research*, 30(14):3059–3066, 07 2002. ISSN 0305-1048. doi: 10.1093/nar/gkf436. URL https://doi.org/10.1093/nar/gkf436.

Debra Lacoste. The cantus database: Mining for medieval chant traditions. *Digital Medievalist*, 7, 2012. URL http://doi.org/10.16995/dm.42.

Debra Lacoste and Jan Koláček. Cantus index: Online catalog for mass and office chants. URL http://cantusindex.org/.

Debra Lacoste, Jan Koláček, Terence Bailey, and Ruth Steiner. A database for latin ecclesiastical chant - inventories of chant sources. URL https://cantus.uwaterloo.ca/.

Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. ISSN 0022-2836. doi: https://doi.org/10.1016/0022-2836(70)90057-4. URL https://www.sciencedirect.com/science/article/pii/0022283670900574.

Cédric Notredame. Recent evolutions of multiple sequence alignment algorithms. *PLoS Comput Biol*, 3, 2007. URL https://doi.org/10.1371/journal.pcbi.0030123.

Cédric Notredame, Desmond G Higgins, and Jaap Heringa. T-coffee: a novel method for fast and accurate multiple sequence alignment11edited by j. thornton. *Journal of Molecular Biology*, 302(1):205–217, 2000. ISSN 0022-2836. doi: https://doi.org/10.1006/jmbi.2000.4042. URL https://www.sciencedirect.com/science/article/pii/S0022283600940427.

Saebyul Park, Taegyun Kwon, Jongpil Lee, Jeounghoon Kim, and Juhan Nam. A cross-scape plot representation for visualizing symbolic melodic similarity. pages 423–430, 2019.

Rebecca Shaw. Differentiae in the cantus manuscript database: Standardization and musicological application. In *Proceedings of the 5th International Conference on Digital Libraries for Musicology*, DLfM '18, page 38–46, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450365222. doi: 10.1145/3273024.3273028. URL https://doi.org/10.1145/3273024.3273028.

T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981. ISSN 0022-2836. doi: https://doi.org/10.1016/0022-2836(81)90087-5. URL https://www.sciencedirect.com/science/article/pii/0022283681900875.

Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 2009. URL http://doi.org/10.1089/cmb.1994.1.337.

David D. Wickland. Evaluating melodic similarity using pairwise sequence alignments and suffix trees. Master's thesis, The University of Guelph, 9 2017.

List of Figures

List of Tables

List of Abbreviations

A. Attachments

A.1 First Attachment