Stockholms universitet

An Interactive Database for Hilbert Vectors of Graded Artinian Algebras

En interaktiv databas för Hilbertvektorer av graderade artinska algebror

Georgii Kimberg

Handledare: Samuel Lundqvist

Examinator: Marc Hellmuth

Inlämningsdatum: 2025-05-25

Kandidatuppsats i datalogi

Bachelor Thesis in Computer Science

Abstract

The Hilbert function of a graded Artinian algebra encodes key combinatorial invariants, yet popular repositories such as the OEIS fall short in systematically organizing Hilbert vectors alongside their algebraic parameters. Moreover, despite OEIS containing over 360,000 integer sequences, it lacks both the volume and structural precision needed to support comprehensive research into Hilbert vectors. To address this gap, we have developed a web service featuring a Django-based Python backend and a lightweight JavaScript/AJAX frontend. Our service currently hosts over 700,000 Hilbert sequences, manually generated for four families of complete intersections using Macaulay2, and stored in a SQLite database indexed by value to support rapid subsequence queries. Users can add, remove, or update sequences via a built-in administrative interface. We establish the correctness of our subsequence-search algorithm through an inductive proof and demonstrate that preprocessing requires $O\big(\sum_i L_i\big)$ time, while each query executes in $O(1)$ or $O(k)$ time, with k denoting the subsequence length. The developed repository already significantly surpasses OEIS in terms of the volume of Hilbert vectors, hosting approximately twice as many sequences specifically tailored for research in graded Artinian algebras, while also remaining extensible for future expansions and improvements. By combining full CRUD functionality with efficient, parameter-aware search, this service provides researchers in commutative algebra with an interactive and extensible repository of Hilbert vectors.

Sammanfattning

Hilbertfunktionen för en graderad artinsk algebra kodar viktiga kombinatoriska invarianter, men populära databaser såsom OEIS saknar en systematisk organisation av Hilbertvektorer tillsammans med deras algebraiska parametrar. Trots att OEIS innehåller över 360 000 heltalssekvenser saknas både volymen och den strukturella precision som krävs för omfattande forskning om Hilbertvektorer. För att fylla denna lucka har vi utvecklat en webbtjänst baserad på Django (Python) med ett lättviktigt JavaScript/AJAX-gränssnitt. Vår tjänst innehåller i dagsläget över 700 000 Hilbertsekvenser, manuellt genererade för fyra familjer av kompletta skärningar med hjälp av Macaulay2, och lagrade i en SQLite-databas indexerad efter värde för snabba delsökningar. Användare kan lägga till, ta bort eller uppdatera sekvenser via ett inbyggt administrationsgränssnitt. Vi bevisar korrektheten hos vår algoritm för delsökning med hjälp av induktion, och visar att förbehandlingen kräver $O\big(\sum_i L_i\big)$ tid, medan varje sökning utförs på $O(1)$ eller $O(k)$, där k är längden på delsekvensen. Det utvecklade arkivet överträffar redan OEIS betydligt när det gäller antalet Hilbertvektorer, med ungefär dubbelt så många sekvenser som är specifikt anpassade för forskning kring graderade artinska algebror, samtidigt som det förblir öppet för framtida utvidgningar och förbättringar. Genom att kombinera full CRUD-funktionalitet med effektiv parameterbaserad sökning erbjuder tjänsten forskare inom kommutativ algebra en interaktiv och utbyggbar databas för Hilbertvektorer.

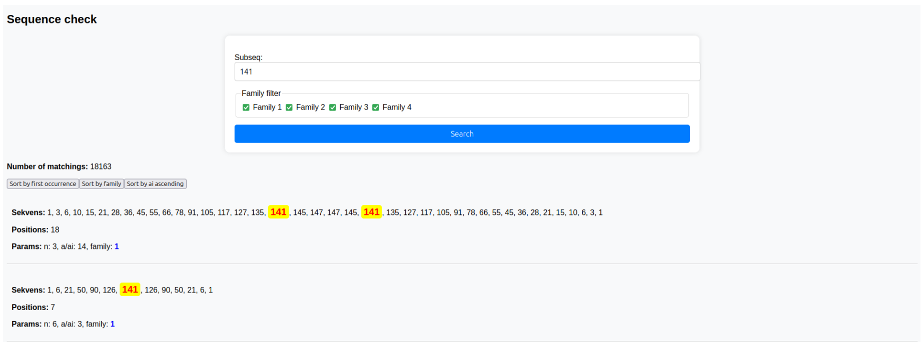
Contents

1 Introduction

At present, the On-Line Encyclopedia of Integer Sequences (OEIS) contains more than 360,000 records of integer sequences and is an indispensable reference resource for mathematicians and discrete mathematics specialists[1]. It enables rapid identification of known sequences by their initial terms and provides extensive literature citations and cross-references across diverse fields. Nevertheless, OEIS currently includes only a limited collection of specialized sequences arising from computations involving graded Artinian algebras—specifically, the so-called *Hilbert vectors*. Moreover, OEIS does not facilitate the exploration of entire families of Hilbert vectors, nor does it provide an efficient way to filter and visualize sequences belonging to a particular algebraic family. For researchers focused on graded Artinian algebras, this makes it challenging to obtain a comprehensive overview or to compare parameters within specific families (see Section 4 for further details on Hilbert vectors and graded Artinian algebras). Additionally, our database already includes approximately twice as many Hilbert vectors as the total number of integer sequences currently listed in OEIS, highlighting a significant improvement in scope and applicability for algebraic research.
The goal of this thesis is to develop a specialized web-based service analogous to OEIS, specifically tailored for collecting, storing, and exploring integer Hilbert vectors of graded Artinian algebras. The envisioned workflow enables users to select one or several algebraic families from an available list, input a numerical subsequence, and quickly obtain a structured and filterable list of all matching Hilbert vectors along with their associated algebraic parameters.
For example, a typical user query might look like this:

- **Selected family**: Family 1; Family 2; etc...;
- **Subsequence**: "141".

The resulting output is presented in the interface shown in Figure 1.



Subseq: 141

Number of matchings: 18163

Sekvens: 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, 117, 127, 135, **141**, 145, 147, 147, 145, **141**, 135, 127, 117, 105, 91, 78, 66, 55, 45, 36, 28, 21, 15, 10, 6, 3, 1

Positions: 18

Params: n: 3, a/ai: 14, family: 1

Sekvens: 1, 6, 21, 50, 90, 126, **141**, 126, 90, 50, 21, 6, 1

Positions: 7

Params: n: 6, a/ai: 3, family: 1

Figure 1: Web service interface

Note: this screenshot reflects an early prototype, a polished visual design will be developed in subsequent iterations.
The resulting output would then provide all Hilbert vectors belonging to the selected family containing the given subsequence, clearly displaying associated

parameters, thereby facilitating immediate insights and further comparative analysis.
To achieve this goal, the following objectives have been addressed:

Data Modeling: Design a relational schema for storing sequences and associated metadata (algebra parameters);

Interface Development: Implement a web interface for adding, removing, and editing records;

Search Functionality: Provide efficient subsequence search capabilities and parameter-based filtering;

Data Generation: Prepare a suite of Macaulay2 scripts for bulk exporting of Hilbert vectors;

Algorithm Optimization: Verify the correctness of subsequence search and retrieval while tuning the implementation for maximum speed and throughput.

The implementation utilizes the Django framework (Python), SQLite database, and Docker containerization, ensuring rapid query responses and easy deployment of the service.
Subsequent sections of this thesis are organized as follows: Section 2 reviews the mathematical foundations of graded Artinian algebras, surveys existing solutions (OEIS, Macaulay2), and provides a detailed rationale for the chosen technologies (Django, Docker) used in the project; Section 3 describes the methodology and system architecture; Section 4 details the implementation; Section 5 presents the results of performance benchmarking and result of optimisation; and Section 6 concludes the work and outlines directions for future development.

2 Theoretical Background and Selected Tools

2.1 Hilbert vectors and sequences

Consider a polynomial ring $R = \mathbb{C}[x_1, \dots, x_n]$ of homogeneous polynomials. By R_d, we denote the vector space over $\mathbb{C}$ consisting of all homogeneous polynomials of degree d. Its dimension equals the number of monomials of degree d in n variables:

$$\dim(R_d) = \binom{n+d-1}{d}.$$

For example, the dimension of R_d with $n = 3$ and $d = 2$ is:

$$\binom{3+2-1}{2} = \binom{4}{2} = 6,$$

corresponding to monomials $x_1^2, x_1x_2, x_1x_3, x_2^2, x_2x_3, x_3^2$.
An ideal $I = (f_1, \dots, f_r)$ generated by homogeneous polynomials f_i is called Artinian if for some degree d, the dimensions of I_d and R_d coincide, and remain equal for all higher degrees. Formally, an ideal I is Artinian if:

$$\dim I_d = \dim R_d, \quad \dim I_{d+1} = \dim R_{d+1}, \quad \ldots$$

Given such an ideal I, we construct the quotient ring R/I. The Hilbert vector of the quotient ring is defined by the sequence:

$$(\dim R_0 - \dim I_0,\ \dim R_1 - \dim I_1,\ \ldots,\ \dim R_d - \dim I_d),$$

where d is the minimal integer for which equality $\dim I_d = \dim R_d$ occurs.

Example:

For instance, let $n = 3,\ d = 2$, and consider the ideal:

$$f_1 = x_1^2, \quad f_2 = x_2^2, \quad f_3 = x_3^2, \quad I = (f_1, f_2, f_3).$$

In this case, the Hilbert vector of the quotient ring R/I is:

$$(1, 3, 3, 1).$$

Explanation: There is one monomial of degree 0 in R, namely $x_1^0 x_2^0 x_3^0$. There are 3 monomials of degree 1: x_1, x_2, x_3. There are 6 monomials of degree 2:

$$x_1^2,\ x_1x_2,\ x_1x_3,\ x_2^2,\ x_2x_3,\ x_3^2.$$

There are 10 monomials of degree 3, 15 of degree 4, and so on.
Now in the ideal I, there are no monomials of degree 0 or 1, but 3 monomials of degree 2: x_1^2, x_2^2, x_3^2, and all 10 monomials of degree 3 are in I since each will be divisible by one of the generators.
So we compute the Hilbert vector as:

$$(1 - 0,\ 3 - 0,\ 6 - 3,\ 10 - 9) = (1, 3, 3, 1).$$

This sequence describes the dimensions of monomials of degrees $0, 1, 2, 3$ that are not included in the ideal I.
Our database contains such sequences, along with the parameters d and n. The classification into families is as follows:

1. $\mathbb{C}[x_1, \ldots, x_n]/(x_1^d, \ldots, x_n^d)$
2. $\mathbb{C}[x_1, \ldots, x_n]/(x_1^d, \ldots, x_n^d, (x_1 + \cdots + x_n)^d)$
3. $\mathbb{C}[x_1, \ldots, x_n]/(x_1^d, \ldots, x_n^d, (x_1 + \cdots + x_n)^d, \ell^d)$, where ℓ is a generic homogeneous linear polynomial.
4. $\mathbb{C}[x_1, \ldots, x_n]/(x_1^d, \ldots, x_n^d, (x_1 + \cdots + x_n)^{dn+1}, \ell^{dn+2})$, where ℓ is a generic homogeneous linear polynomial.

Due to limited computational resources, the parameter ranges for generating each family had to be restricted, as the generation process otherwise became prohibitively time-consuming. The final parameter sets used are as follows:

- Family 1: $n \in \{2, \ldots, 15\}$, $d \in \{2, \ldots, 15\}$

- Family 2: $n \in \{2, \ldots, 15\}$, each test involves all non-decreasing sequences $(d_1, \ldots, d_n)$ with $d_i \in \{2, \ldots, 7\}$
- Family 3: $n \in \{2, \ldots, 7\}$, $d \in \{2, \ldots, 10\}$; for $n = 7$, the last value of d is 6
- Family 4:
  - $n = 3$: $(d_1, d_2, d_3, d_4, d_5) \in \{1, \ldots, 30\}$
  - $n = 4$: example sequences range from $\{1, 1, 1, 1, 1, 1\}$ to $\{6, 8, 8, 9, 16, 22\}$
  - $n = 5$: $d_i \in \{1, 7\}$
  - $n = 6$: example sequences range from $\{1, \ldots, 1\}$ to $\{5, 5, 7, 8, 8, 8, 8, 8\}$
  - $n = 7$: $d_i \in \{1, 4\}$
  - $n = 8$: $d_i \in \{1, 4\}$

In each case, we fix a value of n, and then iterate over all allowed values of d or over all possible non-decreasing sequences $(d_1, \ldots, d_k)$ of fixed length, where:

- for Family 2, the sequence length is n,
- for Family 4, the sequence length is $n + 2$,

producing a unique (within the specified family) Hilbert vector for each valid parameter combination.

2.2 Macaulay2 and Computational Tools

Macaulay2 is a specialized computer algebra system designed for computations in algebraic geometry and commutative algebra. It provides built-in routines for computing Gröbner bases, Hilbert functions, Betti numbers, and other invariants of graded algebras. Within the context of this project, Macaulay2 serves as a "generator" of integer sequences (Hilbert vectors), which are subsequently stored in our database.
Below is an example of how the sequences for **Family 1** were generated and written to a file. The code for the remaining families can be found on the project's GitHub repository [2].

```
familyEquigeneratedCompleteIntersections = (n,a) -> (
    k = ZZ/5557;
    R = k[x_1..x_n];
    I = ideal apply(gens R, i -> i^a);
    S = R/I;
    for i from 0 to (n*(a-1)) list hilbertFunction(i,S)
);
-- Open file for writnig
outputFile = openOut "output1family.txt";

for n from 2 to 15 do (
    for a from 2 to 15 do (
        seq = familyEquigeneratedCompleteIntersections(n,a);
        -- Writing in format: "n;{a};{sequence}"
        outputFile << n << ";{" << a << "};" << toString(seq) << "\n";
    )
);

-- Closing file
close outputFile
```

Thanks to its powerful built-in mathematical capabilities, Macaulay2 is perfectly suited for generating such sequences efficiently and reliably.

2.3 Other chosen technologies

Within this project, Macaulay2 is used offline to generate integer sequences (Hilbert vectors) which are exported as text files and then imported into our database. The web service that stores, queries and visualizes these sequences is built in **Python**, chosen for its extensive standard library, rich scientific ecosystem, and ease of rapid prototyping. On top of Python, we employ the **Django** web-framework, an open-source, BSD-licensed solution providing a powerful templating engine, a robust object–relational mapper, and a built-in administrative interface that greatly accelerates the implementation of CRUD operations[3]. To ensure that the application runs identically on any developer's machine or production server, we containerize the environment with **Docker**, which is available free for educational and research use [4]. For storage of structured metadata and numerical vectors, we use **SQLite**, prized for its simplicity, file-based architecture and zero configuration setup, which is ideal for the prototyping phase. Finally, the front-end interactivity is handled by **JavaScript** (vanilla, with AJAX), enabling client-side form validation and asynchronous loading of search results without full page reloads, and leveraging its native support in all modern browsers.

2.4 Related Work and OEIS

Although the Online Encyclopedia of Integer Sequences (OEIS) serves as a valuable conceptual prototype, it does not provide the fine-grained structural and parameter metadata required for research on graded Artinian algebras (see Introduction).
Beyond OEIS, several online databases target specialized mathematical objects such as graphs, groups, and manifolds, but none offer a comprehensive repository specifically for integer Hilbert vectors of graded algebras. This web service is therefore designed to fill that gap by combining sequence data with explicit algebraic context and supporting interactive queries by both sequence values and their generating parameters.

3 Methodology, Solution Plan, and System Architecture

The project employs a *design-oriented study* approach with an iterative methodology: at each stage, a working prototype was developed, tested against requirements, and subsequently refined.

3.1 Methodology and Solution Plan

The work is structured into six main stages:

1. **Data Generation**
   - Offline generation of Hilbert vectors for graded Artinian algebras using Macaulay2.
   - Exporting results to plain text format (TXT).
   - Python script handling data ingestion into the database (parsing, validation, storage).
2. **Data Model Design**
   - Domain and data volume analysis.
   - Designing a relational schema (SQLite) accommodating search and filtering requirements.
   - Defining indices and JSON fields for storing vectors.
3. **Search Algorithm Implementation**
   - Initial implementation using greedy subsequence search (complexity $O(N \cdot k)$).
   - Performance profiling and bottleneck identification.
   - Development of an optimized algorithm using preliminary indexing and set intersections (see Section 4).
4. **Integration into Web Service**
   - Wrapping business logic within Django models and views.
   - Setting up REST endpoints and forms via `forms.py`.
   - Implementing AJAX requests for dynamic result updates.
5. **Containerization and Deployment**
   - Building Docker images:
     - Python/Django with dependencies;
     - SQLite (prospectively PostgreSQL);
     - Macaulay2 (if required).
   - Test runs and debugging in a unified environment.
6. **Testing and Validation**

- Integration tests verifying the complete workflow: generation → ingestion → search.
- Measuring response times for varying data volumes, complexity analysis:

$$O\big(k \cdot F + S + P \cdot k\big) \subset O(S + P \cdot k),$$

  where k is the pattern length, F the number of selected "families", S the total size of lists, and P the intersection size of candidates.

3.2 System Architecture

The overall architecture, illustrated in Figure 2, consists of three layers:

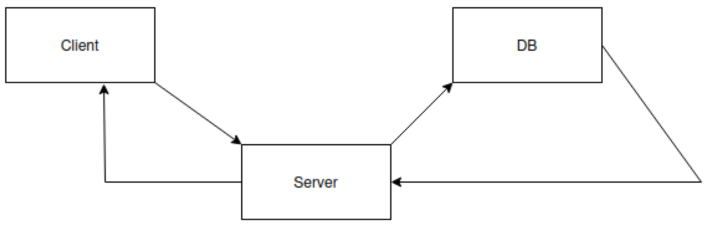


Figure 2: Component Interaction Diagram

Client (Browser + JS/AJAX) Issues search queries and dynamically receives results without full page reloads.

Server (Django, Python) Django organizes the backend into a series of clearly separated modules, and its built-in administrative panel available immediately upon framework installation, which automatically links models, forms, and views into a coherent UI, so we only need to register our models to get a fully functional CRUD interface. The main components are:

- `settings.py`, `apps.py`: General configuration and application initialization.
- `models.py`: Definitions of database schema and business logic.
- `forms.py`: Validation and sanitization of user inputs.
- `urls.py`: Routing of incoming HTTP requests.
- `views.py`: Execution of search algorithms, preparation of template contexts, and construction of JSON responses.
- `admin.py`: Registration of models with Django's admin site, which then provides an out-of-the-box interface for managing all database entities.

DB or Storage (SQLite) Two primary tables:

The `sek_id` field linkage ensures rapid access to each sequence's parameters.

Table 1: Table `Sekvenser_unique`

Field	**Type**	**Not Null**	**Key**
sek_id	INTEGER	+	PK
sekvens	TEXT	+	
family	INTEGER	+	

Table 2: Table `Sekvenser_params`

Field	**Type**	**Not Null**	**Key**
sek_id	INTEGER	+	FK
n_param	INTEGER	+	
a_param	NUMERIC	+	
family	INTEGER	+	

4 Web Service Implementation

4.1 Application Workflow

This section requires detailed attention as it outlines step-by-step all essential aspects of the application's functionality, including descriptions and correctness proofs of key algorithms.
It is convenient to divide this section into three subsections: data preloading, application startup, and retrieval and display of results.

4.1.1 Data Preloading

At application startup, we perform a one-time preprocessing pass in which we build an inverted index that maps each integer value to the list of all `Sequence` objects containing that value. This index is then used to answer subsequence membership queries in constant or logarithmic time, rather than scanning every sequence on each search.

Structure Definition We organize our data using a nested dictionary:

$$\mathsf{sequences\_dict}\colon \{\text{family}\} \longrightarrow \{\mathbb{Z}\} \longrightarrow \text{List}(\mathsf{Sequence}).$$

The top-level keys of `sequences_dict` correspond to family identifiers, and each inner dictionary maps individual integers to the list of sequences containing them, allowing fast access by number and family.
To build this structure, we iterate over each row of the `sequence` table. For each row we construct a `Sequence` instance from the stored string of elements, the family identifier, and the sequence ID. We then extract the set of unique integers occurring in that sequence (via a regular-expression match). If the family is not yet a key in `sequences_dict`, we create an empty inner dictionary. Finally, for each integer in the extracted set, we append the `Sequence` instance to the list held at $\mathsf{sequences\_dict}[\text{family}][\text{integer}]$. This organization allows efficient lookup of all sequences in a given family that contain any specified value(see Figure 3).

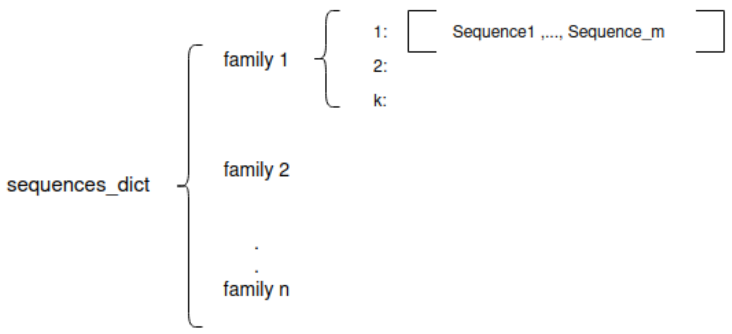


Figure 3: Structure of dictionary

Algorithm 1 Build sequences_dict from database rows

```
1:  $T \leftarrow$ all rows of table sequence
2:  initialize sequences_dict as an empty dictionary
3:  for all row in $T$ do
4:      create seq $\leftarrow$ Sequence( row[0], row[1], row[2] )
5:      $U \leftarrow$ set of all integers extracted from row[0] via regex
6:      if seq.family $\notin$ sequences_dict then
7:          sequences_dict[seq.family] $\leftarrow$ empty dictionary
8:      end if
9:      for all $k$ in $U$ do
10:         if $k \notin$ sequences_dict[seq.family] then
11:             sequences_dict[seq.family][$k$] $\leftarrow$ empty list
12:         end if
13:         append seq to sequences_dict[seq.family][k]
14:     end for
15: end for
```

Initialization Pseudocode

Sequence Class Structure:

- sekvenser (list or array of integers) — the sequence itself;(row[0] above)
- family, sek_id — for database linkage;(row[1],row[2] above)
- pos_dict: dictionary $\{\text{number}\} \rightarrow [\text{positions}]$, mapping each number directly to a list of its occurrence indices;
- get_params() method — retrieves all related parameters from the Sekvenser_params table.

Computational Complexity Estimation: Let N denote the total number of sequences and L_i the length of the i-th sequence. Then:

- Constructing pos_dict for a single sequence takes $O(L_i)$.

- Inserting all occurrences into `sequences_dict` requires a total of

$$\sum_{i=1}^{N} O(L_i) = O\big(\textstyle\sum_i L_i\big).$$

- Searching by a single number within any family executes in $O(1)$ (dictionary access) plus the traversal time for the candidate list.

Overall, preloading takes $O(\sum_i L_i)$ time and memory proportional to the total number of occurrences. Since this phase executes only during application startup, the overhead easily justifies subsequent search operations, performed in $O(1)$ or $O(\log n)$ per indexing step.

Conclusion Initialization ensures a correct and comprehensive index: for any record and number, all relevant `Sequence` objects are correctly positioned in `sequences_dict`. This prepared structure is foundational for describing the primary search algorithm.

4.1.2 Application Startup

After completing the preloading phase, the application enters user request processing mode. The user selects one or more "families" and inputs the desired subsequence $\langle s_1, s_2, \ldots, s_k \rangle$. The function `main_search`, whose pseudocode is given below, is then invoked:

```python
def main_search(request):
    # 1. Load available families for printing
    families = DB.query("SELECT DISTINCT family FROM Sekvenser_unique")
    families.sort()
    if request.method == POST:
        form = SubseqForm(request.POST)
        if form.is_valid():
            # 2. Parse user input
            subseq = parse_int_list(form.cleaned_data["subseq"])
            families_sel = parse_int_list(request.POST.getlist("
                families"))

            # 3. Search matching sequences
            if len(subseq) == 1:
                result_seqs = []
                for family in families_sel:
                    if subseq[0] in sequences_dict[f]:
                        result_seqs.extend(sequences_dict[family][
                            subseq[0]])
            else:
                intersect_set = set()
                first = True
                for target_number in subseq:
                    # Collect all Sequences containing target within
                        selected families
                    current = set()
                    for families in families_sel:
                        if family in sequence_dict and target_number in
                             sequences_dict[family]:
                            current.update(sequences_dict[family][
                                target_number])
                    if first:
                        intersect_set = current
                        first = False
                    else:
                        intersect_set = intersect_set & current
                result_seqs = check_pos(intersect_set, subseq)

            # 4. Generate response

            transfer, params, positions = [], [], []
            for seq in result_seqs:
                transfer.append(seq.sekvens)
                temporary_params = seq.get_params()

                if len(temporary_params) > 1:
                    params.append(temporary_params[0])
                    for p in temporary_params[1:]:
                        transfer.append(seq.sekvens)
                        params.append(p)
                else:
                    params.append(temporary_params[0])

                pos_for_seq = []
                for num in subseq:
                    if num in seq.pos_dict and seq.pos_dict[num]:
                        pos_for_seq.append(seq.pos_dict[num][0])
                    else:
                        pos_for_seq.append("-")
                positions.append(pos_for_seq)

            total = len(transfer)

            # 5. Send response
            if request.accepts_json:
                return JsonResponse({
                  "transfer": transfer,
                  "params": params,
                  "positions": positions,
                  "total": total,
                  "families": families
                })
            else:
                return render(request, "home.html", {
                  "form": form,
                  "transfer": transfer,
                  "params": params,
                  "positions": positions,
                  "total": total,
                  "families": families
                })

    # GET -> empty form
    form = SubseqForm()
    return render(request, "home.html", {"form": form, "families":
        families})
```

Algorithm Description. Once the preloading phase is complete, the application processes each user request by invoking `main_search`. First, the system retrieves the set of all families $\mathcal{F}$ from the database and narrows it to the user-selected subset $\mathcal{F}_{\text{sel}}$. The user's query subsequence $[s_1, \ldots, s_k]$ is parsed into a list of integers. If $k = 1$, the algorithm simply returns the union

$$\bigcup_{f \in \mathcal{F}_{\text{sel}}} \mathsf{sequences\_dict}[f][s_1],$$

which contains exactly those sequences in the chosen families that include s_1. For $k > 1$, for each $i = 1, \ldots, k$ it constructs

$$C_i \;=\; \bigcup_{f \in \mathcal{F}_{\text{sel}}} \mathsf{sequences\_dict}[f][s_i],$$

then computes the intersection $C_1 \cap \cdots \cap C_k$. Finally, the helper check_pos filters this candidate set to retain only those sequences in which the occurrences of $s_1, \ldots, s_k$ appear in strictly right order, as in users input, correctly handling repeated values. The matching sequences are then enriched with parameter and position data before being returned as JSON or rendered in the template.

Proof of Correctness

Denote by

$$A(\mathcal{F}_{\text{sel}}, [s_1, \ldots, s_k])$$

the set returned by the algorithm. We claim

$$A(\mathcal{F}_{\text{sel}}, [s_1, \ldots, s_k]) \;=\; \bigl\{\sigma \mid \sigma \text{ belongs to some } f \in \mathcal{F}_{\text{sel}} \text{ and contains } [s_1, \ldots, s_k]\bigr\}.$$

For $k = 1$, this follows immediately from the preloaded index: each $\mathsf{sequences\_dict}[f][s_1]$ lists exactly those sequences in family f that contain s_1. Assume the claim holds for subsequences of length $k-1$. For length k, intersecting the first i unions yields all sequences containing $s_1, \ldots, s_i$. Intersecting once more with C_k ensures that s_k is also present. The final call to check_pos enforces the correct ordering of occurrences. Hence, by induction, the algorithm returns precisely the set of all sequences in $\mathcal{F}_{\text{sel}}$ that contain the query subsequence.

Complexity Analysis

Let $F = |\mathcal{F}_{\text{sel}}|$, let k be the subsequence length, and define

$$S \;=\; \sum_{i=1}^{k} \Bigl| \bigcup_{f \in \mathcal{F}_{\text{sel}}} \mathsf{sequences\_dict}[f][s_i] \Bigr|, \quad P \;=\; \Bigl| \bigcap_{i=1}^{k} C_i \Bigr|.$$

Building each union C_i requires $O(F + |C_i|)$ time, so assembling all C_i takes $O(kF + S)$. Computing the intersection in a streaming fashion touches each candidate at most once, costing $O(S)$. Finally, check_pos examines each of the P survivors over k positions in $O(P\,k)$. Therefore the total time per request is

$$O\bigl(kF + S + P\,k\bigr) \;\subseteq\; O\bigl(S + P\,k\bigr),$$

which in practice is far smaller than scanning all N sequences. In the best case, when S and P are small, the query is nearly $O(1)$; in the worst case (every sequence matches every value), it degrades to $O(Nk)$. These bounds accurately reflect practical performance using preloaded indices.

4.1.3 Retrieval and Display of Results

After the server completes the search it returns a JSON payload with five top-level fields: `transfer` (sequence strings), `params` (parameter triples), `positions` (first-occurrence indices), `total` (match count), and `families` (the full family list). The client, implemented in vanilla JavaScript with the Fetch API, processes this response in four steps:

1. *Asynchronous submission.* A JavaScript event listener intercepts the form's `submit`, gathers the contents into a `FormData` object, and posts it to the `main_search` endpoint. Suppressing the browser's default action prevents a full-page reload.

2. *Parsing and normalisation.* The returned JSON is converted into a JavaScript object via `response.json()`. The handler then folds the parallel arrays `transfer`, `params`, and `positions` into a single array of result objects for easier downstream processing.

3. *Rendering and interaction.* For each result the script

   (a) highlights every occurrence of the user's query values with a yellow background and red font;

   (b) prints the parameter triple (n, a/a_i, family) with the family number colour-coded;

   (c) lists the first-occurrence indices.

   Above the result list three buttons allow in-browser re-sorting by (i) first occurrence, (ii) family, or (iii) the first entry of a_i. All re-sorting is performed client-side without additional requests.

4. *Error handling and feedback.* Any network or validation error is caught in the promise chain and written to the console; user-visible error messaging can be added via the same handler.

This pipeline yields an entirely fluid search experience: the page never reloads, results can be re-ordered instantly, and the total number of matches is displayed above the list.

4.1.4 Deployment and Portability

All project components run inside a single Docker image that can be saved with `docker save` on one machine and restored with `docker load` on another, guaranteeing a byte-for-byte identical runtime.

Dockerfile. The build recipe (shown below) follows the same structure in every installation:

1. *Base image.* `python:3.11(-slim)` is used.

2. *Environment.* `PYTHONDONTWRITEBYTECODE=1` and `PYTHONUNBUFFERED=1` disable `.pyc` files and stdout buffering.

3. *Dependencies.* `requirements.txt` is copied and installed with `pip`.

4. *Application code.* The entire project tree, including a pre-populated `db.sqlite3`, is copied into `/app`.

5. *Static files.* `python manage.py collectstatic -noinput` gathers CSS/JS once during the build so that runtime containers remain read-only.

6. *Non-root user.* A user `dp` is created and set as the container default, improving security.

7. *Network.* Port 8000 is exposed, and the container starts Django with `python manage.py runserver 0.0.0.0:8000`. (For production the command can be swapped for `gunicorn` without changing the image.)

Advantages. Docker guarantees *portability* (the saved image runs verbatim on Linux, macOS, or Windows), *isolation* (project dependencies never pollute the host), *reproducibility* (the image used in CI/CD is bit-identical to the one deployed in production), and effortless *scalability*: the same image can be orchestrated by Kubernetes or Docker Swarm with zero code changes.

4.2 Web Service Functionality

The application covers the full range of tasks required for working with Hilbert vectors:

- **Interactive subsequence search.** Users can perform instant, asynchronous searches via AJAX; matching integers are highlighted directly inside the sequence text without reloading the page.

- **Flexible filtering and sorting.** Result sets can be restricted by algebraic parameters, such as the family identifier or the length of the parameter list, and re-sorted from client side at any time.

- **Detailed parameter display.** Every sequence is rendered inside an expandable card that reveals all associated rows from the `Sekvenser_params` table.

- **CRUD interface.** A separate maintenance page allows administrators to *add* or *delete* sequences in bulk. New data are uploaded as a plain `.txt` file (one sequence per line) and inserted in a single atomic database transaction. Deletions can be triggered by family, by parameters, or by an sequence itself; each operation is likewise wrapped in a transaction, so it is either fully committed or rolled back guaranteeing that no partial updates are left in the database.

5 Algorithm Optimisation and Memory Usage

Thanks to a one-time preload phase and a hash-based inverted index, subsequence searches execute in near-constant time and, in practice, complete well below one second. Once this speed target was reached, memory footprint became the dominant bottleneck.

Baseline. In the original design every `Sequence` instance stored the full list of parameter rows that describe the algebra, typically several triples $(n,\ a/a_i,\ \text{family})$ per sequence. While this avoided an extra query at display time, it also inflated the resident set: a `tracemalloc` snapshot reported about 2.6 GB for the `sequences_dict` structure alone, leaving little head-room for future datasets.

Refactor. Responsibility was redistributed between the application layer and the database:

- A streamlined `Sequence` object now keeps only three fields: the raw sequence string `sekvens`, its `family` tag, and a primary key `sek_id`.
- All algebraic parameters are fetched *lazily* via a single `SELECT` from the `Sekvenser_params` table when they are first requested subsequent calls are served from a tiny in-object cache.

Results. With parameters no longer duplicated in memory, the dictionary's size dropped from 2.6 GB to 1.7 GB, a reduction of roughly **35**%. The extra latency of the on-demand query is so small that it is imperceptible to users, while the memory savings provide ample head-room for adding new families and sequences.

6 Conclusions and Future Development

6.1 Conclusions

The completed project fully meets the initial objectives:

- Developed and tested a relational storage schema for sequences and metadata;
- Implemented a high-performance subsequence search algorithm, including correctness proof and complexity analysis;
- Created a user-friendly Django-based web interface with comprehensive CRUD functionality and dynamic result display;
- Prepared and documented Macaulay2 scripts for automated Hilbert vector generation;
- Conducted extensive performance and scalability testing, demonstrating stable operation with increasing data volumes;
- Achieved full Docker containerization, ensuring reproducibility and ease of deployment.

The system not only addresses the initial limitations identified with OEIS but also significantly surpasses it in the number of Hilbert vectors stored and their parameter-driven accessibility.

6.2 Future Development

The current prototype provides a robust foundation for ongoing enhancement, reflecting the dynamic and continuously evolving nature of this research project. Further improvements and expansions planned include:

1. **External data integration.** Establish seamless integration with OEIS and other external databases to enrich context, metadata, and available resources.
2. **Migration to PostgreSQL.** Transition to a more robust database system capable of handling parallel queries, advanced indexing, and complex data types to enhance scalability and performance.
3. **Caching and asynchronous task management.** Integrate Redis and Celery to handle heavy computations asynchronously and cache frequently accessed data, thereby improving overall responsiveness and scalability.
4. **Expanded filtering and result presentation.** Introduce advanced filtering options and additional dedicated result pages to improve usability and flexibility, allowing users to perform more precise and customized searches.
5. **User feedback integration.** Conduct extensive user testing to collect direct feedback, identify areas for improvement, and address user-specific needs, enhancing the overall user experience.
6. **Codebase refactoring and technical improvements.** Eliminate temporary or suboptimal solutions in the current implementation to ensure maintainability, reliability, and efficiency in future iterations.
7. **Authentication and role-based access control.** Implement user registration, authentication, and defined user roles to enable collaboration, secure access to specific functionalities, and personalized experiences.

Through these planned enhancements, the platform will progressively mature into a comprehensive and highly interactive research portal for exploring Hilbert vectors associated with graded Artinian algebras.

7 References

References

[1] OEIS available at `https://oeis.org/wiki/Main_Page`.

[2] Github repository availavle at (if some problem with access contact me)
`https://github.com/georgiiKimberg/Bachelor-Thesis-in-Computer-Science`

[3] Django Documentation available at `https://docs.djangoproject.com/en/stable/`.

[4] Docker Documentation available at `https://www.docker.com/`.