LIP-STUDENTS-25-9

Development of an atomic spectra database and accompanying website

Carolina Gomes Freitas[1,a] and João Rodrigo Pinto Jasmins de Freitas[2,b]

[1] *Escola de Engenharia da Universidade do Minho, Braga, Portugal*
[2] *Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal*

Project supervisors: L. Leitão, R. Ferreira da Silva, J. Pires Marques, T. Campante *September 30, 2025*

Abstract. Data on the atomic spectra of heavier ions is not widely available due to several factors. Some of those include a historical lack of necessity as well as a large cost and difficulty to acquire such data experimentally. As such, at the Nuclear Reactions, Instrumentation and Astrophysics (NUC-RIA) research group, part of Laboratory of Instrumentation and Experimental Particle Physics (LIP), there is a large amount of work being done to calculate the levels, transitions and electron-ion collisions data for the ions belonging to the lanthanide group of the periodic table. Currently, most of this data is inaccessible to the wider scientific community despite the publishing of several works, requiring direct contact with the authors of these works. The goal of this work is the creation of a relational database (DB) and accompanying website to facilitate access to this data.

Keywords: Database, Calculated atomic spectra, Lanthanide and actinide ions

1 Introduction

While the question of the origin of all atoms has persisted ever since the known elements were organized into the periodic table, the tools and collective wealth of knowledge required to answer this fundamental question were not always available. Answers to parts of this question have been given by astronomy and astrophysics and their advancements over the years, namely the origin of lighter elements. But the same question about heavier elements, elements heavier than iron, remained unanswered. Many heavier elements were theorized to be present in the same events that produced large amounts of lighter elements, with the understanding that there was a significant mismatch between the expected and observed abundances of these elements. An answer to this question was presented with the detection of the first neutron star merger in 2017 by the LIGO and Virgo Collaborations [1].

Currently, our collective best understanding is that the answer on the origin of most elements lies with the death of stars of varying sizes as well as other cosmic events like merging neutron stars [2], with kilonova being the name given to the electromagnetic transient powered by the radioactive decay of the newly synthesized r-process elements emitted by these mergers. The identification of the elements that are present in these events relies on the observation and registration of the full emission of the electromagnetic spectrum, creating a need for the supporting atomic spectra data on the elements that would possibly be present [3].

This data is gathered by studying the electromagnetic radiation that is emitted and/or absorbed by atoms with the movement of electrons between different orbitals, with each atom and each of its ions having different and unique emission spectra. This uniqueness is the essential characteristic that is used to determine the material compositions of unknown samples and is the foundation of the entire branch of spectroscopy. Since this data is foundational to the work of astronomers that use it to identify the elements present in these cosmic events, as well as other physicists working in other branches of physics, access to reliable data is of utmost importance.

In an attempt to solve this problem of accessibility, there are several different repositories containing data on atomic spectra. These repositories usually contain data obtained experimentally and compiled into a centrally accessible point. The repository that is considered the gold standard is the National Institute of Standards and Technology (NIST) Atomic Spectra Database (ASD) [4]. This is due to several factors such as a wide range of covered ions with very complete data, while being easily accessible and scientifically traceable by linking the various sources it uses and their corresponding published works. Furthermore, the criteria for acceptance and inclusion of data on this DataBase (DB) is very rigorous.

However, despite being considered one of the best collections on this type of data available, it is not complete. Heavier ions, such as those from the lanthanide and actinide groups of the periodic table, are not well represented in this DB. This is visible in figure 1:

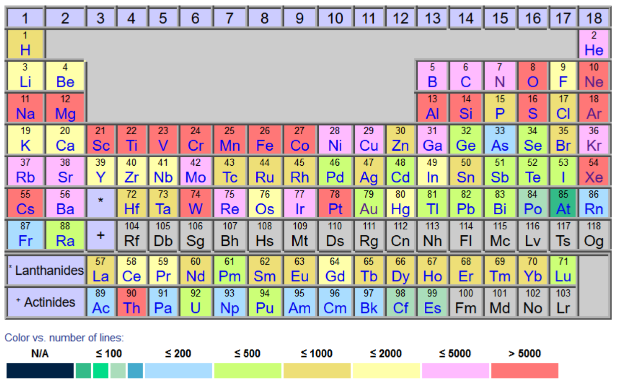


Figure 1. Number of spectral lines registered for each element present in NIST ASD version 5.7.1 (released in October 2019); Image retrieved from [5].

[a]e-mail: carolinagfreitas@sapo.pt
[b]e-mail: joaorpjfreitas@gmail.com

Some of the most relevant reasons for this lack of availability are both historical and economical. Historically, there has not been much interest in gathering this data for heavier elements, in comparison with lighter elements, due to a perceived lack of necessity for the data. Economically, due to the relative rarity of these elements on earth, the acquisition costs for the material will be higher than more common elements. The result is that since 2017 there is a need for this data for further scientific work, while there is very little of it available. The solution to this problem has been to obtain this data theoretically by use of calculation frameworks. This is the work that is being done by several members of NUC-RIA, with some publications having already been made detailing this work. [6–9] Several methods have existed to make these calculations, while the bulk of the data currently stored by NUC-RIA pertains to data generated with Flexible Atomic Code (FAC) [10], with the rest of the data having been generated with other methods for comparison purposes and other ongoing work.

Currently, even though the publications are already part of the academic ecosystem and contributing to the work of other scientists, this data is not easily accessible to the wider scientific community. The goal of this internship is to develop a relational DB to efficiently organize and store this data, and develop a website to facilitate access to the data.

2 Technical Glossary

Due to quite the heavy use of technical programming terminology and details in the following sections of this report that might not be commonly known, this section is dedicated to the identification and clarification of what most of this terminology means. It is recommended to read and familiarize yourself with these terms for proper understanding of the report.

API (Application Programming Interface): A set of HTTP endpoints that external programs can call to request data. In this case, */api/transitions*, */api/collisions*, etc. allow the browser to fetch atomic data without knowing Structured Query Language (SQL).

Asyncio: Python's built-in library for handling thousands of simultaneous tasks without creating separate threads. In the platform, it lets one FastAPI process serve multiple physicists' queries concurrently.

CORS (Cross-Origin Resource Sharing): Browser security rules that will normally block web pages from accessing APIs on different domains. Without CORS setup, the browser would block all the data requests with "blocked by CORS policy" errors.

CPU (Central Processing Unit): The main computational chip in a computer. The platform is CPU-bound when performing a linear regression on 50000 collision points or generating heat-map matrices.

CRUD (Create, Read, Update, Delete): The four basic operations on data. The admin dashboard implements CRUD for tables like users, datasets. Administrators can add new collaborators, edit dataset metadata, etc.

Endpoint: A specific URL path in the API that serves one purpose. For example, */api/transitions* is the endpoint that returns radiative transition data; */auth/token* issues authentication tokens.

JavaScript (JS): Client side programming language, commonly used for web development, improving the UI and interactivity of the websites where it is used.

JSON (JavaScript Object Notation): The text format the API uses to send data. Example: `{"wavelength_nm": 256.15, "TR_rate": 3.2e6}`. Both Python (dict) and JavaScript (object) can easily convert to/from JSON.

FastAPI: The Python web framework used to build the REST API. It automatically generates documentation, validates inputs, and converts Python objects to JSON responses.

Fetch: JavaScript's built-in function for making HTTP requests. For example, *neo.js* uses `fetch('/api/transitions? atomic_number=26')` to request data on iron from the server.

Flask: An alternative Python web framework (FastAPI was chosen instead). Flask is older and simpler but lacks automatic API documentation and built-in async support.

Gunicorn (Green Unicorn): A Python WSGI (Web Server Gateway Interface) server that can run multiple worker processes. It might be used in the future to scale the API horizontally: `gunicorn -w 4 api:app` creates 4 FastAPI workers.

Injection: A programming pattern where functions receive their dependencies as parameters rather than creating them internally. The API routes use `get_db()` injection to receive DB sessions.

JWT (JSON Web Token): A security token that contains encrypted user information. When physicists log in, they receive a JWT that proves their identity for 15 minutes without needing to check the DB on every request.

ORM (Object-Relational Mapping): Software that lets you work with DB tables as if they were Python classes. Instead of writing `SELECT * FROM transitions`, `db.query(Transition)` can be used with SQLAlchemy.

Parsing: Converting text into structured data. The platform parses JSON responses from the API into JavaScript objects, and parses user input like "Fe II" into atomic number 26, charge +1.

Pydantic: A Python library that ensures data has the correct shape and types. The API uses Pydantic models to guarantee that wavelength_nm is always a positive number, preventing unit errors.

Query: DB query: A SQL command like `SELECT wavelength_nm FROM transitions WHERE atomic_number=26`. API query: Parameters in a URL like `?atomic_number=26&wavelength_min_nm=200`.

Routers: FastAPI modules that handle different parts of the API. There are separate routers for levels, transitions, collisions, and admin functions in order to keep the code organized by physics domain.

SQLAlchemy: Python's most popular ORM library. It converts the Python classes (Level, Transition) into SQL table operations and handles connection pooling to MySQL.

TCP (Transmission Control Protocol) socket/packet: The network communication method between the API and the DB. Each DB query travels in TCP packets over a socket connection (usually port 3306 for MySQL).

Token: A piece of data that proves identity. The JWTs are bearer tokens sent by the browser in HTTP headers to prove the user is authenticated without sending passwords repeatedly.

UI/UX (User Interface / User Experience): UI is what physicists see (buttons, tables, plots). UX is how it feels to use (fast loading, intuitive workflows, helpful error messages). The design prioritizes scientific workflow efficiency.

Uvicorn: The Python server that runs the FastAPI application. Command `uvicorn api:app` starts the server on localhost:8000. It handles HTTP requests and converts them to Python function calls.

3 Methodology

Considering all of the requirements of this project, a three-tier approach was settled on. This approach consists of the DB (back-end) itself, the website (front-end) and an API making the connection between both and serving as an interface layer. This approach was chosen over a more integrated website and API due to a multitude of factors, such as:

Scientific Productivity: In the future, the hot-swaping of visual tools without re-deploying the server can be done. With a separate UI it is possible to edit *static/js/neo.js*, refresh the browser, and see the change instantly. If the API and UI were bundled, every tweak would require rebuilding the container or restarting the process, interrupting colleagues' ongoing long-running queries.

Reproducibility and Peer Review: Keeping the API stateless and version-tagged means every published paper can cite the API version for exact query semantics. For example, UI bug-fixing might silently change query behavior and invalidate historic results.

Security Surface Reduction: The DB listens only to the API on a localhost socket and never listens to browsers. The browser receives only pre-sanitised JSON. Even if malicious JS code is injected, it cannot reach raw credentials or the DB port. When the web interface is bundled with the API into a single application, the HTML templating engine and the DB drivers all run in the same process. This makes the system more complex to review and gives attackers more potential entry points, increasing the risk and complicating security audits.

Optimized Resource Usage: Right tool for the right job approach; The back-end is CPU-light, but the server spends more time waiting for responses from the DB (queries, inserts, updates) than performing complex calculations on the CPU; The front-end is CPU heavy. By running in separate processes they can be pinned to different cores, distributing the workload.

Upgrade and update modularity: Since each of the three main components use different tools for different purposes, the rate of releases for updates on each tool is different. The website can have an update cadence that is much more frequent than the API, which in turn is more frequent than the DB. Once the DB is established, the only updates it is likely to receive are related to adding or removing data, with the internal structure remaining unchanged. This means that everything else that is built with this foundation will not require frequent changes. The API however, is reliant on tools and processes that evolve very frequently for security reasons. The website may be updated whenever necessary, as long as the communication pathway to the API, and subsequently the DB, is not changed. This is important in order to maintain the core functionality available, as has been verified by testing.

Scalability Path: Due to the sheer amount of data, when/if users begin batch-fitting 10^7 collision rows, the API can be scaled horizontally while each user still loads the UI locally. In a merged design, horizontal scaling duplicates the heavy UI code, wasting memory and complicating cache invalidation. As previously mentioned, entangling layers would only slow the feedback loop and further burdening the user´s computational resources.

Due to handling large amounts of data for use by the scientific community, some care was also taken to abide by the Findability, Accessibility, Interoperability, and Reuse of digital assets (FAIR) guiding principles for scientific data [11].

For the DB (back-end), the tools used on this project consisted of MySQL 8.0 for the DB Management System (DBMS) due to familiarity with it and because it fit the requirements. Python was used for parsing and reformatting the raw data into the same structure defined in the DB, as well as for the ingestion of new data and querying the DB with the use of the pandas, MySQL-connector-python and SQLAlchemy packages. Some existing python code from members of the group was used as a baseline upon which the rest of the features were built on. For the website (front-end), HTML5, CSS3 and JavaScript ES6+ are the main tools that were used, in order to create a website that was both aesthetically pleasing and functional.

For the API and other tasks, such as hosting the website locally for testing purposes and performing those same tests, the main tools that were used are Uvicorn as an API server, Browser DevTools for debugging and monitoring, Postman, FastAPI and Insomnia for several API features.

4 Full system architecture

To properly explain the full architecture of the prototype and the communication pathways between each of the components, all of the individual pieces themselves and their purposes need to be fully explained first.

4.1 The database

The schema for the DB that was settled on after testing several different designs can be seen in figure 2.

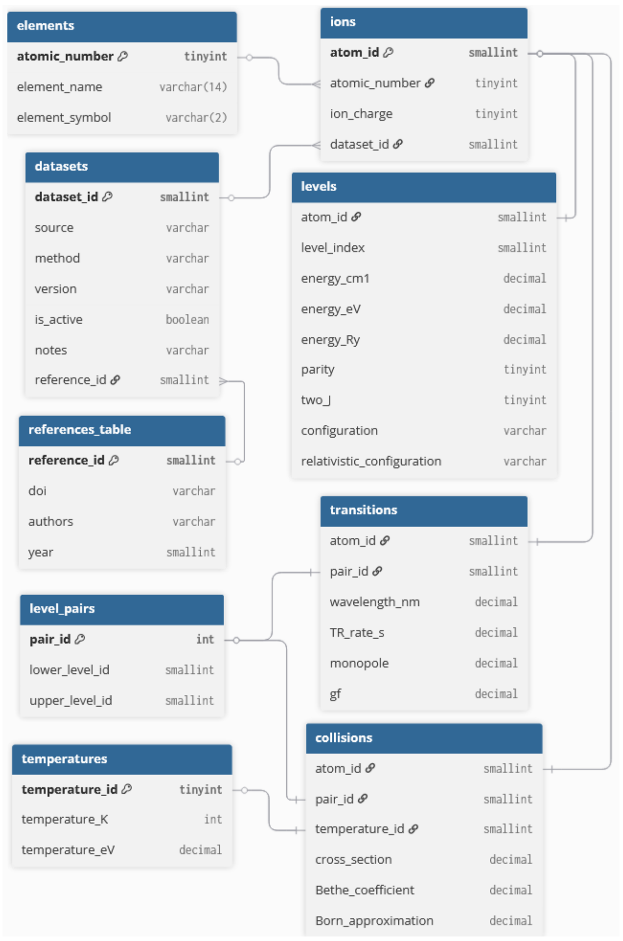


Figure 2. Schema for the DB. Schematic made with dbdiagram.io.

The work on the schema was started by analyzing the raw data that was output by the calculation codes. The output consists of three files per ion; the levels, transitions and collisions files. After further analysis, some patterns were identified with the first of these patterns being that the transitions and collisions are always being identified by two different levels that were registered in the levels file. The second pattern is that the cross sections of each of the electron-ion collisions are averaged over a maxwellian velocity distribution and so, the rates are given for a set of temperatures. Each of these set temperature values is repeated for each collision in the raw data. For example, if there are 10 temperatures and 50000 collisions, each of the 10 temperatures will be repeated 50000 times.

Some of the most important design considerations for the DB were the storage efficiency and the scalability, with several things being done in pursuit of those goals. The two main things being the creation of the temperatures and level_pairs tables, and the optimization of the data types being used for all fields throughout the DB. By normalizing the level pairs, the need for an additional JOIN operation in every query that links transitions or collisions to their specific level energies was introduced. This was tested and it was considered an acceptable performance trade-off for the significant gains in storage efficiency and data integrity.

The final design consists of 9 interlinked tables with several different identifiers (IDs) being attributed to important things within each table and being used in other tables. These tables and their purposes are:

Elements table Stores basic information on each element, linking the element's name with its atomic number and symbol. This is intended to be used on the website for search flexibility. Links to the ions table.

References table Stores the information on the published work associated with the data entries and attributes an ID to it. Uses the name references_table name due to *references* being a reserved keyword for SQL. Links to the datasets tables.

Datasets table Stores the information on the method used for acquisition of data and is different than the references due to the possibility of different methods being used in the same work, attributing an ID for each dataset. There is also an *is_active* boolean flag that allows older datasets to be deprecated in favor of newer versions without deleting the original data, ensuring that past research citing the older data remains reproducible. Links to the references and ions tables.

Ions table Stores the information for the specific ion, attributing an ID to the triplet consisting of the atomic number, the charge and the dataset ID that acquired the data. This way, there can be data on the same ion originating from multiple datasets from different publications, or even from the same publication. Links directly to the datasets, elements, levels, transitions, collisions tables, and indirectly to the level pairs table.

Levels table Stores the levels information, directly parsed from the output files. The pair of atom ID and the level index uniquely identify any level, with the level index being created after sorting the data with increasing energy. The different energy values are conversions from the output file using the Committee on Data of the International Science Council (CODATA) internationally recommended values of the fundamental physics constants from NIST [12] for consistency and accuracy. Links to the ions table, and to both the collisions and transitions tables through the level pairs table.

Transitions table Stores the transitions information, directly parsed from the output files. Every transition is uniquely identified by the combination of atom ID and pair ID. Links to the ions and the level pairs tables.

Collisions table Stores the collisions information, directly parsed and reformatted from the output files. Every collision is uniquely identified by the triplet of atom ID, temperature ID and pair ID. Links to the ions, temperatures and level pairs tables.

Level pairs table Stores the upper and lower levels by use of the level index identifier, from the levels table, by attributing a unique level pair ID. Does not rely on specific atom IDs, effectively serving as an index for the levels table's level index field. This solution was reached to reduce the amount of repeated information (upper and lower levels) in the transitions and collisions tables. Links to the transitions and collisions tables.

Temperatures table Stores the temperatures used to make the calculations on the collisions, assigning a unique ID to each value. This table also stores the temperature converted to Kelvin for easier readability on the website.

The ingestion of the data into the MySQL server was done using several python scripts that parsed the raw data and, after reformatting into the appropriate structure, inserted the data in the DB itself.

4.2 API

The communication flow works as follows: MySQL uses the binary protocol to exchange data between storage and Python, next the interaction between Python and the browser occurs through HTTP/JSON and finally the communication between the business logic and the visualization libraries is handled via in-memory JavaScript objects.

Table 1 summarizes the transport layers that move information through the stack: from the MySQL binary protocol that carries raw query results, through HTTP/JSON for the REST API, to the direct file-system reads that deliver the static HTML and JavaScript to the browser.

Next, in table 2 the main Python entry points can be seen, which are primary modules that the application imports at start-up and through which the rest of the codebase is accessed, explaining how each module contributes.

In table 3 the software bridges are identified. These play a fundamental role in this project due to bearing the responsibility of translating functionality and communication between different tools, languages and protocols.

Finally, a router layer is the part of the FastAPI back end that defines the actual HTTP endpoints, which means the URLs that the browser can call to get data or perform actions. FastAPI organizes endpoints into separate router modules, each one focused on a specific scientific domain or task. A summary of this can be found in table 4.

Table 1. Physical & Logical Transport Layers.

Layer	**Protocol**	**Purpose**
DB ↔ API	TCP socket and MySQL binary (InnoDB engine)	Execute SQL generated by SQLAlchemy; return result-sets or write acknowledgements.
API ↔ Front-End	HTTP 1.1 over localhost (127.0.0.1)	Deliver REST JSON responses; receive filter parameters, auth credentials, CRUD payloads.
Static Assets → Browser	file:// URI scheme (direct disk read)	HTML, CSS, JavaScript, images are loaded straight from the filesystem. No web server required because the content is static.

Table 2. Top-Level Python Entry Points.

File	**Purpose**
api.py	Main FastAPI application: CORS, router registration, custom OpenAPI, DB-session dependency.
auth.py	JWT authentication: password hashing (bcrypt), token creation, first-login password set.
app_database.py	SQLAlchemy ORM models + get_db() generator.
schemas.py	Pydantic response/request models (LevelsResponse, TransitionsResponse, etc.).
api_functional.py	Minimal FastAPI variant used in some test scripts (functional sandbox).

Table 3. Software Bridges & Their Responsibilities.

Bridge	**Technology**	**Encapsulated Data**	**Key Responsibility**
SQLAlchemy ORM	Python package	Python objects ↔ SQL rows	Translates high-level query expressions in routers into parameterised SQL. Ensures type safety, prevents injection.
Pydantic / FastAPI Serialiser	Pydantic BaseModel	ORM models ↔ JSON	Converts Python types (Decimal, datetime) to JSON-safe primitives; validates outgoing schema for every endpoint.
Fetch API (browser)	native JS fetch()	JSON ↔ JS objects	Adds Authorization header, parses JSON, converts to JS arrays/objects for UI rendering.
Plotly / Tabulator	JS libraries	JS objects ↔ SVG/Canvas DOM	Visualise data; reflect user edits back into JSON for PATCH/POST.
SheetJS (xlsx)	JS library	JS objects ↔ binary .xlsx	Client-side Excel export.
JWT Bearer Token	RFC 7519	Base64-encoded JSON	Carries username + role flags; attached to every API call for stateless authentication.

Table 4. Router Layer.

Router	**Key Endpoints**	**Why It Exists**
levels.py	GET /levels	Paginated energy-level groups.
transitions.py	GET /transitions, /expand, helpers	Radiative-transition data delivery.
collisions.py	GET /collisions, /series, /pairs	Electron-collision tables + diagram series.
auxiliary.py	GET /auxiliary/elements, /datasets	Lightweight look-ups for element metadata.
admin.py	CRUD endpoints for users	User & role management by admins.
admin_dashboard.py	Generic CRUD for whitelisted tables	Powers Tabulator dashboard.
__init__.py	Re-exports nothing; marks directory as package.	Empty but required for Python import.

4.3 Website

For the website, several interconnected HTML pages were built using JS modules and CSS styling. Some operational and utility scripts were also necessary to ensure proper functionality of the website and the admin dashboard. These are detailed in the tables below.

Table 5 lists the HTML pages that form the user-facing website, each page tailored to a specific task such as exploring transitions or visualizing collision strengths.

Next, in table 6 a description of the role of the files that handle data fetching, plotting, and role-controlled editing.

With table 7, identification of the CSS files being used can be seen. The usage of a consistent styling across the whole website gives the interface a cohesive appearance and identity. It also ensures the dashboards and plots remain readable. More details about specific files are shown in this table.

Lastly, in table 8, all of the scripts that are currently implemented and that perform specific actions can be seen listed, along with their individual functions.

Table 5. HTML pages currently available on the website.

File	**Role**
index.html	Landing splash; links to Levels / Transitions / Collisions pages.
levels.html	UI for energy-level browsing & Excel export.
transitions.html	UI for transition filters, heat-map, hot-levels logic.
collisions.html	UI for collision tables and Ω(T) diagram.
admin.html	Admin dashboard container (Tabulator grids).
login.html	Simple login form acquiring JWT.

Table 6. JavaScript Modules (static/js/).

File	**Purpose**
neo.js	Unified front-end logic for Levels, Transitions, Collisions pages: fetch helpers, UI wiring, Plotly plots, Excel export.
admin.js	Builds Tabulator grids, handles CRUD and role checks.
login.js	Performs /auth/token, stores JWT in localStorage, redirects on success.
libs/xlsx.full .min.js	SheetJS library (bundled) for client-side .xlsx creation.
libs/tabulat or.min.js	Tabulator grid engine used in admin.js.

Table 7. CSS styles used for UI design.

File	Role
static/css/neo.css	Global styling (buttons, tables, colour maps).
static/css/tabulator.min. css	Table theme required by Tabulator.
static/config.js	Defines window.__API_BASE__ (switch dev vs. prod API URL).

Table 8. Operational & Utility Scripts (ops/ and root).

File	Purpose
ops/bootstrap_admin.py	Create first admin user.
ops/ensure_admin_role. py	Verifies at least one user has is_admin.
ops/test_protected_endp oints .ps1	PowerShell smoke test for 401/403 flow.
ingest/ingest_data.py	Bulk importer for CSV atomic tables.

4.4 How These Pieces Fit Together

When the platform is launched with the command `uvicorn api:app` the server loads the file *api.py* which imports all DB models from *app_database.py* and registers every API router so that each functional area such as levels transitions and collisions is ready to receive requests. A user then opens, for example, *transitions.html* in a web browser and because the interface is fully static the browser reads the JavaScript logic in *static/js/neo.js* and the styling in *static/css/neo.css* directly from the local disk without needing a separate web server or build step. After the page has loaded *neo.js*, it sends HTTP requests to the endpoints defined in the router files. When the responses come back, they are structured as JSON that follows the data structures described in *schemas.py*, which keeps field names and types consistent. For administrative tasks, an administrator opens *admin.html* which loads *admin.js* and these scripts call the endpoints with access controlled by the authentication and role checking logic in *auth.py*. All routes that touch the DB use the same session generator defined in *app_database.py*, which ensures that every query and transaction relies on a single connection pool configuration and behaves consistently throughout the application.

4.5 Typical Request Life-Cycle

As in section 4.4, using the transitions page as an example, the typical process for a processed request is as follows. After authenticating the browser session, the user navigates to the *Transitions* page and selects the desired filters. For example, selecting iron (Fe, atomic number 26), singly-ionized (charge 1), with wavelengths between 200 nm and 400 nm.

When the **Load** button on the page is pressed, the browser executes a JavaScript routine that assembles these criteria into a structured query string, including a request for oscillator-strength values and a limit of 50000 records. The routine issues a single HTTP request to the locally running FastAPI service on port 8000, carrying the user's authorization token in the header. Upon receiving this request, the FastAPI application matches the path */api/transitions* to the function `get_transitions()` in *routers/transitions.py*.

Next, a DB session is obtained through the shared `get_db()` dependency, which either re-uses a pooled MySQL connection or opens a new one as required. SQLAlchemy then constructs an optimized `SELECT` statement that joins the relevant tables, which in this example are the levels, transitions, pairs, ions and datasets tables, while enforcing the wavelength limits and record threshold. The MySQL engine then sends the resulting rows back to the Python layer in an efficient, buffered fashion.

Each row is mapped into strongly typed Pydantic data models such as `TransitionItem` and `LevelBrief`. FastAPI's encoder converts these models into a standard JSON document, translating numerical fields to floating-point values and time stamps to ISO-8601 (YYYY-MM-DD and hh:mm:ss format) strings. The server returns this document with the appropriate `Content-Type: application/json` header and the browser then resolves the response. The client-side JavaScript parses the JSON payload and injects the data into the interactive table and the Plotly visualization components.

If everything works as intended, within a second the user is presented with a fully filterable list of radiative transitions and corresponding spectral plots, all generated from the connected DB. The way some of the most common sources of errors are dealt with, across the different layers of a web application, are described in table 9. Whether through automatic error codes, middle-ware configurations, or library-level optimizations, all of this ensures reliability and a smooth user experience.

Table 9. Error & State Handling Across Layers.

Stage	**Possible Failure**	**Detection & Mitigation**
Validation	Missing query param	Pydantic automatic 422 response; JS displays "invalid input".
Auth	Expired token	python-jose.JWTError → 401 JSON; JS clears LocalStorage, redirects to login.
Network	CORS blocked	FastAPI CORS middleware adds permissive headers; browser accepted.
Client	JS plot overflow	The goal is to keep the graph interactive and fast while plotting up to hundreds of thousands of points in the browser, which could make the page slow or even crash. So if the dataset is larger than 50 000 rows, Plotly internally keeps only a representative subset of the data to display, while the full dataset remains unchanged on your side.

4.6 Additional Communication Details

4.6.1 SQLAlchemy Connection Pool

Pool size defaults to 5; overflow 10; recycle after 1 h.

Each FastAPI request receives its own session via yield from `get\_db()`.

On response completion the session is closed by the `.close()` function, returning the underlying connection to the pool. The benefit of this process is that dozens of concurrent Plotly requests won't spawn uncontrolled MySQL threads.

4.6.2 Token Refresh Logic for Long Sessions

Client polls `/auth/me` every 10 minutes; The server returns "exp": <unix> in claims.

If remaining life is <2 min, browser silently reuses cached username/password combo on the `/auth/token` to refresh the session.

Failure to refresh will give a "Session expired; please log in again" error.

4.7 Practical Engineering Concerns Addressed While Building Code-Base

- **Numeric Precision vs. Front-End Rendering**
  - **Problem:** The stored atomic data can have up to 20 orders of magnitude of precision in storage, so performing consecutive conversion from decimal → float → string could introduce rounding errors or a JS *1e+308* overflow error.
  - **Fix:** Kept decimal format inside API, converted to float only at JSON serialization, then formatted in the browser with a custom `formatNumber()`(scientific notation).
- **CORS under file://**
  - **Problem:** Browsers treat pages loaded via *file://* as having a null origin. By default, this blocks cross-origin requests unless specific headers are set.
  - **Solution:** The CORS middleware is configured with allow_credentials=False (since credentials cannot be sent from a null origin). Now, authentication can be handled via JWT in the HTTP headers rather than not cookies, avoiding the blocked credential issue.
- **MySQL Connection Credentials**
  - **Problem:** Hard-coded fallback connection strings (DSNs) could accidentally expose production passwords if used in the wrong environment.
  - **Solution:** The *.env* file must define SQLALCHEMY_DATABASE_URL, so the fallback string is now clearly marked as "development-only". If the placeholder is detected, Uvicorn aborts the startup, preventing accidental exposure of sensitive credentials.
- **First-Login Password Setting**
  - **Requirement:** New user with NULL hashed_password sets password on first login.
  - **Implementation:** In `authenticate\_user()`, if hashed_password is "None" then hash supplies the password and commits it to the DB in the same request; Prevents race where two logins could hash different passwords.
- **Logging Convention _logger.LogWithContext**
  - Uniform `Class.Method` prefix enforced to help correlate UI requests with API traces. This is used to organize and better track system logs, making debugging and monitoring easier.
  - `Class.Method` prefix also identifies which class and method each log entry comes from, leading to a correlation between a UI action with the corresponding API processing. This method makes it easier to quickly locate where and why an error or behavior occurred.

Next, endpoints are a fundamental part of an API because they define how clients interact with the system. Each endpoint represents a specific resource or action. The main groups of endpoints that were used are specified in table 10.

And finally, due to this DB being accessed through the internet and being available as an online resource, security was also a large concern. The implemented security measures start with binding services to localhost. This ensures that only local processes can reach them, reducing exposure to external threats. Meanwhile, configuring CORS allows to explicitly control which domains can interact with the API. Together, these measures prevent unwanted traffic, limit attack surfaces, and provide a safer environment for client-server communication. More details can be found in table 11.

Input validation and injection defense are critical to protecting applications like this one from malicious data. By enforcing strict rules on incoming requests, SQL injection attacks can be prevented, where attackers attempt to manipulate queries to access or alter DBs. Similarly, validating data against a JSON schema helps detect and block tampering, ensuring that only properly structured and expected input is processed. These safeguards maintain data integrity, protect sensitive information, and are shown in the table 12.

Table 10. Endpoint Groups in the API.

Route Prefix	**Representative Endpoints**	**Primary Functionality**
/auth	• POST /auth/token • GET /auth/me	Issue and validate JSON-Web-Tokens; first-login password initialization.
/levels	• GET /levels	Returns energy-level groups with dataset provenance; supports limit, offset, element and ion filters.
/transitions	• GET /transitions
 • GET /transitions/expand
 • GET /transitions/visible-range
 • GET /transitions/top-by-region	Delivers radiative transitions; fine filters (λ-range, method, g f), grouping by dataset; expansion endpoint provides deep metadata; helper endpoints supply visible-band and "top-5 per region" subsets.
/collisions	• GET /collisions (paginated Ω rows)
 • GET /collisions/series	Supplies electron-collision strengths Ω and temperature series for a chosen level pair; server limits protect memory.
/auxiliary	• GET /auxiliary/elements • GET /auxiliary/datasets	Lightweight look-ups: periodic-table metadata, list of dataset methods/sources.
/admin	• GET/POST/ PATCH/DELETE /admin/users	Strictly user-account management; change roles, disable accounts.

Table 11. Transport & Network Hardening.

Aspect	**Measure**	**Why**
Localhost Binding	uvicorn api:app –host 127.0.0.1 by default.	Prevents remote machines from even opening a TCP connection unless explicitly re-configured.
CORS	allow_origins="*" but allow_credentials= False.	Works with file:// front-end while ensuring browsers still need the bearer token header.

Table 12. Input Validation & Injection Defence.

Attack vector	**Mitigation**
SQL injection	All dynamic queries use SQLAlchemy parameter substitution (`query.filter(User.name == supplied\_name)`). Raw `text()` statements are parameterized via `:name` bindings.
JSON schema tampering	Pydantic models (*schemas.py*) enforce type (e.g. `wavelength\_nm: float > 0`); Invalid payloads trigger automatic error 422 responses.

4.8 User-Centered Front-End Design Choices

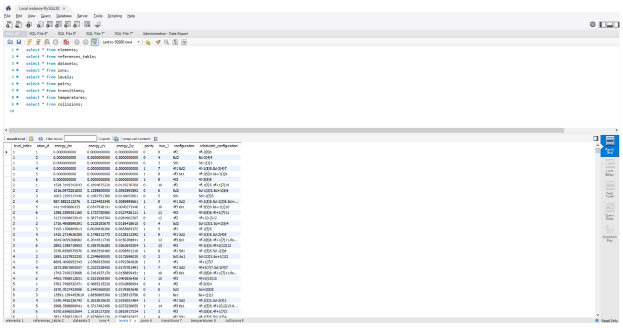
Lastly, some detail on the design choices that were chosen is essential. Beyond technical functionality, a primary goal was to create a tool that is intuitive and efficient for its target audience of research physicists, to guarantee that this is a tool that people want to use, rather than a tool they reluctantly need to used due to specific data not being available anywhere else, as is the current status quo of data and DBs in the field. To achieve this, a user-centric design approach was adopted, by identifying several key work habits and potential pain points for scientists interacting with atomic data. Then specific UI/UX solutions were engineered to address them. The process is summarized in table 13, linking what was observed to the resulting design choices in the application.

Table 13. Website design choices and reasoning.

Concern	**Observation of Physicists' Work Habits**	**Design Response in the UI**
Unit awareness	Physicists switch between nm, Å, eV; unit mistakes are costly.	• Column headers include units ("Wavelength [nm]", "Energy [cm^{-1}]"). • Tooltip on wavelength field reminds "input values in nm".
Visual pattern recognition	Scientists spot anomalies faster in color plots than in tables.	• Visible-band rows are color-coded against the rainbow (UV→IR). • Heat-map for transition connectivity uses perceptually uniform "Viridis" palette.
Provenance & Metadata	Papers require citing data source, method, DOI.	• Dataset badge appears above every table, for example: *Method: R-Matrix · Source: Smith 2025*. Hovering over it shows full reference information.
Minimal clicks to first plot	Demonstrations should produce a graph in <15 s.	• "Show Diagram" button appears immediately after data load; default sample size and linear fit are pre-selected.
Edge-case safeguards	Blank tables are discouraging.	• If query returns zero rows, the page shows a muted "No data for current filters" instead of an empty grid.

5 Working system showcase

This section showcases live results from a locally hosted website server and MySQL server populated with real data. In figure 3 the populated back-end can be seen being accessed by MySQL workbench. This is not visible to the website user.



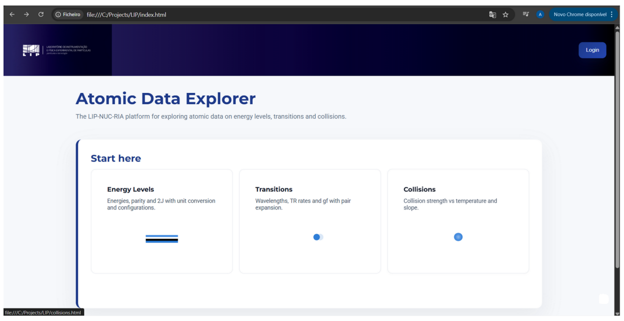
Local instance MySQL80
File Edit View Query Database Server Tools Scripting Help
fetch-all SQL File 5* SQL File 6* SQL File 7* SQL File 7* Administration - Data Export
Limit to 50000 rows
Result Grid Filter Rows: Export: Wrap Cell Content:

```
select * from elements;
select * from references_table;
select * from datasets;
select * from ions;
select * from levels;
select * from pairs;
select * from transitions;
select * from temperatures;
select * from collisions;
```

level_index	atom_id	energy_cm	energy_eV	energy_Ry	parity	two_J	configuration	relativistic_configuration
1	1	0.0000000000	0.0000000000	0.0000000000	0	8	4f2	4f-2(8)8
1	2	0.0000000000	0.0000000000	0.0000000000	0	4	5d2	5d-2(4)4
1	3	0.0000000000	0.0000000000	0.0000000000	0	3	5d1	5d-1(3)3
1	4	0.0000000000	0.0000000000	0.0000000000	1	7	4f1 5d2	4f-1(5)5.5d-2(4)7
1	5	0.0000000000	0.0000000000	0.0000000000	1	8	4f3 6s1	4f-3(9)9.6s+1(1)8
1	6	0.0000000000	0.0000000000	0.0000000000	1	9	4f3	4f-3(9)9
2	1	1528.3199342043	0.1894875220	0.0139270760	0	10	4f2	4f-1(5)5.4f+1(7)10
2	2	1016.0972251833	0.1259800000	0.0092593592	0	6	5d2	5d-1(3)3.5d+1(5)6
2	3	1603.2299317440	0.1987751780	0.0146097061	0	5	5d1	5d+1(5)5
2	4	987.5882212576	0.1224453340	0.0089995661	1	9	4f1 5d2	4f-1(5)5.5d-1(3)8.5d+...
2	5	441.9499806433	0.0547948141	0.0040273446	1	10	4f3 6s1	4f-3(9)9.6s+1(1)10
2	6	1398.3399351160	0.1733720560	0.0127426111	1	11	4f3	4f-2(8)8.4f+1(7)11
3	1	3127.0998633919	0.3877109700	0.0284962307	0	12	4f2	4f+2(12)12
3	2	1716.4958896391	0.2128183670	0.0156418615	0	4	5d2	5d-1(3)3.5d+1(5)4
3	3	7195.1396809813	0.8920836260	0.0655669372	1	5	4f1	4f-1(5)5
3	4	1410.2714636385	0.1748513770	0.0128513392	1	9	4f1 5d2	4f-1(5)5.5d-2(4)9
3	5	1649.0099268682	0.2044511740	0.0150268841	1	12	4f3 6s1	4f-2(8)8.4f+1(7)11.6s...
3	6	2893.1398719953	0.3587036280	0.0263642304	1	13	4f3	4f-1(5)5.4f+2(12)13
4	1	3276.6598575970	0.4062540460	0.0298591216	1	8	4f1 5d1	4f-1(5)5.5d-1(3)8
4	2	1895.1527933330	0.2349690000	0.0172699030	0	2	5d1 6s1	5d-1(3)3.6s+1(1)2
4	3	8695.4096052343	1.0780933900	0.0792384026	1	7	4f1	4f+1(7)7
4	4	1873.8907693557	0.2323328450	0.0170761491	1	7	4f1 5d2	4f+1(7)7.5d-2(4)7
4	5	1743.7199233668	0.2161937170	0.0158899451	1	10	4f3 6s1	4f-2(8)8.4f+1(7)11.6s...
4	6	4453.7598012831	0.5521958390	0.0405856456	1	15	4f3	4f+3(15)15
5	1	3762.7498332471	0.4665215220	0.0342886994	0	4	4f2	4f-2(4)4
5	2	1970.7027433908	0.2443360000	0.0179583648	0	8	5d2	5d+2(8)8
5	3	13591.1394453619	1.6850865300	0.1238515756	0	1	6s1	6s+1(1)1
5	4	2140.4426236743	0.2653810630	0.0195051484	1	1	4f1 5d2	4f-1(5)5.5d-2(4)1
5	5	2998.3598690941	0.3717492450	0.0273230655	1	14	4f3 6s1	4f-1(5)5.4f+2(12)13.6...
5	6	9370.6596052694	1.1618137200	0.0853917334	1	3	4f3	4f-2(8)8.4f+1(7)3
6	1	3821.5298317613	0.4738093130	0.0348243422	1	4	4f1 5d1	4f-1(5)5.5d-1(3)4

Figure 3. Populated test DB in MySQL workbench.

In figure 4, the landing page for the website can be seen. There is a login button at the top, intended for administrative purposes, as well as large boxes for each of the three main data types available on the website. The first one is for the levels data, the second one for the transitions, and the third one for the collisions.



file:///C:/Projects/LIP/index.html
Login
Atomic Data Explorer
The LIP-NUC-RIA platform for exploring atomic data on energy levels, transitions and collisions.
Start here
Energy Levels
Energies, parity and 2J with unit conversion and configurations.
Transitions
Wavelengths, TR rates and gf with pair expansion.
Collisions
Collision strength vs temperature and slope.

Figure 4. Landing page for the website.

For administrators, valid credentials are necessary. This is done through logging in by pressing the login button, upon which a small window will pop up, visible in figure 5. After the login, the administrator dashboard is available, seen in figure 6.

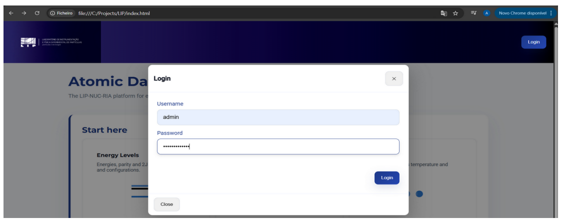
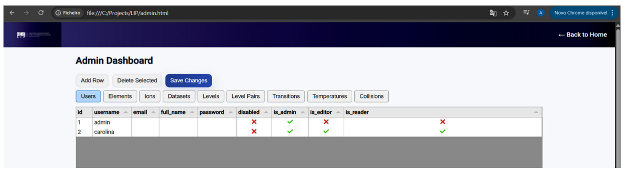


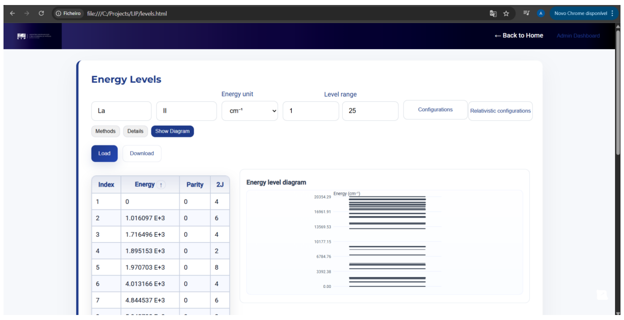
Figure 5. Administrator login popup.



id	username	email	full_name	password	disabled	is_admin	is_editor	is_reader
1	admin				✗	✓	✗	✗
2	carolina				✗	✓	✓	✓

Figure 6. Administrator management dashboard and options.

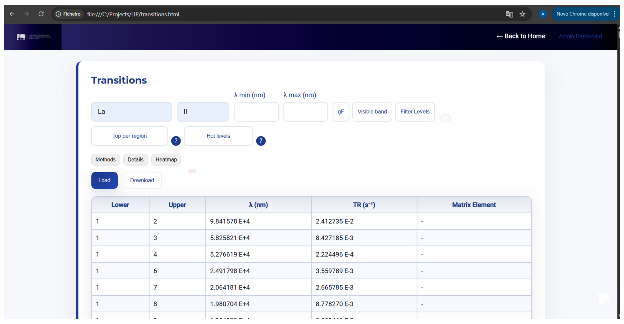
For every other user of the DB, no login is required. Upon clicking each of the boxes, for the levels, transitions and collisions, the user will be taken to the correct page, as seen in figures 7, 8 and 9 respectively. Upon reaching the intended page, the required user inputs are the element they want to check, as well as the ion charge, with these inputs being the same for the three pages. The other features in common between all three pages are the download button, the identification of a level (levels) or level pair (transitions and collisions), the option to click on the headers of any column to sort the shown data, the details button for information on the reference associated with the data and the methods button, to allow the user to choose what data they want to see based on the method that was used.



Index	Energy	Parity	2J
1	0	0	4
2	1.016097 E+3	0	6
3	1.716496 E+3	0	4
4	1.895153 E+3	0	2
5	1.970703 E+3	0	8
6	4.013166 E+3	0	4
7	4.844537 E+3	0	6

Figure 7. Example of levels page.

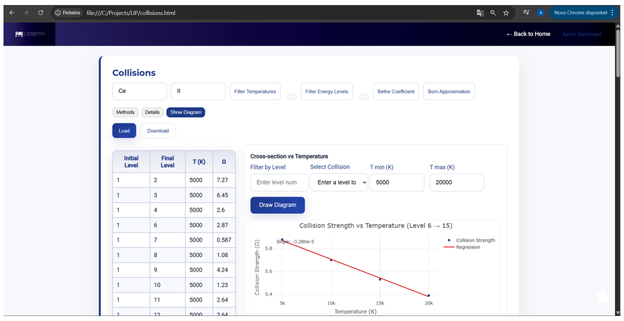
On the levels page, seen in figure 7, the user can also specify a level range, the energy unit (options are cm^{-1}, eV and Ry), a toggle button for whether the columns with the configuration and/or relativistic configuration for each level is shown, and a toggle button for the diagram showing the levels.



Lower	Upper	λ (nm)	TR (s⁻¹)	Matrix Element
1	2	9.841578 E+4	2.412735 E-2	-
1	3	5.825821 E+4	8.427185 E-3	-
1	4	5.276619 E+4	2.224496 E-4	-
1	6	2.491798 E+4	3.559789 E-3	-
1	7	2.064181 E+4	2.665785 E-3	-
1	8	1.980704 E+4	8.778270 E-3	-

Figure 8. Example of transitions page.

On the transitions page, seen in figure 8, it is possible to narrow the search to specific wavelengths and/or specific levels, as well as toggling the column for **gF**. The transfer rate is also available in the column **TR (s^{-1})**.



Initial Level	Final Level	T (K)	Ω
1	2	5000	7.27
1	3	5000	6.45
1	4	5000	2.6
1	6	5000	2.87
1	7	5000	0.587
1	8	5000	1.08
1	9	5000	4.24
1	10	5000	1.23
1	11	5000	2.64
1	12	5000	2.64

Figure 9. Example of collisions page.

On the collisions page, seen in figure 9, the Bethe coefficient and Born approximation columns are toggleable by button as is the diagram of the collision strength vs temperature with filtering. It is also possible to filter by temperatures and energy/level ranges.

6 Conclusions and future work

At this time, all of this work has culminated in a working prototype that has yet to be deployed. In the hopefully near future, the intention is to see this work to be made available and for it to become a tool for the usage of the rest of scientific community. Some things may require modifications depending on the tools used by the future host for this DB and website, as well as continual maintenance work due to the dependencies used being ever evolving with continued updates.

For the initial deployment, the data from NUC-RIA on a range of ions is a good starting point due to being mostly done with the same method (FAC [10]). Since this data on lanthanides is currently mainly used in astrophysics, the most useful application of this initial state of this DB would be to astrophysicists that need it. But the eventual expansion of this DB to accept data from different sources is a loose long term goal. A possible differentiating factor of this DB compared to others could be the acceptance of data from theoretical computer calculations, exclusively, or from both experimental datasets as well as theoretical. There are other groups doing this same type of work with various calculation tools and frameworks, as seen on [13]. As such, it would be feasible to receive data from other sources since the data exists.

An important piece of feedback that was received when starting this project on why there are so many different DBs, each with very sparse data available on it, is that there is a significant hassle involved to do the parsing and conversion of data from different sources into one singular place and structure. From what was also mentioned in this information, this happens due to those same DBs expecting the submitters to write the code to make the data compatible with the DB's structure. This may present a significant amount of work to scientists that already have busy schedules. Which effectively leads to the stranding of datasets to the most convenient repositories, or in a worst case scenario, this data never being made available to those that may need it. Feedback like this was received quite early in this work, so an easy structure to implement for the schema was paramount.

It may be necessary for the future maintainers of this resource to have the ability to write these scripts for each reliable and proven method in order to diminish the workload on the side of the data submitters, requiring them only to submit the raw data along with the reference the data belongs to and some proof of identity. This may be what is required to reduce the inertial barrier for the submission of new data in order to have a reliable and wide reaching scientific resource.

Acknowledgements

We would like to thank our advisors for all the help they provided during this internship, from the answers to our relentless barrage of questions about this and that, the lots of helpful tips they provided us whenever necessary, to all they collectively, and individually, taught us about atomic physics. Further, we would also like to thank the rest of the NUC-RIA team for the companionship during this internship, as well as professor Jorge Sampaio and professor Daniel Galaviz for helping us in searching for a possible future host for this DB and website.

References

[1] LIGO Scientific Collaboration and Virgo Collaboration. GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral. *Physical Review Letters*, 119(16), 2017.

[2] Jennifer A Johnson, Gail Zasowski, David Weinberg, Yuan-Sen Ting, Jennifer Sobeck, Verne Smith, Victor Silva Aguirre, David Nataf, Sara Lucatello, Juna Kollmeier, et al. The Origin of Elements Across Cosmic Time: Astro2020 Science White Paper. *arXiv preprint arXiv:1907.04388*, 2019.

[3] Albert Sneppen, Darach Watson, Rasmus Damgaard, Kasper E Heintz, Nicholas Vieira, Petri Väisänen, and Antoine Mahoro. Emergence hour-by-hour of r-process features in the kilonova AT2017gfo. *Astronomy & Astrophysics*, 690:A398, 2024.

[4] A. Kramida, Yu. Ralchenko, J. Reader, and NIST ASD Team. NIST Atomic Spectra Database (version 5.12). Online, https://physics.nist.gov/asd, 2024. Accessed 2025-09-22.

[5] Yuri Ralchenko and Alexander Kramida. Development of NIST Atomic Databases and Online Tools. *Atoms*, 8(3), 2020.

[6] A Flörs, R F Silva, J Deprince, H Carvajal Gallego, G Leck, L J Shingles, G Martínez-Pinedo, J M Sampaio, P Amaro, J P Marques, S Goriely, P Quinet, P Palmeri, and M Godefroid. Opacities of singly and doubly ionized neodymium and uranium for kilonova emission modeling. *Monthly Notices of the Royal Astronomical Society*, 524(2):3083–3101, 07 2023.

[7] Andreas Flörs, Ricardo Ferreira da Silva, José P Marques, Jorge M Sampaio, and Gabriel Martínez-Pinedo. Calibrated Lanthanide Atomic Data for Kilonova Radiative Transfer. I. Atomic Structure and Opacities. 2025.

[8] Ricardo Ferreira da Silva, Andreas Flörs, Luís Leitão, José P. Marques, Gabriel Martínez-Pinedo, and Jorge M. Sampaio. Systematic Bayesian optimization for atomic structure calculations of heavy elements. *Phys. Rev. A*, 112:012802, Jul 2025.

[9] Ricardo F. Silva, Jorge M. Sampaio, Pedro Amaro, Andreas Flörs, Gabriel Martínez-Pinedo, and José P. Marques. Structure Calculations in Nd III and U III Relevant for Kilonovae Modelling. *Atoms*, 10(1), 2022.

[10] Ming Feng Gu. Flexible Atomic Code. Online; Open source code available on Github: https://github.com/flexible-atomic-code/fac.

[11] M. Wilkinson, M. Dumontier, and I. et al. Aalbersberg. The FAIR Guiding Principles for scientific data management and stewardship. *Scientific Data*, 3, 2016.

[12] NIST. CODATA Internationally recommended values of the fundamental physics constants. Online, https://physics.nist.gov/cuu/Constants/energy.html. Accessed 2025-07-28.

[13] Sultana N. Nahar. Theoretical Spectra of Lanthanides for Kilonovae Events: Ho I-III, Er I-IV, Tm I-V, Yb I-VI, Lu I-VII. *Atoms*, 12(4), 2024.