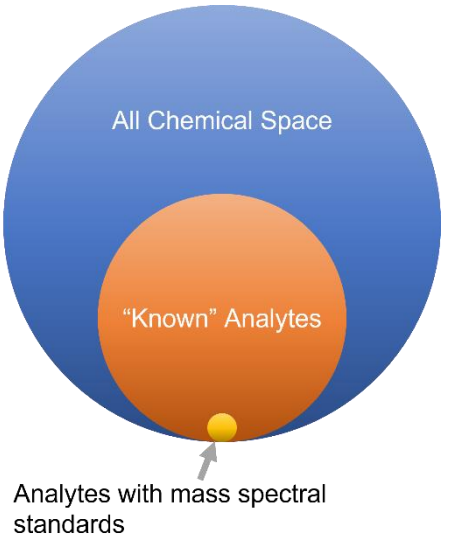
A User Guide for the NIST Database Infrastructure for Mass Spectrometry (DIMSpec) Tool Set

Jared M. Ragland and Benjamin J. Place

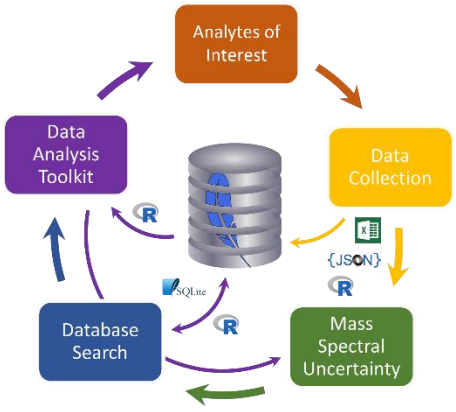
2022-11-02

Preface

The use of mass spectral libraries is essential for the confident identification and reporting of analytical chemistry measureands, whether they be environmental contaminants or novel compounds. Toward that end the US National Institute of Standards and Technology provides a series of mass spectral libraries in use at analytical research, development, and contracting laboratories throughout the US. Contrasting with targeted methods, where analyte identities and their mass spectral properties are known *a priori* are non-targeted analysis (NTA) methods, where compound identity is unknown. Even when analytes are "known" (i.e. their mass spectral properties have been measured and reported in the literature) different extraction and measurement methods may show slight differences in mass spectral properties. The gold standard for matching mass spectra is use of a library of mass spectral standards, yet such standards are available only for a small fraction of "known" analytes.



For analyte classes with decades of analytical study (e.g. polychlorinated biphenyls, vitamins, etc.), such "known" analytes are generally covered well by analytical standards. This is not the case for analyte classes of more recent interest (e.g. per- and polyfluorinated alkyl substances). Research needs generally run ahead of standards availability; new paradigms such as NTA are therefore necessary to assess analyte identity and properties by identifying mass spectral patterns of both the analyte itself and its fragmentation patterns. Analytes characterized in this fashion can then be published for identification in other methods and laboratories. As of 2022, the process of distributing such data still lags behind many research needs.



This book describes a tool set produced by the NIST Chemical Sciences Division to capture data from high resolution accurate mass spectrometric experiments in a formal manner. The Database Infrastructure for Mass

Spectrometry (DIMSpec) allows for the creation of portable databases that tie such data with sample and methodological metadata. DIMSpec uses SQLite, a common portable database engine, for data storage and a collected set of data management and NTA tools written in the R language. This includes the ability to rapidly iterate and launch new databases to hold data for a particular project, analyte class of interest, or research program, and results in a single database file that may be shared widely without restriction on resulting use. When used with the associated tools, researchers can leverage NTA tools in active use at NIST for quality assurance, identification of unknown analytes using current state-of-the-science techniques, and to record their data and contribute back to the research community. Only open access frameworks were used in the development of DIMSpec.

We believe strongly in the public availability of and open access to research data and hope that the tool set described here can be of use in moving the NTA research community toward a data structure amenable to sharing and reuse and move analytical chemistry data for NTA toward the FAIR principles.

Sincerely,

Jared [1] and Ben [2]

National Institute of Standards and Technology Material Measurement Laboratory
Chemical Sciences Division
[1] Chemical Informatics Group, Research Biologist
[2] Organic Chemical Metrology Group, Research Chemist

Introduction

In analytical chemistry, the objective of **non-targeted analysis (NTA)** is to detect and identify unknown (generally organic) compounds using a combination of advanced analytical instrumentation (e.g. high-resolution mass spectrometry) and computational tools. For NTA using mass spectrometry, the use of reference libraries containing fragmentation mass spectra of known compounds is essential to successfully identifying unknown compounds in complex mixtures. However, due to the diversity of vendors of mass spectrometers and mass spectrometry software, it is difficult to easily share mass spectral data sets between laboratories using different instrument vendor software packages while maintaining the quality and detail of complex data and metadata that makes the mass spectra commutable and useful. Additionally, this diversity can also alter fragmentation patterns as instrument engineering and method settings can differ between analyses.

This report describes a set of tools developed in the NIST Chemical Sciences Division to provide a database infrastructure for the management and use of NTA data and associated metadata. In addition, as part of a NIST-wide effort to make data more Findable, Accessible,

Interoperable, and Reusable (FAIR), the database and affiliated tools were designed using only open-source resources that can be easily shared and reused by researchers within and outside of NIST. The information provided in this report includes guidance for the setup, population, and use of the database and its affiliated analysis tools. This effort has been primarily supported by the Department of Defense Strategic Environmental Research and Development Program (DOD-SERDP), project number ER20-1056. As that project focuses on per- and polyfluoroalkyl substances (PFAS), DIMSpec is distributed with mass spectra including compounds on the NIST Suspect List of Possible PFAS (Place, Benjamin J. 2021b) as collected using the Non-Targeted Analysis Method Reporting Tool.

This tool set[i] was developed as part of the NIST PFAS program in the Material Measurement Laboratory's Chemical Sciences Division. It is primarily developed in R and SQLite. The remainder of this book describes the tool set, its technical details, and how to use it.

Contributors

The main two contributors to code and data in this project are

- Benjamin J. Place (PI, Organic Chemical Metrology Group)
- Jared M. Ragland (co-PI, Chemical Informatics Group)

Additional members of NIST PFAS project team providing guidance, input, and testing for this project include

- Jessica L. Reiner (co-PI, Biochemical and Exposure Science Group)
- Alix Rodawa (Biochemical and Exposure Science Group)
- Katherine Peter (Univ. of Washington)
- John Kucklick (Group Leader, Biochemical and Exposure Science Group)
- Catherine A. Rimmer (Group Leader, Organic Chemical Metrology Group)
- Vincent K. Shen (Group Leader, Chemical Informatics Group)

Contributing

Contributions to this project are encouraged! NIST engaged in this project to meet the goals of a single research project. We have made this project available in the name of the public interest and with hopes it will prove useful outside the immediate context. Issues with the code are most effectively reported through GitHub. Pull requests are also encouraged.

About this Book

Chapters and sections in this book were originally drafted and approved as NIST Reports of Analysis within the Chemical Sciences Division. This book serves as the official User Guide for the DIMSpec project and will be expanded and updated appropriately on the project website. Internal NIST Reports of Analysis describing each aspect of the project were converted to R Markdown documents and stitched together using the bookdown package (v0.29). The webbook version will be maintained as a living document on GitHub, as long as the underlying project is active and as obligations allow. The version you are reading now is a static one created from that webbook on [Publish Date]. Function references are included in the online version but removed here for simplicity. If a section is insufficiently documented, let us know.

Instructions

Installation

At the moment, this tool set is only available outside of NIST through GitHub (the preference, either by fork, clone, or download) or directly from one of this book's authors. For now, this tool set includes the NIST PFAS Spectral Library. It is best used as an R project which can be opened directly in the RStudio Integrated Development Environment (IDE)[ii] which may be downloaded and installed free of charge if not already installed on a target system. Initial set up does require an internet connection to download software installers and dependencies; on a system which does not contain any software components this can take a considerable amount of time.

System Requirements

DIMSpec has been tested on both Windows 10 and Ubuntu 20.0.4.3 LTS 64-bit[iii] platforms and should run on any system able to install R, Python, SQLite3, and a web browser, though installation details may vary for other operating systems. Follow the instructions for each requirement on the target operating system.

[REQUIRED] R 4.1+ (download) and many packages are required (R Core Team (2021); various); necessary packages will be installed when the compliance file is sourced, which may take some time when the project is first installed. The RStudio IDE (download; RStudio Team (2020)) is highly recommended for ease of use as this project is distributed as an R project.

[STRONGLY RECOMMENDED] SQLite3 (download) and its command line interface (CLI; download) provide the database engine in structured query language (SQL) and are not technically required as the build can be accomplished purely through R, but are highly recommended to streamline the process and manipulate the database. A lightweight database interface such as DBeaver Lite is also suggested for interacting with the database in a classical sense. **Git** (install instructions) is a repository manager which will make it much easier to install and update the project. The sqlite3 CLI and git executables must be available via PATH.

[RECOMMENDED] For chemical informatics support, both **Python 3.9+** and the **rdkit** (RDKit: Open-Soure Cheminformatics (version 2021.09.4)) library are required for certain operations supporting display and calculations, primarily generation of machine-readable identifiers (e.g. InChI, InChIKey, SMILES, etc) but the full capabilities of rdkit are available (see the RDKit documentation for details); these are turned on by default but are completely optional. An **anaconda** or **miniconda** installation is required. Python integration is not required for spinning up the basic database infrastructure. Users may need to add the conda executable to their PATH and, if conda is already installed, should pay close attention to the Python section of Technical Details. If these are not available, R

will install miniconda (this requires user confirmation at the console) and create the necessary environment as part of automated setup during the compliance script. (Another option for chemical informatics is to use the Java-based R package rcdk instead; users will need to install the Java framework prior to installing rcdk (see Windows; Ubuntu). This package is not well supported in this project and rdkit is preferred.)

[OPTIONAL] It is helpful to have some data on hand to populate and evaluate the database. Every effort has been made to simplify the process of building databases using this tool, and data can be populated from CSV files of a defined structure; examples are provided but the process of generating them can be somewhat onerous as key relationships must be defined to automatically populate in this manner. Future work may be able to simplify this process further, if necessary, but for now, interested researchers are encouraged to contact the authors for guidance on how to transform data to fit this schema.

The following sections provide more detailed information on how to use the tools provided to interact with the database and customize it for other uses.

Quick Start Guide

This section provides instructions in a "quick start" format. While every effort was made to make this as painless as possible, success may vary from system to system. This assumes that R v4.1 or later is installed.

- *If using RStudio:*
    1. Open the project in RStudio.
    2. Open the file at "R/compliance.R" in the editor.
    3. Run the compliance script by clicking the "Source" button at the top right of the editor pane or typing `source("R/compliance.R")` in the console pane.
- *If not using RStudio:*
    1. Open an R session in the project directory or launch R and set your working directory to that of the project (e.g. `setwd(file.path("path", "to", "dimspec_dir"))`).
    2. Execute the command `source("R/compliance.R")`.

Using either method should in most cases establish the compute environment, including logging and argument validation, binding to a python environment providing rdkit support, launching an API server, and listing out the shiny apps available. The project is distributed with a database populated with high resolution mass spectrometry data for per- and polyfluoroalkyl substances (PFAS) for evaluation purposes both to distribute this data set and to evaluate capabilities for reuse in other projects.

Project Directory

The project directory contains the following directories of interest:

- `/config` Files pertaining to the build, description, and population of the underlying database, as well as certain compute environment settings and import settings. This serves for rapid rebuilding and reuse of the underlying database structure. Also included are environment establishment files for the project ("env_glob.txt"), the R session ("env_R.R") and the optional logging ("env_logger.R") functionality.
  - `/sql_nodes` Files containing the sql scripts defining the database schema, as run by the "build.sql" script. Files are separated into database "nodes" with the hope that many can be repurposed or used a la carte in future projects. A graphical representation of the database schema, the entity-relationship diagram (ERD) is also available.
  - `/data` Comma-separated-value (CSV) files which can be used to populate tables defined by their SQL nodes and which will be populated according to the chosen population script. This directory contains common data which should be applicable to all database produced by this tool (i.e. normalization tables, elements and isotopes, etc.) and subdirectories containing project-specific CSV files.
- `/example` Files providing examples of (mainly) import files in JavaScript Object Notation (JSON) format. These are the files used to populate empirical data and were produced by the NIST Non-Targeted Analysis Method Reporting Tool.
- `/images` If images of molecular models are produced using rdkit through this toolset, they will be housed here, named by the molecule's known structure identifier (e.g. SMILES, InChI, etc.). Other images may be produced during routine work and should also be placed in this directory, though user-produced images and graphics can be saved anywhere.
- `/inst` Files for rdkit integration (`/rdkit`), the API service (`/plumber`), and Shiny applications (`/apps`).
  - `/rdkit` Environment establishment and rdkit functions are located here. These will determine how R connects to the python environment to integrate rdkit into an R session as well as the files necessary to build the environment (e.g. "environment.yml"). Functions in the "py_setup.R" file should suffice for most use cases.
  - `/plumber` Environment establishment and API definition functions are located here. These will determine how requests to the API are routed and functionality are provided through http protocols in a RESTful manner. It comes complete with Swagger documentation available when the server is running.
  - `/apps` Environment establishment, general resources, and shiny application files are located here. Each application is contained within its own directory.

- `/logs` If logging functionality is turned on, logs will be written here according to the namespace of the log (e.g. logs written to the "db" namespace will be written to "logs/log_db.txt").
- `/R` directory; most general R functions are housed here or in one of the subdirectories.

Project Set Up

Running the compliance script at `"R/compliance.R"` will establish the project for you in most cases. It leverages several files to determine project settings; these are detailed here for clarity and customization options, with further details provided in the Compute Environments section. To accept the default settings, source the compliance file and move on to the Using DIMSpec section. This may take a while to resolve package dependencies. To customize your implementation, read on. Changing any of these settings is entirely optional.

Step 1 - Customize global environment settings

Several options are available to customize the use of DIMSpec to any given project component; settings are in the file "/config/env_glob.txt". These values are not set at the system level to add flexibility across operating systems; they are instead session values that are available while a session is active.

Table 1: Customizing global settings in the "/config/env_glob.txt" file

Setting	Type	Description
DB_TITLE	String	The title to use for this implementation.
DB_NAME	String	The name of the database to create or use. For SQLite this should be the name of the database file.
EXPLICIT_PATHS	Logical	Whether or not file names are fully qualified with their path.
DB_BUILD_FILE	String	The .sql file name of the script used to build the database (e.g. "build.sql"; see Database Schema.
DB_BUILD_FULL	String	The .sql file name of the fallback build script that should be used if the sqlite3 command line interface (CLI) tool is not available (e.g. "build_full.sql"; see Database Schema).
DB_DATA	String	The .sql file name of the data population script to run when populating data at build time (e.g. "populate_common.sql"; see Populating Data.
SQLITE_CLI	String	The name of the terminal command to launch the sqlite shell (e.g. `sqlite3`). This must be available in your PATH.
CONDA_CLI	String	The name of the terminal command to execute ana-

		/miniconda commands (e.g. `conda`). This must be available in your PATH.
INIT_CONNECT	Logical	Whether or not to connect to the database when starting a session by sourcing the compliance script.
LOGGING_ON	Logical	Whether or not to establish an environment to perform action logging, which will carry additional information about what functions in the DIMSpec tool set are doing (see Logger).
USE_API	Logical	Whether or not to activate the plumber application programming interface (API) for this session (see Plumber). If this is set to `TRUE`, the plumber service will launch in a background process by default and return control to the console.
INFORMATICS	Logical	Whether or not to establish an environment providing informatics support, primarily with RDKit. To streamline installation of only the database and R tools, set this to `FALSE`.
USE_RDKIT	Logical	Whether or not to use RDKit for informatics (requires python). If set to `FALSE`, the packages `BiocManager`, `ChemmineR`, and `rcdk` will be installed if not available, though support for these is not provided at this time.
USE_SHINY	Logical	Whether or not to establish an environment providing support for web applications provided as part of the project (defaults to `TRUE`).
SHINY_BG	Logical	[PLANNED FEATURE] Whether or not to launch shiny apps as part of a background process, making them immediately available from a web browser when the compliance script is executed (defaults to `FALSE`).

Step 2 - Customize R session settings in the "env_R.R" file

More customization options that require R are available to set up the project specifically for your application. Open the file "config/env_R.R" to customize these for your use. These values are not set at the system level to add flexibility across systems; they are instead session values that are available during use of the project, and many depend on settings from the section above, which will be applied automatically if they are not already set.

Table 2: Customizing settings specific to the R environment

Setting	Type	Description
DB_DATE	Date	The date the database file was last created, as determined by file properties. Override with a date value (e.g. `as.Date("2022-06-01")`)

DB_RELEASE	String	The major and minor release versions for this database.
DB_VERSION	Generated String	Combines the `DB_RELEASE` and `DB_DATE` (if built) values for a complete version of the database.
DB_PACKAGE	String	The name of the R package allowing connection to your database (e.g. `RSQLite` in most cases, but could be any database connection package).
DB_DRIVER	String	The name of the database driver function allowing connection to your database, which must be a function available in `DB_PACKAGE` (e.g. `SQLite`).
DB_CLASS	String	The class of an R object resulting from a call to the function defined by `DB_PACKAGE::DB_DRIVER` (e.g. `SQLite`); this will be used to search for and manage connections in the session.
DB_CONN_NAME	String	The name to be used for the R object database connection (e.g. `con` in most cases here); this defaults to a session variable named `DB_CONN_NAME` if it exists to facilitate independent management of multiple connections.
DEPENDS_ON	String Vector	The list of packages required by your project. The list provided is the bare minimum required for functionality in the project as distributed. Add more to expand functionality for your use cases if necessary.
EXCLUSIONS	String Vector	The list of files and directories to exclude from automatic loading when the compliance script is run.
IMPORT_MAP	String	Imports the mapping file determining relationships between import files and the database structure (see Importing Data); change the name of the CSV file (change also the function calling it if using formats other than CSV) to point to a different map.
LOGGING_ON	Logical	Whether to activate logging functionality when a session begins. This defaults to the session variable named `LOGGING_ON` and, if not present, to `TRUE`. If `TRUE`, adds the `logger` package to the dependency list.
VERIFY_ARGUMENTS	Logical	Whether or not to activate function argument verification for this project. The default of `TRUE` will check arguments provided to many functions for compliance with function expectations and is good for development work, but also slows down

		execution times. Set to `FALSE` to turn this off.
MINIMIZE	Logical	If `TRUE`, turns off both `LOGGING_ON` and `VERIFY_ARGUMENTS` to speed up execution time.
USE_API	Logical	Defaults to the global setting of `USE_API`. If `TRUE`, several options are provided to customize properties of the API. Set these as appropriate for advanced use cases; the defaults will make the API available on your local system, e.g. at http://127.0.0.1:8080.
USE_SHINY	Logical	Defaults to the global setting of `USE_SHINY`. If `TRUE`, a list of installed shiny applications will be available to your session under the named character variable `SHINY_APPS` which contains absolute paths to the application directories. These are launchable (and should resolve their environment) at any time during a session from the console using `shiny::runApp(SHINY_APPS["app_name"])` where `"app_name"` is the name of a shiny app in the variable. See the Mass Spectral Match for Non-Targeted Analysis (MSMatch) application that ships with this project as an example to match user supplied mass spectral data against the library.

Step 3 - Customize logger settings in the "env_logger.R" file

To provide support information about performance and support troubleshooting, a logging utility is provided with the project. Logs are managed by namespace and generated with the function `log_it` which uses the `logger` package for additional functionality (see Logger). Customization options for the format of these logging messages are provided, though under most circumstances should be left as-is to support reading logs back into a session. These values are not set at the system level to add flexibility across systems; they are instead session values that are available during use of the project, and many depend on settings from the sections above, which will be applied automatically if they are not already set.

Three support functions are also provided in this file to update the logger settings during a session (`update_logger_settings`), read logs from a file back into the session (`read_log`), and convert a log file into a session data frame for deeper inspection (`log_as_dataframe`).

Environment set up files that follow the same approach are also provided for rdkit integration, the plumber API server, and shiny web applications; these are not detailed here and should only be changed when necessary. See those sections in Technical Details for more information.

Table 3: Customizing logger settings; generally, these should not be changed, but LOGGING is easily extended for developing different applications of DIMSpec.

Setting	Type	Description
LOG_DIRECTORY	String Path	The relative path to the project directory housing logs. This defaults to the session variable named `LOG_DIRECTORY` and, if not present, to `"logs"`. If that directory is not present, it will be created.
layout_console	Function String	The format to use when printing logs to the console, by default using the `logger::layout_glue_generator`, but could be any `logger` generator.
layout_file	Function String	The format to use when printing logs to a file, by default interpreted by `logger::layout_glue_generator`, but could be any `logger` generator.
log_remove_color	Regex String	A regular expression describing color formatting to strip out when reading logs back into a data frame. If printing to the console in RStudio, colors will be maintained. This should coordinate with the `layout_console` format.
log_split_column	Regex String	A regular expression describing character formatting used to split log records into columns when reading logs back in as a data frame object with the function `log_as_dataframe`. This should always be coordinated with the `layout_file` format.
LOGGING	List	A nested list object defining logging settings for different namespaces. Each must include the following named settings: • ***log*** determines whether to log a given namespace (`TRUE/FALSE`); • ***ns*** is the character scalar namespace called as part of `log_it`; • ***to*** is the destination of the log message, one of `"file"`, `"console"`, or `"both"`; • ***file*** is the file path to the log file which will be created if it does not exist; • ***threshold*** determines what level at which to log messages (e.g. setting a threshold of `"info"` will not log messages at the `"trace"` level; see the logger package documentation for details). New namespaces can be added during the session if desired, but this list should define the most common ones. More information about the logging environment is provided in the Logger section of Technical Details.
LOGGING_WARNS	Logical	Whether to log all warning messages generated during

		this session by default.
LOGGING_ERRORS	Logical	Whether to log all error messages generated during this session by default.

Using DIMSpec

There are several R packages required for this project, so initial set up may take some time. To streamline this process once set up is complete, a compliance script is available that will install and load required packages; run `source("R/compliance.R")` in the console to establish the runtime environment. See References for the complete list of library dependencies. Based on project settings, components can be turned on or off as desired for lighter weight applications. In many cases helper functions are available to turn these components back on during an active session without interrupting the current environment. The following sections assume the compliance script has run and that all functions are available. At any time, use `fn_guide()` or `fn_help("fn")` where `"fn"` is the name of a function (quoted or unquoted) to view function documentation from within R.

Database Connections

Connecting to an Existing Database

This project uses SQLite by default as a portable database engine where the database is contained to a single file. To connect a project to a particular database (e.g. you have multiple databases for different projects), simply change the value of `DB_NAME` in "env_glob.txt" prior to sourcing the compliance file. The database distributed with the project contains mass spectral data for per- and polyfluoroalkyl substances as an example. It (and any databases created using this project), opens in write-ahead logging (WAL) mode for speed and concurrency. This does generally require the database file to be present on the same machine as the project but allows installation on instrument controllers that may not comply with network security restrictions. As with all SQLite databases, foreign key enforcement must be turned on when connecting with `pragma foreign_keys = on;` the `manage_connection` function takes care of this and other connection management aspects automatically and is the recommended way to connect and disconnect to DIMSpec databases. Call `manage_connection(reconnect = FALSE)` to close the connection. Calling `manage_connection` calls `DBI::dbConnect` and `DBI::dbDisconnect` with certain checks and parameter defined side effects to manage the connection.

Creating a New Database

Tooling to create a new SQLite database using this schema are built into the project; functions are in the "R/db_comm.R" file and help documentation is available from within

the project using the `fn_guide()` and `fn_help` functions. When creating a new database, prior to sourcing the compliance file, set options in the "env_glob.txt" and "env_r.R" files appropriately. If the file identified by `DB_NAME` does not exist it will be created according to the SQL scripts selected as `DB_BUILD_FILE` and `DB_DATA`; edit those files if necessary for your use case. To build a new, empty database users need only set `DB_NAME` to a file that does not exist in the project directory, and `DB_DATA` to "populate_common.sql" which contains the majority of source data necessary to populate normalization tables (see the Database Schema and Populating Data sections for more detail).

Alternatively, once the compliance file has been sourced, a new database may be created directly from R with the `build_db` function; this function takes as default values those provided in the environment, but you can at any time define different specifications. For example, to create a new database with a different SQL definition and population script use:

```
build_db(
  db = "new_database.sqlite",
  build_from = "this_file.sql",
  populate = TRUE,
  populate_with = "new_data.sql",
  connect = FALSE
)
```

If a connection already exists that you wish to maintain in the session, be sure to call this with `connect = FALSE` in order to not drop the connection (see next section for managing multiple connections). If you do not wish to maintain a connection to the previous database, this can be safely called with `connect = TRUE` (the default) and the prior connection will be replaced with the new one.

Connecting to Multiple Databases

If your project needs to connect to multiple databases, separate connections can be made and managed within a single R session. For convenience, the supplied `manage_connection` function will apply to the database and connection object defined in the setup files (see Project Set Up). Enable a second connection alongside existing connections (e.g. the one created in the previous section) with `manage_connection(db = "new_database.sqlite", conn_name = "con2")` . There is no limit to the number of connections that can be made in this manner, and the WAL will be flushed each time this function is called if no other connections exist.

Using a Database Connection in an R Session

If `INIT_CONNECT = TRUE`, sourcing the compliance file will establish a connection to the database named in `DB_NAME` and make the connection available as an R session object with

the name defined by `DB_CONN_NAME` (the default is `con`). Several convenience functions are available with those options set.

Functions from the `dplyr` package support database operations as implemented in the `dbplyr` package, meaning you can work with database objects using the "tidyverse" as if they were local objects (e.g. `tbl(src = con, "contributors")` where `con` is your database connection object and "`contributors`" is the name of a database table or view). Simple database operations (e.g. filters, joins, column selection, etc) are supported and the resulting object is an external pointer to a lazy database query; to pull data as a data frame (e.g. necessary to join a local data frame with a database query result) use `collect()` on the tbl object. There are, however, some tasks (e.g. complicated or programmatic queries) where that may prove insufficient. In that case, two options are available.

The connection object fully supports direct communication for SQL queries through the `DBI` package and is likely a familiar option for users comfortable with SQL. To continue the example, `dbGetQuery(con, "select * from contributors")` will return the same data as in the `tbl` example above, except that it returns a data frame rather than a pointer object.

For users less familiar with SQL, the function `build_db_action` is provided to support nearly all database operations. There may be edge cases where it fails. Results from the following function are equivalent to the `dbGetQuery` result but will construct the query programmatically, allowing for the passing of arguments and always returning a data frame:

```
build_db_action(
  action = "select",
  table_name = "contributors"
)
```

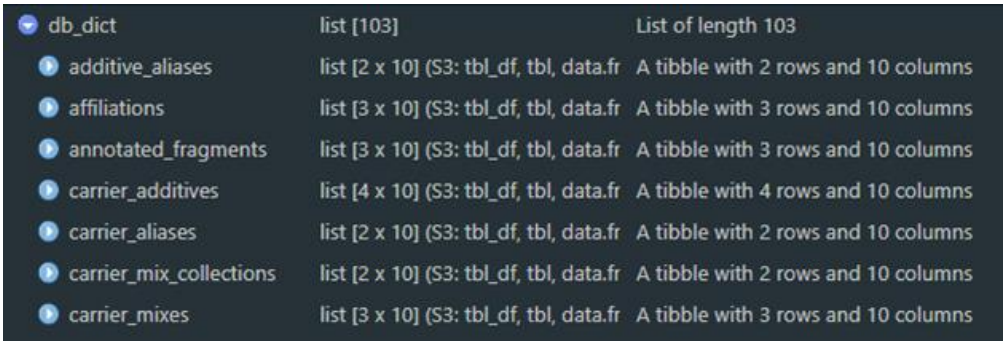
As this function performs argument verification and SQL interpolation to protect queries from unintended side effects, this is the recommended manner to directly interact with the database for anything other than basic queries. It supports typical database actions (including `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, as well as a custom `GET_ID` action that returns an integer vector of the `id` column for all records matching the query) and operations (`GROUP BY`, `ORDER BY`, `DISTINCT`, `LIMIT`). Search and filter options can be passed programmatically to `match_criteria` as a list and are parsed by the `clause_where` function.

Queries do not have to be executed; set the argument `execute = FALSE` to examine queries prior to execution or save common queries for reuse. See the full function reference with for advanced use of the `build_db_action` and `clause_where` functions with `fn_help`.

Inspecting Database Properties

Code decoration conventions used in the SQL files enable reading table definitions and properties from SQLite into R with the function `pragma_table_info`. Supply the name of a database table or view to get information about that table; different connections can also

be used for comparison if desired. This is the interactive version; a version in JSON format can be saved using `save_data_dictionary`. This saved file is loaded during the compliance script as object `db_dict` which is a named list of data frames; names correspond to database entities. This can be regenerated and brought back into the R session at any time (see `data_dictionary`) and should be updated if modifications are made to the underlying schema.



db_dict	list [103]	List of length 103
additive_aliases	list [2 x 10] (S3: tbl_df, tbl, data.fr	A tibble with 2 rows and 10 columns
affiliations	list [3 x 10] (S3: tbl_df, tbl, data.fr	A tibble with 3 rows and 10 columns
annotated_fragments	list [3 x 10] (S3: tbl_df, tbl, data.fr	A tibble with 3 rows and 10 columns
carrier_additives	list [4 x 10] (S3: tbl_df, tbl, data.fr	A tibble with 4 rows and 10 columns
carrier_aliases	list [2 x 10] (S3: tbl_df, tbl, data.fr	A tibble with 2 rows and 10 columns
carrier_mix_collections	list [2 x 10] (S3: tbl_df, tbl, data.fr	A tibble with 2 rows and 10 columns
carrier_mixes	list [3 x 10] (S3: tbl_df, tbl, data.fr	A tibble with 3 rows and 10 columns

Figure 1a. An example of the data dictionary object.

```
> db_dict$samples
# A tibble: 10 x 10
    cid table_name table_comment   name  type  notnull    pk hidden unique field_comments
  <int> <chr>      <chr>           <chr> <chr>   <int> <int>  <int> <lgl>  <chr>
1     0 samples    Samples from w~ id    INTE~       1     1      0 FALSE  primary key
2     1 samples    Samples from w~ mzml~ TEXT        1     0      0 FALSE  user-defined ~
3     2 samples    Samples from w~ desc~ TEXT        1     0      0 FALSE  user-defined ~
4     3 samples    Samples from w~ samp~ INTE~       1     0      0 FALSE  foreign key t~
5     4 samples    Samples from w~ sour~ TEXT        0     0      0 FALSE  citation for ~
6     5 samples    Samples from w~ samp~ TEXT        1     0      0 FALSE  generator of ~
7     6 samples    Samples from w~ gene~ INTE~       1     0      0 FALSE  data generate~
8     7 samples    Samples from w~ gene~ TEXT        1     0      0 FALSE  datetime the ~
9     8 samples    Samples from w~ ms_m~ INTE~       0     0      0 FALSE  foreign key t~
10    9 samples    Samples from w~ samp~ INTE~       0     0      0 FALSE  foreign key t~
```

Figure 1b. Details of the "samples" table from the data dictionary.

Relationships between database entities can also be queried programmatically. Use the `er_map` function to read the same decoration convention in the SQL definitions to extract relationships. An object is created during the compliance script as `db_map` to make it available to your session. This results in a nested list with names corresponding to database entities, and elements describing the object name, its type, which table(s) and column(s) it references, which table(s) reference it, which table(s) it normalizes, and which view(s) use it.

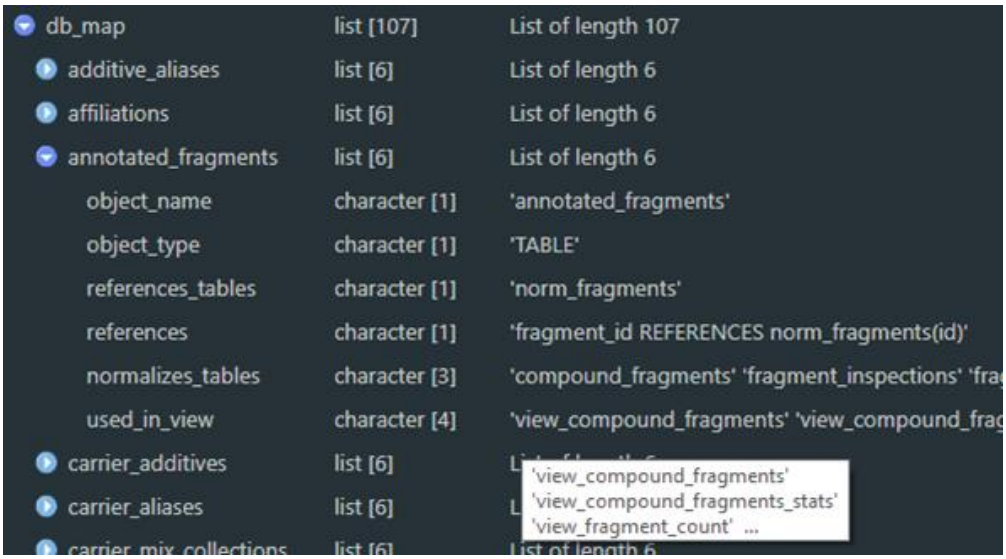


Figure 2a. An example of the R object structure of an entity map as a list.

```
> db_map$samples
$object_name
[1] "samples"

$object_type
[1] "TABLE"

$references_tables
[1] "contributors"         "ms_methods"           "norm_carriers"
[4] "norm_generation_type" "norm_sample_classes"

$references
[1] "sample_class_id REFERENCES norm_sample_classes(id)"
[2] "ms_methods_id REFERENCES ms_methods(id)"
[3] "sample_contributor REFERENCES contributors(id)"
[4] "generation_type REFERENCES norm_generation_type(id)"
[5] "sample_solvent REFERENCES norm_carriers(id)"

$normalizes_tables
[1] "mobile_phases"  "peaks"          "qc_data"        "qc_methods"     "sample_aliases"

$used_in_view
[1] "view_contributors" "view_masserror"    "view_samples"
```

Figure 2b. Details of the "samples" table from the data entity map object in Figure 2a.

Using the Application Programming Interface (API)

Application Programming Interfaces (APIs) enable software components to communicate with each other. Most modern machine communication happens through APIs. In the context of this project, an API server is launched using the `plumber` package to reduce computational load on R sessions or shiny applications and ensure consistent results across multiple sessions. It does not have to be used (set `USE_API = FALSE` in "env_glob.txt" to turn it off) but is encouraged and is a requirement for all shiny applications that ship with this project.

The compliance script launches this in a background process by default at http://localhost:8080. Use `api_open_doc` to open the documentation page directly in a browser. To start the service manually from an interactive session and load the documentation immediately for exploration and testing, use `api_reload(background = FALSE)`; if it is already running in a background process and desirable to launch a second service (e.g. for testing new endpoints or changes to existing ones), set the `pr` parameter to a different name and the `on_port` parameter to an open port (it will fail if the port is already in use). Documentation is produced by Swagger and is interactive, allowing for users to enter values and get both the return and the URL necessary to execute that endpoint (Figure 3). See the Plumber section in Technical Details for more information. If the compliance script is run with `USE_API = FALSE` and `api_reload` is not available, it may be more intuitive to use `start_api`.

Endpoints for many predictable read and search interactions are available. Session variables define the connections, and communication and control functions default to those expected values for streamlining (e.g. functions like `api_reload`, `api_open_doc`, and `api_endpoint` may be called without referring explicitly to a session object or URL for the current project).

The main interactivity with the API from an R session or shiny application is through the `api_endpoint` function. The first argument (i.e. `path`) should always be the endpoint being requested. Additional named parameters are then passed to the API server; the same example endpoint result in Figure 3 called from the console would be

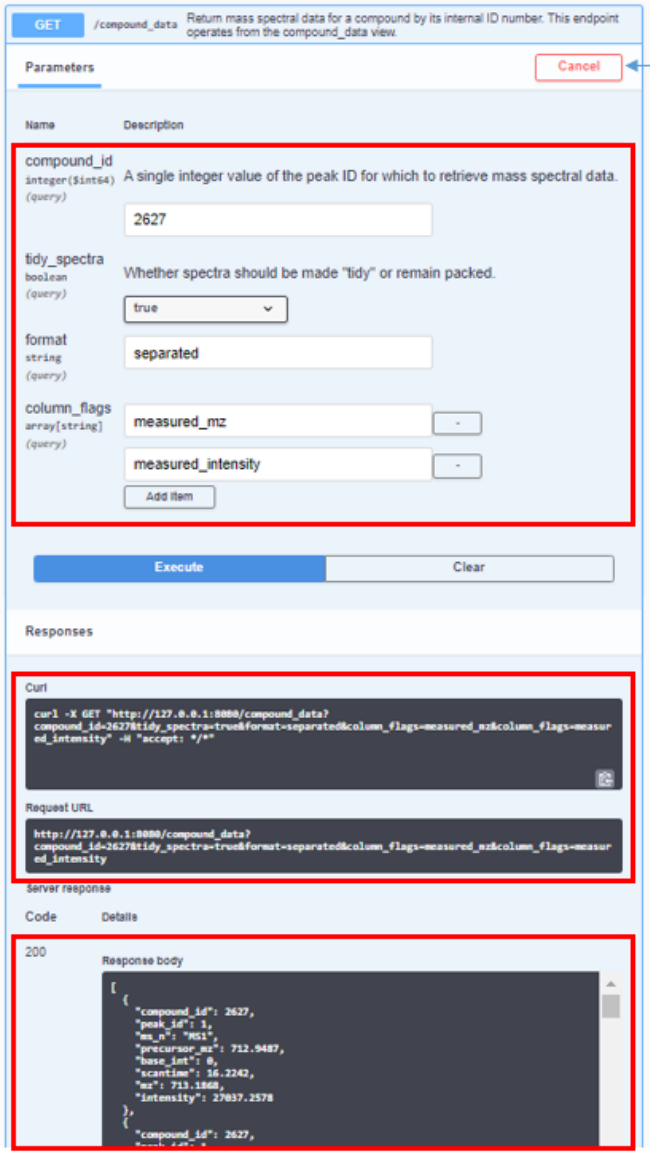
```
api_endpoint(
  path                          =                          “compound_data”,
  compound_id                             =                          2627,
  return_format                       =                      “data.frame”
)
```

with an example of the results in Figure 4. Endpoints of most use to those using the service will vary according to needs and are detailed in the Plumber section in Technical Details. Call them with `api_endpoint(path = *X*)` and any other arguments required by the endpoint. Paths listed here are likely of most use:

- **“_ping”**, **“db_active”**, and **“rdkit_active”** indicate that the server is alive and able communicate with the database and rdkit, respectively;
- **“list_tables”** and **“list_views”** return available tables and views respectively;
- **“compound_data”** and **“peak_data”** return mass spectrometry data associated with a compound or peak and must be called with `compound_id` or `peak_id` equal to the database index of the request; in most cases these should be called with `return_format = "data.frame"`;
- **“table_search”** is a generic database query endpoint analog for `build_db_action` to construct `SELECT` queries and has the most parameters for flexibility; for more information see `fn_help(build_db_action)` for details; relevant parameters are summarized here:
  - *table_name* should be the name of a single table or view;

- *column_names* determine which columns are returned;
- *match_criteria* should be a list of criteria for the search convertible between R lists and JSON as necessary; values should generally follow the convention `list(column_name = value)` and can be nested for further refinement using e.g. `list(column_name = list(value = search_value, exclude = TRUE))` for an exclusion search (see `fn_help(clause_where)` for additional details); when called via `api_endpoint` R objects can be passed programmatically;
- *and_or* should be either `"AND"` or `"OR"` and determines whether multiple elements of `match_criteria` should be combined in an AND or OR context (e.g. whether `list(column1 = 1, column2 = 2)` should match both or either condition);
- *limit* is exactly as in the SQL context; leave as `NULL` to return all results or provide a value coercible to an integer to give only that many results;
- *distinct* is exactly as in the SQL context and should be either TRUE or FALSE;
- *get_all_columns* should be either `TRUE` or `FALSE` and will ensure the return of all columns by overriding the `column_names` parameter;
- *execute* should be either `TRUE` or `FALSE` and determines whether the constructed call results are returned (`TRUE`) or just the URL (`FALSE`); and
- *single_column_as_vector* should be either `TRUE` or `FALSE` and, if `TRUE`, returns an unnamed vector of results if only a single column is returned.

These and other endpoints can be easily defined, expanded, or refined as needed to meet project requirements. Use `api_reload` to refresh the server when definitions change, or test interactively prior to deployment using Swagger by launching a separate server either by opening the plumber file and clicking the "Run API" button in RStudio, or using the `api_start` or `api_reload` functions as described above. To support eventual network deployment, any number of API servers may be launched manually on predefined ports to allow for load balancing.



Expand one of the endpoint boxes to see the "Try it out" button. Click it to enable parameter entries, then click "Execute" to submit the request or "Cancel" to close the endpoint tester.

Endpoint parameters may be entered directly in the testing system and are somewhat limited in scope. If parameters do not match expectations, the response will fail with a message.

These URLs can be included in any application to execute the same request of the API server.

Data are returned in JavaScript object notation (JSON). These can be formatted in different ways using the `return_format` parameter of the `api_endpoint` function.

Figure 3. Screen shot and descriptions of the interactive Swagger documentation page for the endpoint `/compound_data`, available using `api_open_doc()`. Click the "Try It Out" button to activate the testing mode.

```
> api_endpoint("compound_data", compound_id = 2627, return_format = "data.frame")
Endpoint http://127.0.0.1:8080/compound_data?compound_id=2627 is valid.
No encoding supplied: defaulting to UTF-8.
    compound_id peak_id ms_n precursor_mz   base_int scantime       mz  intensity
1          2627       1  MS1     712.9487       0.00  16.2242 713.1868   27037.26
2          2627       1  MS1     712.9487   15094.41  16.2023 712.9501   15094.41
3          2627       1  MS1     712.9487   15094.41  16.2023 713.1851   34809.98
4          2627       1  MS1     712.9487   17113.98  16.2132 712.9506   17113.98
5          2627       1  MS1     712.9487   17113.98  16.2132 713.1822   16714.91
6          2627       1  MS1     712.9487   21320.14  16.2679 712.9485   21320.14
```

Figure 4. Screen shot of the result of calling the same API endpoint as in Figure 3 from an R session.

Using rdkit

For chemometrics integration, `rdkit` is made available as part of the project. This user guide does not provide details about `rdkit`; users are instead directed to the documentation for details. All functionality provided as part of `rdkit` is supported with some limitations through the `reticulate` package. In most cases the required environment should resolve during the compliance script. On certain systems it may be desirable to install the environment manually (instructions in the Python section of Technical Details).

Once an R session has activated and bound to a python environment it cannot be deactivated, but instead must be terminated to drop this binding. Once bound to a session object, all `rdkit` functions are accessible as a list of functions (just as in any python integration using reticulate) following `rdkit` module structures e.g.

```
rdk$Chem$MolFromSmiles("CN1C=NC2=C1C(=O)N(C(=O)N2C)C")
```

Though these can be chained together or piped, for stability it is recommended to store the return of each call as a variable; returned objects may not always be readily used in further functions.

A few custom R functions are made available to assist with the process. The implementation will depend on the environment definition found in "inst/rdkit/env_py.R" but in the standard use case will result in a session object named `rdk` tied to a python environment named "nist_hrms_db" using packages built from conda forge. See the function reference guide using `fn_guide()` for additional details, but the following functions are likely the most useful:

- `setup_rdkit` is a convenience function that should install and bind to python in a session;
- `rdkit_active` is the main check to determine whether or not rdkit has been bound to the current session and allows for setting multiple bindings if desired by setting `rdkit_ref` to a different value, and will trigger `setup_rdkit` if called with `make_if_not = TRUE`;
- `molecule_picture` creates a graphic of a molecular model from structural notation and is an example of `rdkit` functionality; and
- `rdkit_mol_aliases` generates machine-readable structural notation in a variety of formats (e.g. InChI and InChIKey) given a notation with a known format and can interchange between these to create molecular aliases; all formats supported by `rdkit` are attempted if `get_aliases = NULL` (Figure 5) but generally these would be specific by project needs; results that fail or are blank are removed and the return is by default a data frame to support any number of identifiers with one pass.

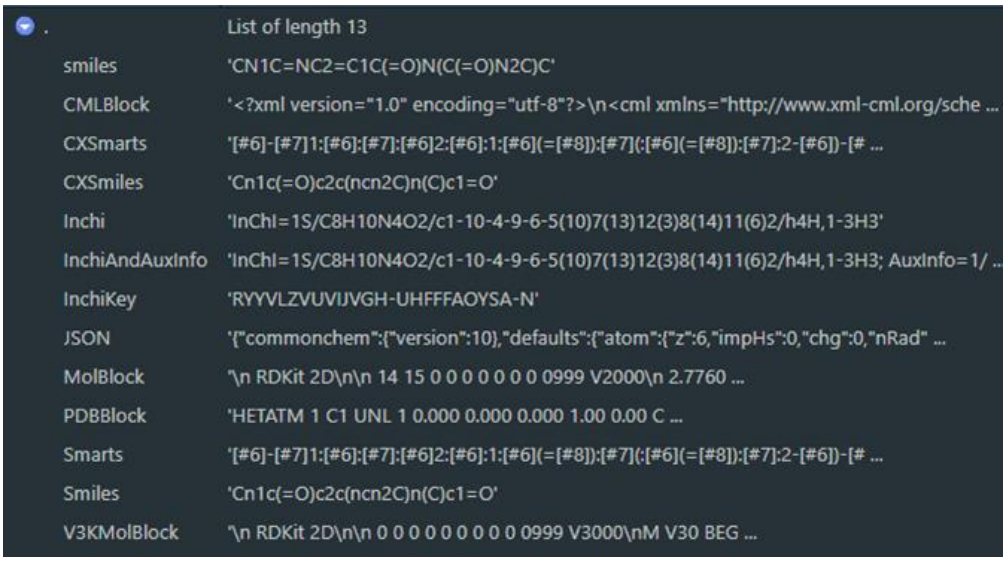


Figure 5. All molecular aliases as seen in the RStudio viewer for results of a call to `rdkit_mol_aliases("CN1C=NC2=C1C(=O)N(C(=O)N2C)C", get_aliases = NULL)`

Logging

Logging messages for statuses, information, warnings and errors are provided throughout functions used in this project and is executed through the `log_it` function. This function builds on top of the logger package to construct, decorate, and write to file any logging messages necessary, and offers console messages in case logger is unavailable. If logging is enabled and the `logger` package available, logs may also be written to files in the "`logs`" directory and later retrieved with the utility functions `read_log` and `log_as_dataframe`, whose first parameter is the name of the file to read from the `/logs` directory. Logs written to disk by default are separated by namespace (e.g. `/logs/log_db.txt` vs `/logs/log_api.txt`) to facilitate support, but output files may be defined as any available .txt file path and will be appended to existing files. Logs may look odd if viewed directly as they include text decorations to display in the console.

Settings are available for five namespaces by default (see Logger and Project Set Up for more details) as established by the "config/env_logger.R" file; more can be enabled at any time using the `add_unknown_ns` and `clone_settings_from` parameters of `log_it`. Logs can then be generated from within any function using e.g.:

```
log_it(
  log_level = "info",
  msg = "Log message text",
  log_ns = "global"
)
```

where `log_level` is the category of message, `msg` is the message itself, and `log_ns` is the namespace. Settings defined in the `LOGGING` session variable determine how logs are processed. Each message produced with `log_it` includes the timestamp, namespace, status (i.e. `log_level`), function calling the message, and the message itself. While `log_it` will print to the console messages of any level, `log_level` should be one of the supported logging levels (trace, debug, info, success, warn, error, or fatal) to integrate with `logger`, which is required if the logging message is to be written to a log file.

Users developing on top of this infrastructure are encouraged to take advantage of the logging functionality and make liberal use of the `log_it` function to ease debugging and maintenance.

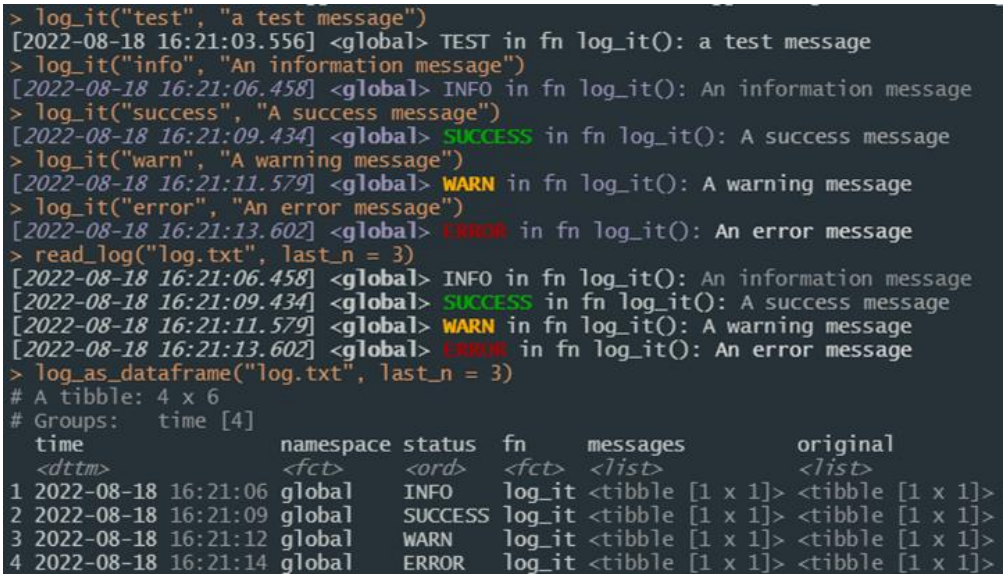


Figure 6. Example uses of `log_it` to create logging messages.

Using Shiny Applications

The Shiny package enables web applications written using R, which often meaningfully make custom processing code like that written for this project available to broader audiences. Additionally, inputs can easily be type verified and restricted to preset expectations. When the compliance script is run, a named vector of available shiny apps will be available as `SHINY_APPS`. These can be started with the `start_app(app_name = X)` where `X` is the name of the application as found in `names(SHINY_APPS)`. Shiny apps are fluid and responsive; will automatically arrange themselves to best fit your browser size and can be custom designed with any layout or functionality. By default all communication with the database is routed through the plumber API.

This allows environment resolution to launch applications directly from the console, without any need to run the compliance script. Launching an app is then possible directly from the console (or batch file shortcuts which could be included in later updates) using e.g.

```
shiny::runApp("inst/apps/table_explorer")
```

from the project directory.

Three shiny applications ship with this project as of the time this document was written.

- `table_explorer` allows users to explore database tables and views by selecting it from a drop-down list and details definitions and connections to other tables and views; this app should be amenable to any database created with DIMSpec and is detailed in its own section;
- `spectral_match` allows users to upload an mzML file of mass spectral data and search user-defined features of interest by mass to charge ratio and chromatographic retention time for matches in the database for both known compounds and annotated fragments, while providing contextual information about the method and samples used to generate reference spectra. The MSMatch application is detailed in its own section.
- `msqc` allows users to perform the quality control evaluation of potential imported data and generates the necessary JSON object to be incorporated into the database. Additional information, including the workflow, will be described in a forthcoming ROA. The MSQC application is detailed in its own section.

An application template is also included which should accelerate development of additional applications on top of the DIMSpec infrastructure to facilitate project needs.

Importing Data

For now, data imports are only supported from the command line and those generated by the NIST Non-Targeted Analysis Method Reporting Tool (NTA MRT). That tool is a macro-enabled Microsoft Excel® workbook available on GitHub that

> "…allows for the controlled ontology of method data reporting and the export of the data into a single concise, human-readable file, written in a standard JavaScript Object Notation (JSON)."

Users fill out the workbook annotating features of interest and associated fragmentation identities. Generated method files are submitted alongside the mzML file (converted from instrumentation output using Proteowizard's msConvert software (Adusumilli, Ravali and Mallick, Parag 2017). After quality control checks are performed, the resulting JSON object holds everything necessary to import data into the database.

Data passing quality control checks (see the Mass Spectral Quality Control section for a shiny application to check quality control aspects of mzML files) are imported using functions found primarily in the "`/R/NIST_import_routines.R`" file. Field mapping is defined by the "`/config/map_NTA_MRT.csv`" file, which contains a list of import file elements and their properties, with connections for each to their destination tables and columns; individual elements are resolved by the `map_import` function which does much of the transformation. New maps can be created and used in support of other import formats in the future, and as the import functions are heavily parameterized they may need to be customized.

The order of operations is controlled largely by the pipeline function `full_import` which is the typical use case method for importing data. That function will check that the import file(s) include requirements and recommendations as defined in the file at "`/config/NIST_import_requirements.json`" which is a JSON list of expected elements and headers within each element and whether the elements are required. When using the NTA MRT format and process to import data the default arguments to this function and the import map should not be changed, but flexibility is supported by `full_import` having a nearly exhaustive list of parameters passed to underlying functions to resolve each database node in the required order (contributors, methods, descriptions, samples, chromatography, quality control, peaks, compounds, and fragments; see SQL Nodes in Technical Details for more details about schema nodes); parameters are passed largely by name matches for underlying functions using `do.call`. The import process is only available from the console, provides logging (if enabled) throughout, and fully supports batch imports from a list of import files read in via `jsonlite::fromJSON(readr::read_file(X))` where `X` is a vector of file paths. Files may alternatively be imported one at a time directly from JSON files using the `file_name` parameter and leaving the `import_object` parameter as `NULL`. A live connection to the database is required, and when additional information is needed (e.g. to resolve or add unknown controlled table entries), users will be prompted at the command line during the process.

Alternatively, data can be imported when a database is built or rebuilt from comma-separated value (CSV) files. This process is not likely amenable to many projects as it requires data indices be prepopulated and accurately cross-linked across CSV files, with one CVS file for each database table being populated; this should be considered if data are already in a database-like format and can be easily cross-linked, in which case only the table and column mappings need be solved. Several such files are used to populate a "clean" database install with certain controlled vocabulary and reference tables (see files "`/config/populate_common.sql`" and the "`/config/data/`" directory). Contact these authors for assistance with using the NTA MRT and msconvert process, or conversion of data into the DIMSpec schema if you feel a project's data would be amenable to the database structure described in this document.

Ending Your Session

Unclosed database connections can have unintended consequences. Generally, connections to the database during a session should be managed with `manage_connection` which allows for both disconnect and reconnect (to flush the WAL and establish a new connection). The API server will need to be spun down separately using `stop_api`. Alternatively, and to preserve any data frame objects that may have been created as external pointers (i.e. as `dplyr tbls`), when users finish with their connection needs they may use the convenience function `close_up_shop`. Connections may not flush completely in all cases. If users notice the `-shm` and `-wal` files are still open in the directory, the best way to flush them is to establish a new connection and then disconnect from it, using either `manage_connection` or `DBI::dbConnect/DBI::dbDisconnect`.

Updating the Schema

At the time this book was written, the schema should be well defined for most use cases. Extensions can however be added at any time to suit project-specific needs. To avoid data loss, it is recommended that any table extensions be performed directly in SQL and those commands saved to an SQL script. Views can be added freely as required. If users of this database framework apply any schema extensions, the authors would be interested in learning about both the need and the implementation so it may be evaluated for inclusion in future versions.

This concludes the User Guide for the Database Infrastructure for Mass Spectrometry. The following section contains technical details about the implementation and user customization.

Technical Details

This section contains additional technical details that may be of interest to advanced users, and for future reference as DIMSpec schema and tools mature.

Database Schema

The schema for the underlying database is defined by a series of SQL scripts in the `config` directory. Data are structured in a series of "nodes" and are detailed in this section. If the sqlite3 CLI is available, these are created by a script using a series of `.read` commands, one for each node defined. See the file at "`/config/build.sql` for the standard implementation.

Schema files are used from within R as part of the database build routine (see `build_db`) using the `shell` (on Windows) or `system` (on Unix-likes) functions. If the sqlite3 CLI is not available, a fully qualified native SQL script (such as the one provided at `/config/build_full.sql`) can be generated by `create_fallback_build` to build and populate in one step by parsing CLI commands in the SQL scripts to build underlying statements directly; this is less customizable and will take considerably longer but serves as a bridge for when CLI tools are not installed.

SQL Nodes

Each node file defines the tables and views necessary to store and serve data for a set of conceptually related entities in the database. Code decorations are used to facilitate translatability to R. Entity definitions (e.g. CREATE TABLE commands) are separated by the defined text string `/*magicsplit*/`, which is used as string split points by database communication functions to return information about the database. Headers are defined as a long `/*=== ... ===*/` SQL comment with equal signs delineating beginning and ending. Table and column comments are defined as `/* ... */` SQL comments; one must be present for each entity (i.e. one for the table or view, and one for each column in that table or view). Comments are ignored by SQL and can be used to parse table definitions and return information to an R session. This is what allows R to parse the SQL files for Inspecting Database Properties by reading a table definition (see next page) to obtain entity properties Figure 7a, pull column comments directly from the definition Figure 7b, inspect mapping between entities (Figure 7c; here `db_map` is an object created by `er_map`), or pull information together into a formalized data dictionary (Figure 7d; here `db_dict` is read from a JSON object created by `save_data_dictionary`) and available for all database tables and views.

This allows for programmatic accessibility, as R sessions can now understand the linkages between tables easily. One implementation example is checking for, resolving, and automatically adding new normalization values with `resolve_normalization_value` for

import resolutions, but could be as simple as understanding that `fragment_id` here references the `norm_fragments` table Figure 7e. In addition, once relationships are in a structured format, applications like the Table Explorer can be built to display to users in a more natural manner the structure of the underlying data tables Figure 8.

The decorated table entity command:

```
/*magicsplit*/
CREATE TABLE IF NOT EXISTS annotated_fragments
  /* Potential annotated fragment ions that are attributed to one or more mass spectra. */
(
  id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    /* primary key */
  mz REAL NOT NULL,
    /* m/z value for specific fragment, derived */
  fragment_id INTEGER NOT NULL,
    /* smiles structure of fragment ion, can be NULL, user submitted */
  /* Check constraints */
  /* Foreign key relationships */
  FOREIGN KEY (fragment_id) REFERENCES
    norm_fragments(id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

can then be parsed to create session-available expressions of entities and relationships.

```
> pragma_table_info("annotated_fragments")
  cid          table_name        name    type notnull dflt_value pk hidden unique
1   0 annotated_fragments          id INTEGER       1         NA  1      0  FALSE
2   1 annotated_fragments          mz    REAL       1         NA  0      0  FALSE
3   2 annotated_fragments fragment_id INTEGER       1         NA  0      0  FALSE
```

Figure 7a. An example of using `pragma_table_info` *to explore table definitions.*

```
> pragma_table_info("annotated_fragments", include_comments = TRUE)[, c(2,4,11)]
          table_name        name                                                field_comments
1 annotated_fragments          id                                                   primary key
2 annotated_fragments          mz                       m/z value for specific fragment, derived
3 annotated_fragments fragment_id smiles structure of fragment ion, can be NULL, user submitted
```

Figure 7b. Comments can be easily added to the definition.

```
> db_map$annotated_fragments
$object_name
[1] "annotated_fragments"

$object_type
[1] "TABLE"

$references_tables
[1] "norm_fragments"

$references
[1] "fragment_id REFERENCES norm_fragments(id)"

$normalizes_tables
[1] "compound_fragments"   "fragment_inspections" "fragment_sources"

$used_in_view
[1] "view_compound_fragments"       "view_compound_fragments_stats" "view_fragment_count"
[4] "view_fragment_mz_stats"
```

Figure 7c. Entity mapping can also be parsed directly from the SQL decoration convention.

```
> db_dict$annotated_fragments
# A tibble: 3 x 10
    cid table_name          table_comment    name  type  notnull    pk hidden unique field_comments
  <int> <chr>               <chr>            <chr> <chr>   <int> <int>  <int> <lgl>  <chr>
1     0 annotated_fragments Potential annot~ id    INTE~       1     1      0 FALSE  primary key
2     1 annotated_fragments Potential annot~ mz    REAL        1     0      0 FALSE  m/z value for~
3     2 annotated_fragments Potential annot~ frag~ INTE~       1     0      0 FALSE  smiles struct~
```

Figure 7d. Entity mapping can be saved to disk and recalled conveniently using the SQL decoration convention.

```
> ref_table_from_map("annotated_fragments", table_column = "fragment_id")
[1] "norm_fragments"
```

Figure 7e. Entity relationships can also be parsed to programmatically return foreign key relationships.

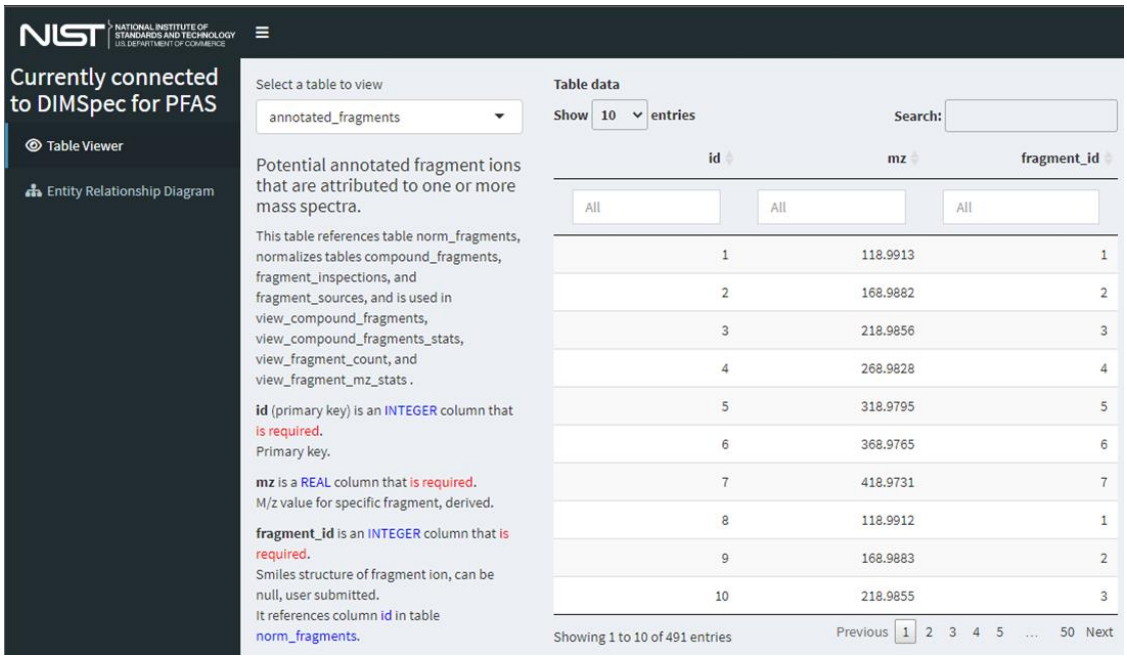
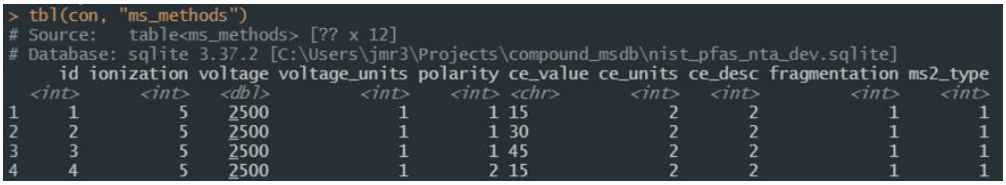


Figure 8. Decorated SQL definitions provide R with metadata regarding the database. The "Table Explorer" shiny application allows visual exploration of both data and context for any given database entity. With the project active and the compliance file sourced, launch this application in your browser from the console with `shiny::runApp(SHINY_APPS['table_explorer'])`.

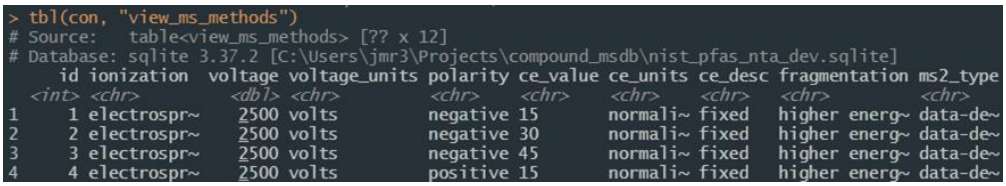
The following subsections contain summary information about each of the database nodes, its purpose, and the tables and views held within. Some of this information is subject to change as the database schema is refined and maintained. See the full database schema definition as a JSON object in the project directory as a file ending in "`_data_dictionary.json`". Entities found in each node and snapshots of their structure from the complete entity relationship diagram (ERD) are provided here; generally, node tables are in a color, automatic views are in grey, and functional views are in white.

Some views are automatically generated (see the flag "`[autogenerated by sqlite_auto_view()]`" in the description) to display human-meaningful values instead of the index linkages for normalized columns, e.g. the `ms_methods` table can then be viewed in a "denormalized" way using the `view_ms_methods` view to get display values for normalized fields Figure 9.



```
> tbl(con, "ms_methods")
# Source:   table<ms_methods> [?? x 12]
# Database: sqlite 3.37.2 [C:\Users\jmr3\Projects\compound_msdb\nist_pfas_nta_dev.sqlite]
     id ionization voltage voltage_units polarity ce_value ce_units ce_desc fragmentation ms2_type
  <int>      <int>   <dbl>         <int>    <int> <chr>       <int>   <int>         <int>    <int>
1     1          5    2500             1        1 15              2       2             1        1
2     2          5    2500             1        1 30              2       2             1        1
3     3          5    2500             1        1 45              2       2             1        1
4     4          5    2500             1        2 15              2       2             1        1
```

Figure 9a. Screenshot of the first four rows of the normalized database table 'ms_methods'.



```
> tbl(con, "view_ms_methods")
# Source:   table<view_ms_methods> [?? x 12]
# Database: sqlite 3.37.2 [C:\Users\jmr3\Projects\compound_msdb\nist_pfas_nta_dev.sqlite]
     id ionization  voltage voltage_units polarity ce_value ce_units ce_desc fragmentation ms2_type
  <int> <chr>         <dbl> <chr>         <chr>    <chr>    <chr>    <chr>   <chr>         <chr>
1     1 electrospr~    2500 volts         negative 15       normali~ fixed   higher energ~ data-de~
2     2 electrospr~    2500 volts         negative 30       normali~ fixed   higher energ~ data-de~
3     3 electrospr~    2500 volts         negative 45       normali~ fixed   higher energ~ data-de~
4     4 electrospr~    2500 volts         positive 15       normali~ fixed   higher energ~ data-de~
```

Figure 9b. Screenshot of the first four rows of the denormalized database table 'ms_methods'.

The Analyte Node

This node contains information relevant to analytical targets. This node does not contain analytical data, but rather identifying information and views to compare that identifying information with measurements held in the "data" node and is also linked to the "contributors" node. It contains two sub-nodes. One describes compounds and one describes fragments. These are linked through the "compound_fragments" table (which includes a link outside this node to the "peaks" table of the data node to allow for existence in either and flexible bidirectional linkages to be established for known links, without assuming presence in both. Both *_alias tables are normalized by `norm_analyte_alias_references`, and `fragment_sources` is normalized by `norm_generation_type` (described in the data node but also generated here for modularity and not shown below).

Entity Name	Description
Tables	
annotated_fragments	Potential annotated fragment ions that are attributed to one or more mass spectra.
compound_aliases	List of alternate names or identifiers for compounds
compound_categories	Normalization table for self-hierarchical chemical

	classes of compounds.
compound_fragments	Bidirectional linkage table to tie peaks and compounds to their confirmed and annotated fragments.
compounds	Controlled list of chemical compounds with attributable analytical data.
fragment_aliases	List of alternate names or identifiers for compounds
fragment_inspections	Fragment inspections by users for ions that are attributed to one or more mass spectra.
fragment_sources	Citation information about a given fragment to hold multiple identifications (e.g. one in silico and two empirical).
norm_analyte_alias_references	Normalization table for compound alias sources (e.g. CAS, DTXSID, INCHI, etc.)
norm_fragments	Normalization list of annotated fragments
norm_source_types	Validation list of source types to be used in the compounds TABLE.
Views	
compound_data	View raw data from all peaks associated with compounds.
compound_url	Combine information from the compounds table to form a URL link to the resource.
view_annotated_fragments	Measured fragments as compared with fixed masses
view_compound_aliases	[autogenerated by sqlite_auto_view()] View of "compound_aliases" normalized by "norm_analyte_alias_references".
view_compound_fragments	Fragments associated with compounds.
view_compound_fragments_stats	[DRAFT] Summarization view of statistics associated with compound fragments, including the number of times they have recorded, their measured masses, and ppm error as compared with nominal exact masses.
view_compounds	`[autogenerated by sqlite_auto_view()]` View of "compounds" normalized by "norm_source_types".
view_fragment_count	Number of fragments associated with compounds.
view_fragment_mz_stats	[DRAFT] Mean measures of measured_mz values - a supplementary calculation table.
view_fragment_sources	`[autogenerated by sqlite_auto_view()]` View of "fragment_sources" normalized by "norm_generation_type".

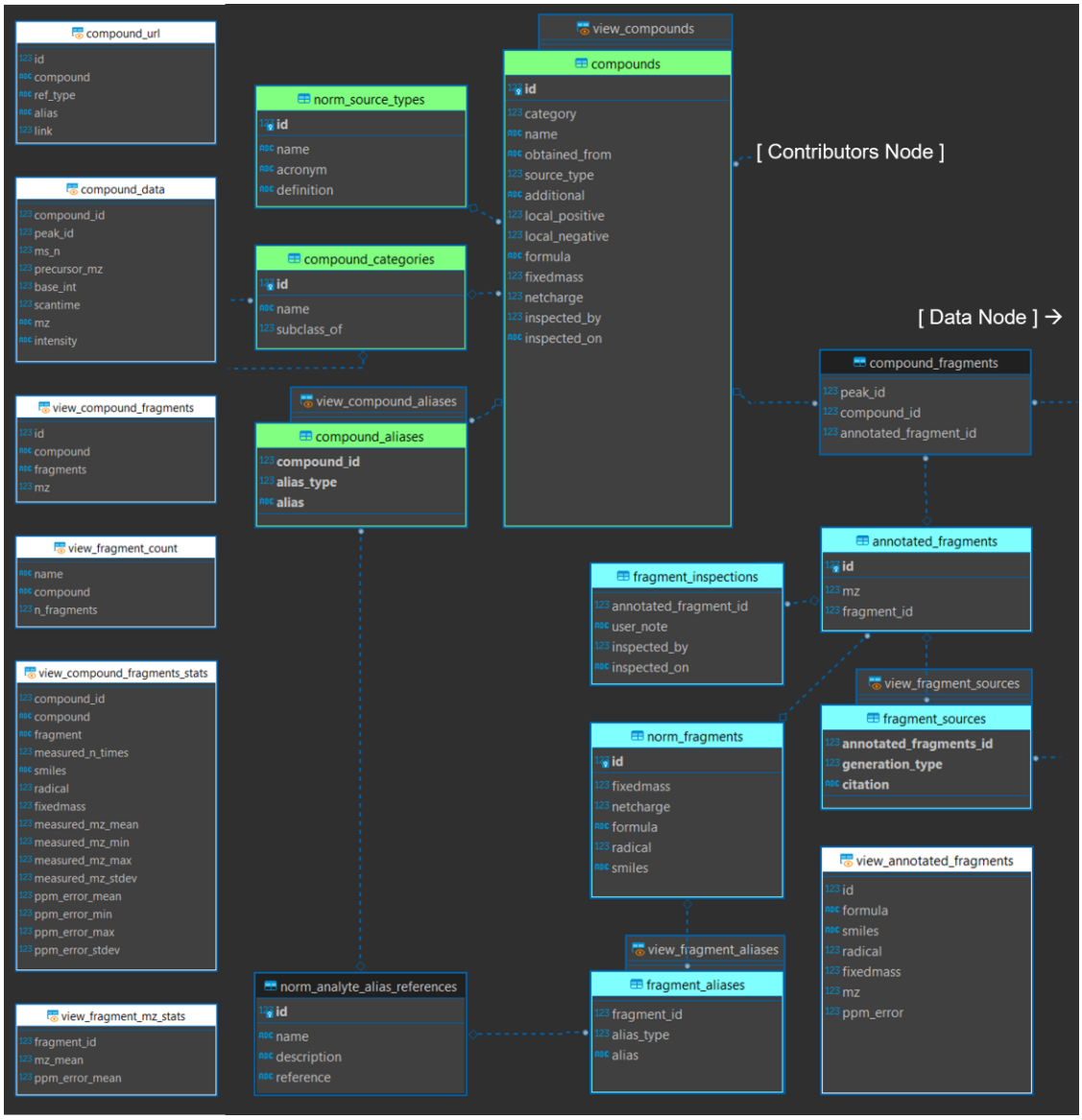


Figure 10. Analyte node of the entity relationship diagram (modified).

The Contributors Node

This node contains information relevant to identifying data contributors, similar to a "users" table. It is used primarily to provide contribution statistics and tie data to data producers in the samples and analytes node, both of which are connected to the peaks node. When the database is built, a "sys" username with the affiliation "system" is automatically added as a default user.

Entity Name	Description
Tables	
contributors	Contact information for individuals contributing data to this database
affiliations	Normalization table for contributor.affiliation
Views	
view_contributors	Readable version of the contributors table that can be expanded with counts of contributions from various places.

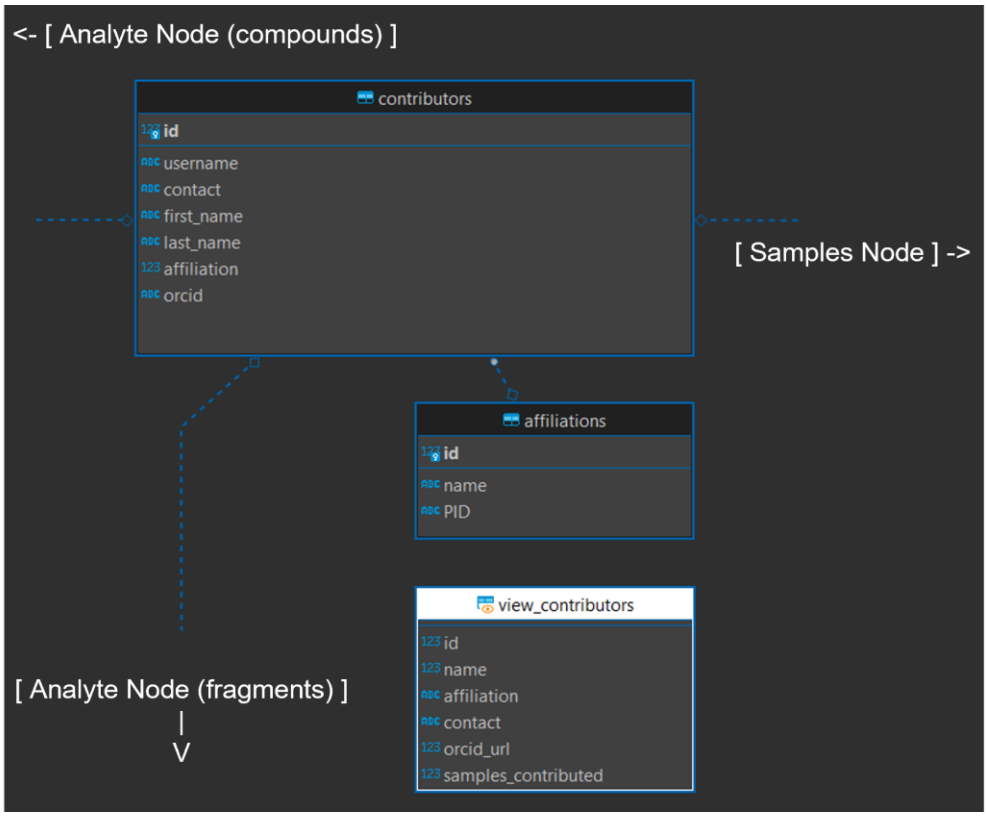


Figure 11. Contributors node of the entity relationship diagram (modified).

The Data Node

This node contains mass spectral data, metadata about samples and the software used to generate it, identification confidence, and quality control measures, as well as views to consume it directly. It is linked to the `analyte` node through the `compound_fragments` table and to the `contributors` and `methods` nodes through the `samples` table. This node contains two sub-nodes and is the main location of analytical data. One describes samples and one describes peaks generated from those samples Figure 12.

Entity Name	Description
Tables	
conversion_software_peaks_linkage	Linkage reference tying peaks with the conversion software settings used to generate them.
conversion_software_settings	Settings specific to the software package used to preprocess raw data.
ms_data	Mass spectral data derived from experiments on a compound-by-compound basis. Empirical isotopic pattern.
ms_spectra	Retained mass spectra associated with ms_data, unencoded from ms_data.measured_mz and .measured_intensity respectively.
norm_generation_type	Normalization table for fragment generation source type
norm_ion_states	Normalization table for the measured ion state as compared with the molecular ion.
norm_peak_confidence	Normalization levels for peak identification confidence
norm_sample_classes	Normalization table linking to samples to hold controlled vocabulary.
opt_ums_params	Table of optimal parameters for uncertainty mass spectra
peaks	Peaks (or features) identified within the results from a sample.
qc_data	Detailed quality control data as assessed by expert review (long format).
sample_aliases	Alternative names by which this sample may be identified e.g. laboratory or repository names, external reference IDs, URIs, etc.
samples	Samples from which analytical data are derived; physical artifacts that go into an analytical instrument. Deleting a contributor from the contributors table will also remove their data from the system.
Views	
peak_data	View raw peak data for a specific peak
peak_spectra	View archived and verified peak spectra for a specific peak
view_masserror	Get the mass error information for all peaks
view_peaks	View of “peaks” with text values displayed from

	normalization tables.
view_sample_narrative	Collapses the contents of view_samples and view_contributors into a single narrative string by ID
view_samples	[autogenerated by sqlite_auto_view()] View of “samples” normalized by “norm_sample_classes”, “norm_generation_type”, and “norm_carriers”.

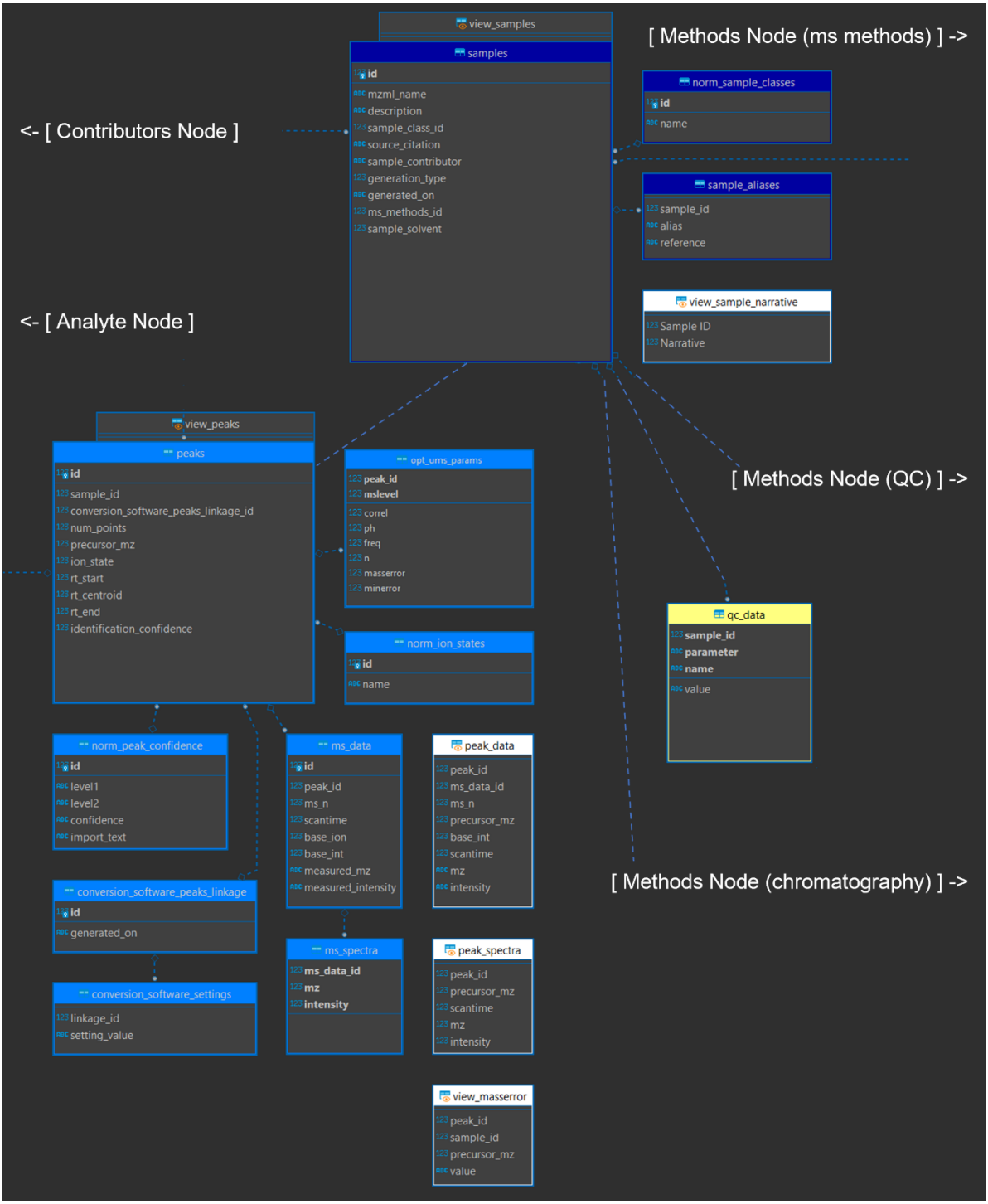


Figure 12. Data node of the entity relationship diagram (modified) showing the samples (top) and peaks (bottom) subnodes with node connections to the contributors node, the analyte node, and subnodes in the methods node.

The Logging Node

This node is included for automatic logging within the database itself (a future development opportunity), with tables to store and normalize logs and store a database version history. It is not used by default, but rather serves as a placeholder in case logging should be enabled via triggers if required by the data management and quality control systems for a given project. As it is not in use and has not been tested, details are not included but can be queried like any other tables.

The Methods Node

This node contains data describing experimental settings, both for the chromatographic separation and the mass spectrometer. It is the largest node, composed of four subnodes. The mass spectrometer ("mass spec") subnode contains information about the mass spectrometer settings used to collect data for an experiment and is closely related to the "descriptions" node which contains vendor descriptions for all instrumentation used in the experiment, allowing a single mass spectrometric method to describe multiple detectors and chromatographic separators. The quality control subnode describes the quality control procedures that were applied. Finally, the "mobile phase" subnode describe chromatographic conditions, allowing for multiple chromatographic components to be described, and multiple stages of mobile phase conditions.

Entity Name	Description
Tables	
additive_aliases	List of common aliases for each entry in norm_additives
carrier_additives	Mobile phase additives mixture for a given carrier mix collection
carrier_aliases	List of common aliases for each entry in TABLE norm_carriers
carrier_mix_collections	An intermediary identification table linking mobile_phases and carrier_mixes
carrier_mixes	Mobile phase carrier mixture for a given elution method
chromatography_descriptions	Full description of all chromatography types used for a given entry in ms_methods.
instrument_properties	Expandable properties describing performance properties of the mass spectrometer at the time the method was run.
mobile_phases	Description of mobile phases used during a chromatographic separation.
ms_descriptions	Full description of all mass spectrometer types used for a given entry in ms_methods.
ms_methods	Mass spectrometer method settings.

norm_additive_units	Normalization table for mobile phase additive units: controlled vocabulary
norm_additives	Normalization table for the carrier additives list: controlled vocabulary.
norm_carriers	Mobile phase carrier list: controlled vocabulary.
norm_ce_desc	Normalization table for collision energy description: controlled vocabulary.
norm_ce_units	Normalization table for collision energy units: controlled vocabulary.
norm_chromatography_types	Normalization table for chromatography types: controlled vocabulary.
norm_column_chemistries	Normalization table for chromatographic column type: controlled vocabulary.
norm_column_positions	Normalization table for chromatographic column position: controlled vocabulary
norm_duration_units	Normalization table for mobile phase duration units: controlled vocabulary
norm_flow_units	Normalization table for mobile phase flow rate units: controlled vocabulary
norm_fragmentation_types	Normalization table for fragmentation type: controlled vocabulary.
norm_ionization	Normalization table for mass spectrometer ionization source types: controlled vocabulary
norm_ms_types	Normalization table for mass spectrometer types: controlled vocabulary.
norm_polarity_types	Normalization table for ionization polarity: controlled vocabulary.
norm_qc_methods_name	Normalization table for quality control types: controlled vocabulary.
norm_qc_methods_reference	Normalization table for quality control reference types: controlled vocabulary.
norm_vendors	Normalization table holding commercial instrument vendor information: controlled vocabulary.
norm_voltage_units	Normalization table for ionization energy units: controlled vocabulary.
qc_methods	References to quality control (QC) methods used to vet experimental results
Views	
view_additive_aliases	`[autogenerated by sqlite_auto_view()]` View of

	“additive_aliases” normalized by “norm_additives”.
view_carrier_additives	View complete mobile phase used in a mixture
view_carrier_aliases	[autogenerated by sqlite_auto_view()] View of “carrier_aliases” normalized by “norm_carriers”.
view_carrier_mix	View complete mobile phase used in a mixture
view_carrier_mix_collection	Tabular view of carrier mix components by mixture ID
view_carrier_mixes	[autogenerated by sqlite_auto_view()] View of “carrier_mixes” normalized by “norm_carriers”.
view_chromatography_types	View all chromatography types in methods
view_column_chemistries	Convenience view to build view_method_as by providing a single character string for column chemistries used in this method
view_detectors	Convenience view to build view_method_as by providing a single character string for detectors used in this method
view_mass_analyzers	View all mass analyzers used in methods
view_method	View mass spectrometer information and method settings
view_method_narrative	Collapses the contents of view_method into a single narrative string by ID
view_mobile_phase_narrative	A print convenience view creating a narrative from the elution profile of each ms_methods_id, with one row for each profile stage.
view_mobile_phases	[autogenerated by sqlite_auto_view()] View of “mobile_phases” normalized by “norm_flow_units” and “norm_duration_units”.
view_ms_descriptions	[autogenerated by sqlite_auto_view()] View of “ms_descriptions” normalized by “norm_ms_types” and “norm_vendors”.
view_ms_methods	[autogenerated by sqlite_auto_view()] View of “ms_methods” normalized by “norm_ionization”, “norm_voltage_units”, “norm_polarity_types”, “norm_ce_units”, “norm_ce_desc”, “norm_fragmentation_types”, and “norm_ms_n_types”.
view_qc_methods	[autogenerated by sqlite_auto_view()] View of “qc_methods” normalized by “norm_qc_methods_name” and “norm_qc_methods_reference”.
view_separation_types	Convenience view to build view_method_as by providing a single character string for chromatography type

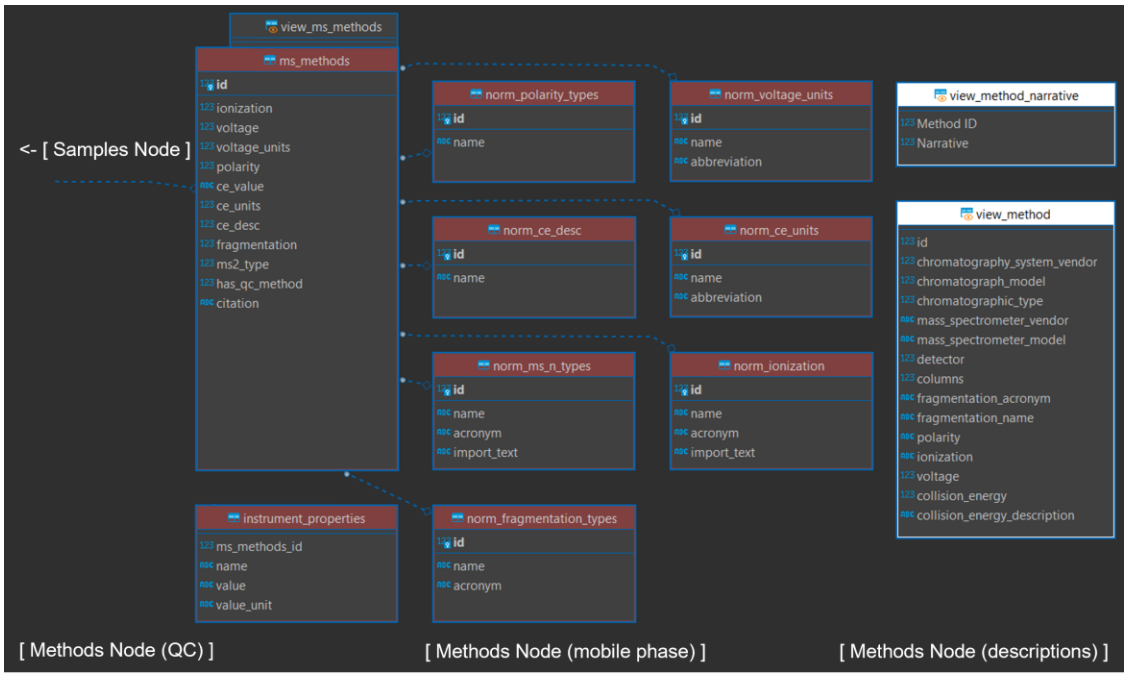


Figure 13a. Mass spectrometer subnode of the methods node (modified).

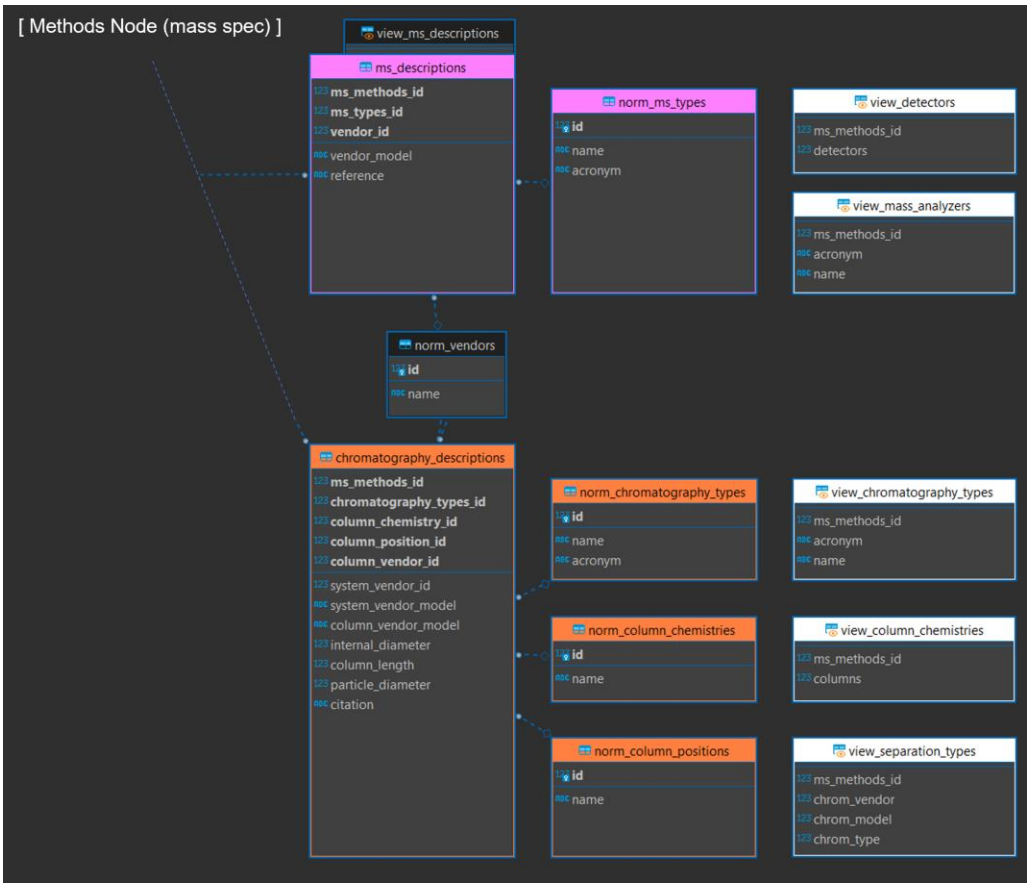


Figure 13b. Chromatographic and instrumental descriptions subnode of the methods node (modified).

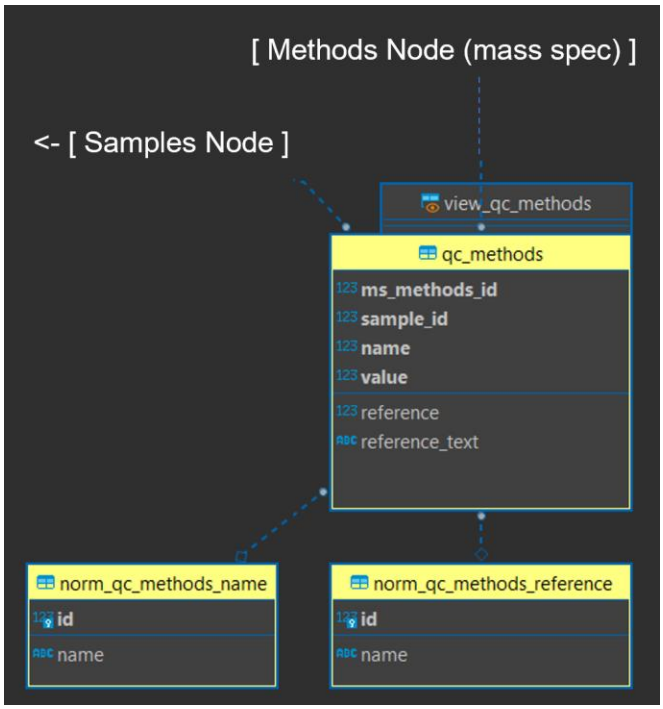


Figure 13c. Quality control subnode of the methods node (modified).

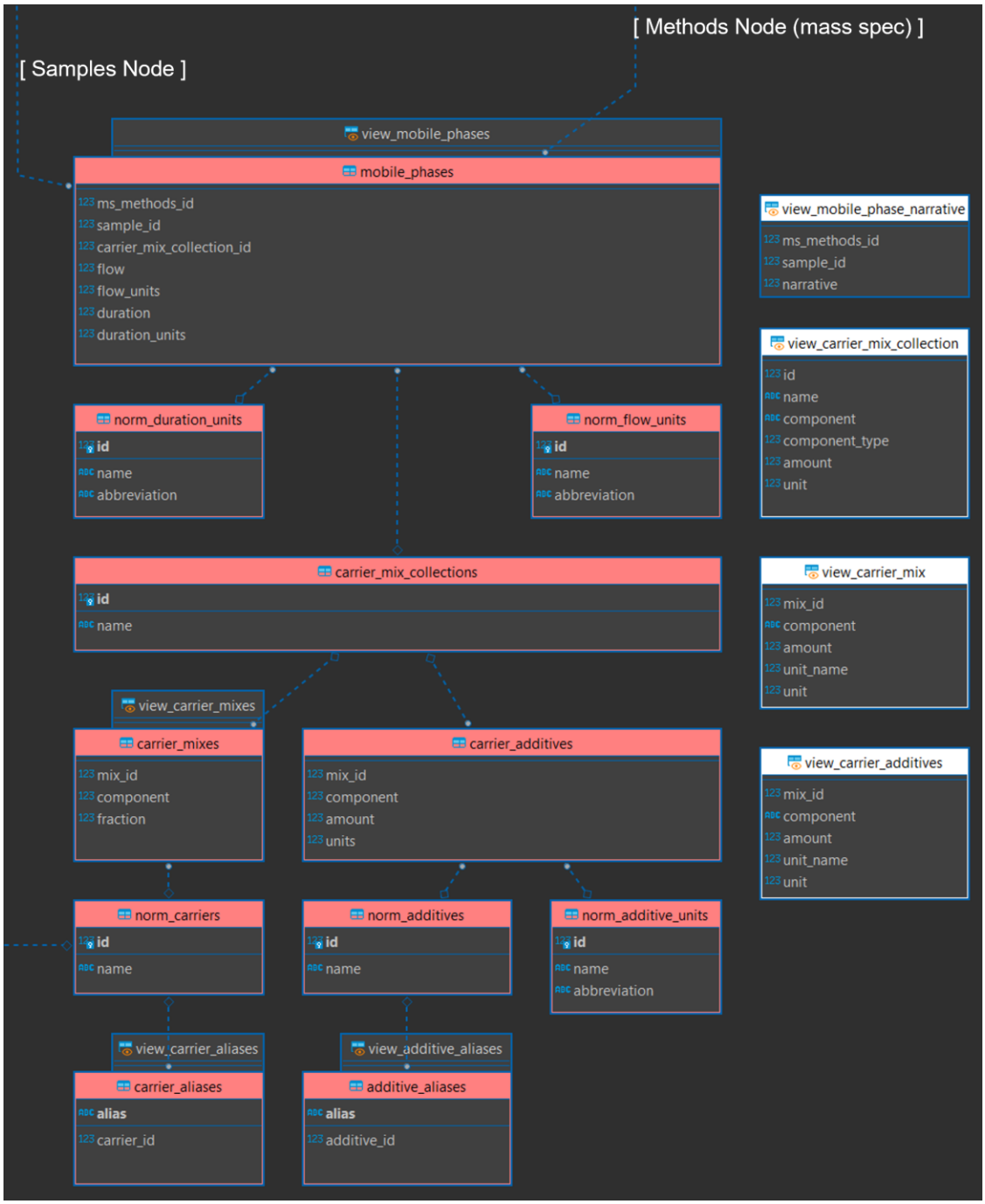


Figure 13d. Chromatographic mobile phase information subnode of the methods node (modified).

The Reference Node

This node contains universally applicable reference information for chemical metrology. Data for elements, their exact masses, and their natural isotopic abundances are automatically added as part of the database build process. It also includes a configuration table that, when built, will contain the current datetime stamp and an 8-character HEX installation code that should assist with any later combinations or referencing across database installations. This node does not directly connect to any others but serves only for computational convenience.

Entity Name	Description
Tables	
config	Installation code to facilitate widespread usage.
elements	Normalization list of periodic table elements 1-118.
isotopes	Elemental isotope abundance ratios for comparison and deconvolution.
Views	
view_exact_masses	Exact monoisotopic masses for elements at their highest abundance.
view_element_isotopes	A view of all elemental isotopes and their relative abundances joining reference tables “elements” and “isotopes”.

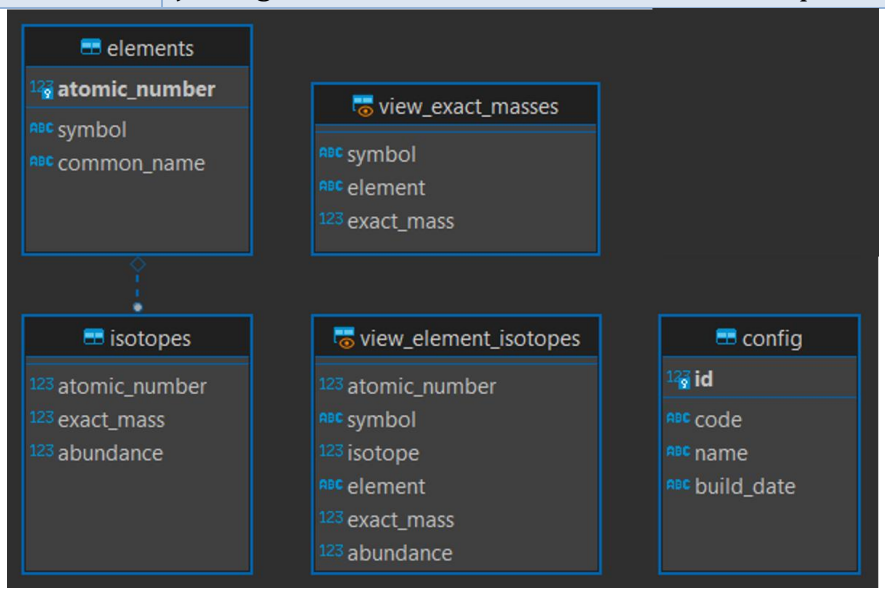


Figure 14. Reference node of the entity relationship diagram (modified)

Script Generated Views and Triggers

Creation of "denormalization views" Figure 9 and certain triggers can be accomplished in R with the `sqlite_auto_view` and `sqlite_auto_trigger` functions. These parse SQL definitions as demonstrated in Figure 4. For views, this results in a view of the table where foreign key indices are replaced by their human readable values where the linked normalization table is the simplest case of `id` and `value` columns.

Foreign key enforcement is provided for table columns linked to two-column normalization tables by automatically generated triggers. These triggers will examine the value of any supplied value to the column and replace it with the linked index. If a value does not exist, it will be added to the normalization table and the resulting index used. This is a crude data integrity measure for when the database is accessed without explicitly turning on foreign keys. Under normal circumstances the foreign key enforcement will take care of this issue but these are provided here as a backup.

To exclude automatic views and triggers, simply remove calls to those files in the `build.sql` file.

Populating Data at Build

Populating data can be accomplished automatically at build time from CSV or SQL files, allowing for rapid iteration and rebuild. Data population is defined by SQL scripts in the config directory. If the sqlite3 CLI is available (recommended) data are imported from CSV files using a series of `.import` commands, one for each table being populated. If the sqlite3 CLI is not available, instead use a fully qualified native SQL script such as the one provided at "`config/build_full.sql`" to build the schema and populate data in one step; this is less customizable but serves as a bridge for when CLI tools are not installed.

Compute Environments

Several environment resolution options are available in this project depending on which aspects are requested by the user. The default setting in the R/compliance.R file creates an R session that (1) connects to a database, (2) turns on the logging functionality, (3) turns on argument validation for certain R functions, (4) makes rdkit available to that session, and (5) spins up an API server in a background process can take a "considerable" amount of time (e.g. up to 3 minutes on slower systems; more typically this is around 90 seconds) but is still much slower than users may be accustomed to by loading packages. Environment values are set by configuration files, most of which are described in the Project Set Up section. To increase system compatibility, these are not set at the system level, but rather kept at the session level.

Users likely will not need every aspect each time. Starting an R session that can (re)build or connect directly to a database, or simply launching the API server, does not require every

aspect. Editing the environment files, primarily in config/env_glob.txt and config/env_R.R (Table 1; Table 2) will determine which aspects are made available in the session.

Users may choose whether to connect to the database (and load all the database communication support) at load time; this generally is not a time intensive operation, but connection validation and loading of additional packages does increase load time somewhat. If the connect option is selected and a database file with the defined name does not exist, it will be built according to the build settings in Project Set Up.

Users similarly may choose whether to launch the API server which adds additional dependencies. This can be done in an active session at any time after loading using `api_reload` with the background parameter determining if it should be launched in the current session or in a background process; the default is to launch it in a background process and return control to the session. The same function will allow the service to reload if any changes are made to API functionality in the "`inst/plumber/plumber.R`" file.

Using `rdkit` slows down the load time considerably as the R session must resolve, activate, and bind to a python environment. While it is recommended it is not required; set `INFORMATICS` to `FALSE` to turn this feature off.

For developers, two convenience functions are available to jump to specific files within the project. These require RStudio to use but will open identified files for viewing and editing.

- `open_env` will open an environment file. The name parameter must be one of the six defined environments (i.e. "R", "global", "logging", "rdkit", "shiny", or "plumber"); the default is "R" (e.g. `open_env("logging")` to edit the logger environment settings). Name options are hard coded to specific paths for this version;
- `open_proj_file` will open any file in the project, though its main use is for R script files. File identification is accomplished by regular expression matching on files in the project directory, and in the case of multiple matches it will instead return a list of those files. As always, functions can be viewed in the RStudio viewer with `View(fn)`.

To determine whether an environment has been established, boolean session variables are set when each file is sourced with the prefix `RENV_ESTABLISHED`, one for each aspect. Adding functionality during an active session will check for these and if they are required and do not exist, will automatically add necessary components to the current environment.

Shiny Applications

This user guide does not provide details on developing shiny applications. Shiny apps are enabled through the environment file at `inst/apps/env_shiny.R` and will load necessary packages (see References). These are automatically installed if not present the first time the application is launched on any given system.

Applications are located in the `inst/apps` directory and are self-contained in subdirectories by app name. The three that ship with the project are in the "three file" format of `global.R`, `ui.R`, and `server.R` and make use of the API for database communication; they will launch the API server in a background process if it is not already running. To add a live database connection to a new app, simply add the connection object to `global.R` for that app and develop as normal.

A skeleton application making use of project tooling is also provided. Simply copy the `inst/apps/app_template` directory under a new name and begin developing any needed shiny app as normal. Additional helper functions are defined in the `inst/apps/shiny_helpers.R` file such as the ability to easily append tooltips to any given shiny widget.

This section lists out the three applications that ship with the project and gives a brief description of them. Each are detailed in their own chapters.

Table Explorer

This Shiny application is simple example included with the project. It provides information about tables and views in an intuitive interface and details not only their contents but the entity definition and links to other entities in a human-readable format. Launch it after the compliance script has been run with `run_app("table_explorer")` or from the console or command line with `shiny::runApp("inst/apps/table_explorer")` which will solve the environment and launch the application. A developer mode is also available, allowing users to click the "Inspect" button to drop into an interactive R session to view (set `dev = TRUE` in `global.R` inside the app directory). The application includes only two screens, one ("Table Viewer") to preview data available there and to view structural information for an entity selected, and another to view a high-resolution picture of the entity relationship diagram that can be downloaded or examined in a separate browser tab with the right-click context menu option to open an image in a new tab.

This application served as proof-of-concept for environment resolution and API communication, and to provide DIMSpec users and developers a way to visualize connections between database entities.

The Table Explorer application is detailed in its own section.

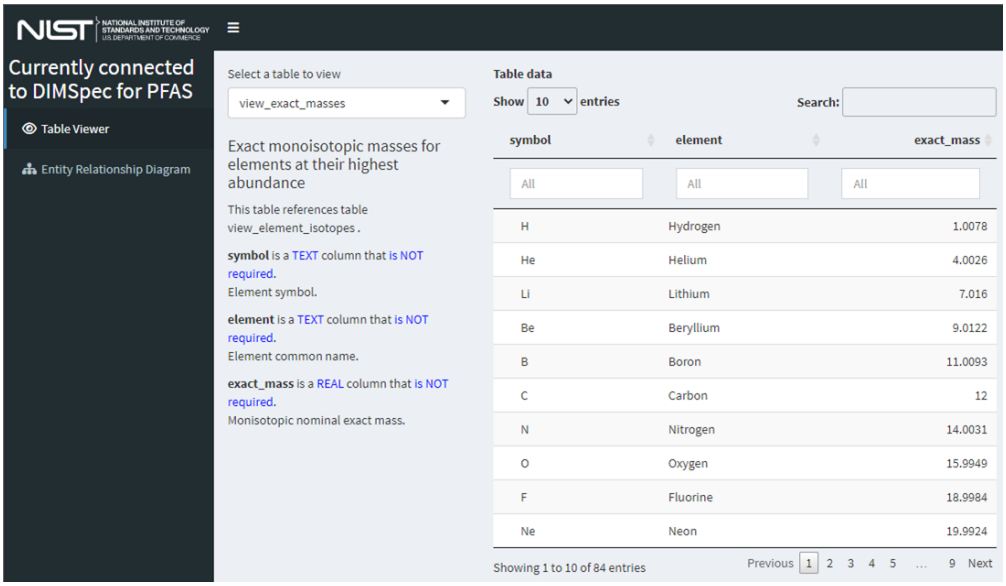


Figure 15. Screenshot of the "Table Explorer" shiny application showing database structural information available at a glance for the view_exact_masses database table.

Mass Spectral Match (MSMatch)

This Shiny application could easily be considered the entire reason behind the genesis of the DIMSpec project. It was built specifically to accelerate non-targeted analysis projects by searching experiment result data in mzML format for matches against a curated mass spectral library of compounds and annotated fragments. MSMatch is a web application built using the Shiny package in R and installs alongside DIMSpec and is one example of a tool that can built on top of the DIMSpec tool set. Launch it after the compliance script has been run with `run_app("spectral_match")` or from the console or command line with `shiny::runApp("inst/apps/spectral_match")` which will solve the environment and launch the application. Databases built and managed with DIMSpec are SQLite files used within a distributed R Project. Scripts for automated setup are included. For this initial release, DIMSpec is distributed with data populated for per- and polyfluorinated alkyl substances (PFAS); that effort has been primarily supported by the Department of Defense Strategic Environmental Research and Development Program (DOD-SERDP), project number ER20-1056.

The MSMatch application is detailed in its own section.

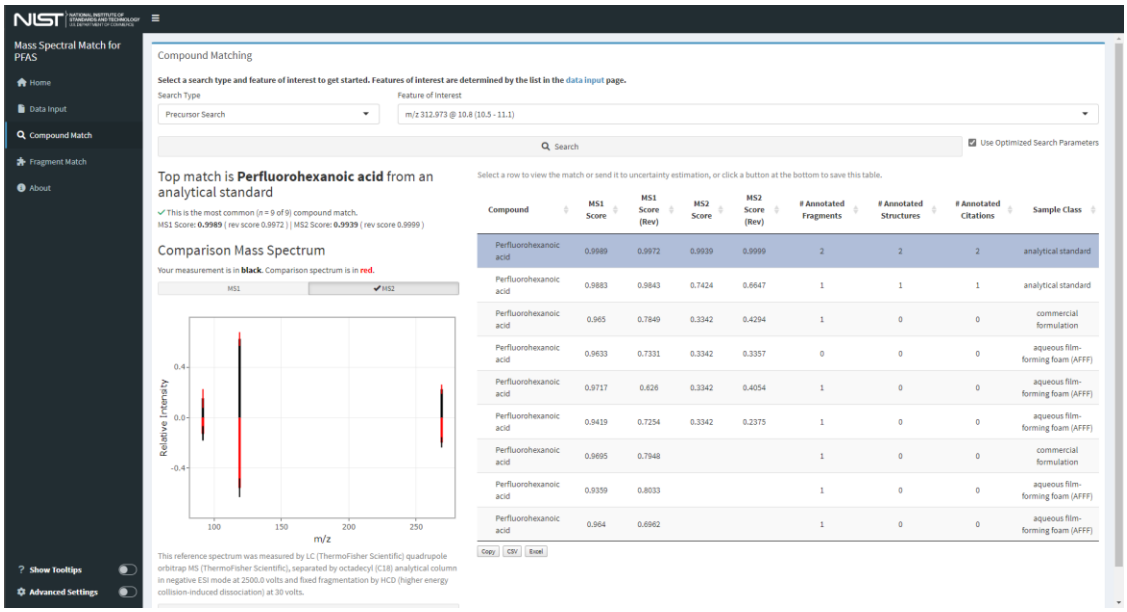


Figure 16. Screenshot of the "Mass Spectral Match" application with user data showing matches to known mass spectral signatures.

Mass Spectral Quality Control (MSQC)

This Shiny application serves as an automated application of quality control and assurance functions built to assess data contributions to DIMSpec. There is no "hard stop" based on these quality checks, but the results of the quality checks are included alongside data contributions when they are selected for addition to a DIMSpec database. Data in the mzML format and an associated JSON file produced by the NIST NTA-MRT tool are loaded into the application and then each may be selected to evaluate data quality. Specific conversion parameters must be used during conversion from the raw instrument output. For each peak within each raw file the following checks will be performed.

1. Is the reported precursor ion *m/z* value within the reported instrumental error of the calculated precursor ion *m/z* of the designated compound?
2. Does the MS1 isotopic pattern of the submitted data match the calculated isotopic pattern with a match score above an expected value?
3. Is the reported precursor ion *m/z* value present in the MS1 mass spectrum of the submitted data?
4. Are the reported annotated fragment ion *m/z* values present in the MS1 mass spectrum of the submitted data?
5. Are the reported annotated fragment ion *m/z* values value within the reported instrumental error of the fragment ion *m/z* of the designated fragment, calculated from the elemental formula?
6. If there is a SMILES structure provided for an annotated fragment, does the elemental formula of the SMILES structure match the elemental formula provided for the same annotated fragment?

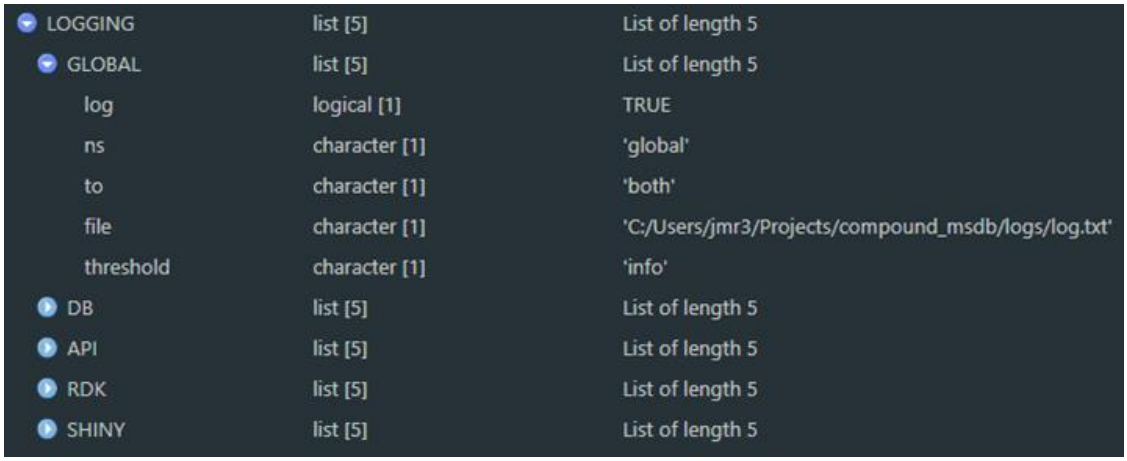
Finally, optimized settings for the uncertainty mass spectrum of the MS1 and MS2 data are calculated to facilitate searches for submitted compounds.

The MSQC application is detailed in its own section. Launch it after the compliance script has been run with `run_app("msqc")` or from the console or command line with `shiny::runApp("inst/apps/msqc")` which will solve the environment and launch the application.

Logger

Logging status and meaningful messages to both the console for interactive sessions and to disk allows for more efficient troubleshooting and debugging, as well as status reports on performance. Logging is enabled by default in the project and implemented by custom functions on top of the logger package. That package supports namespaced logging at various levels and prints nicely to the console if provided with a formatter function. This user guide does not go into detail about the logger package, but documentation is readily available online.

Logging parameters are set when the compliance script is sourced (see Project Set Up). Settings are stored in a session list `LOGGING` with each element named for a namespace. Turn off logging by setting `LOGGING_ON = FALSE` in the set up process. At any time during a session, you may change the logging settings and run the `update_logger_settings` function to change the way to logging functions are working. Setting the `$log` property to `FALSE` will cease printing and recording of log messages for that namespace. Options for the `$to` property include "file", "console", and "both" and, if this is either "file" or "both", logs will be written to the path in the `$file` property. The threshold must be one of the valid options in logger, which is a ranked vector (i.e. one of "`trace`" < "`debug`" < "`info`" < "`success`" < "`warn`" < "`error`" < "`fatal`"); the `$threshold` property represents the minimum level at which to record logs. If, for example, the threshold is set to "info" then "trace" and "debug" messages will be ignored.



Users may then issue logging statements directly through `logger` using the `$ns` property namespace. The primary manner of issuing logging messages in the project however is the `log_it` function which works without `logger` to display messages in the console and includes an abstraction and validation layer that provides the benefits of argument validation, ignoring certain function calls, inclusion of the function from which `log_it` was called, and setting up new namespaces by cloning the settings from an existing one. If `logger` is available, logs will print nicely to the console and be saved to disk if set to do so. Logs written to disk can at any time be read back into the session either printed to the console (`read_log`) or to a data frame (`log_as_dataframe`) for examination.

Logs are not tracked in the git repository for the project as each installation should have isolated logs. The `logger` package supports rollover past a certain size. The log directory (or any directory, though that is not recommended except in the case of a directory of temporary files) can be flushed at any time with the included function `flush_dir(directory = "logs", pattern = ".txt")` where `directory` is the name of an available directory and `pattern` is a regular expression used to match files to remove. Matching files will be removed by default but can be archived with the current date suffixed to the file name by setting `archive = TRUE`.

Plumber

Communication with the underlying database is performed by default using a background process through the plumber package, which produces a RESTful API and offloads processing to a separate R process. It is recommended to communicate with the API using the `api_endpoint` function which takes as arguments mainly the endpoint path and any named criteria necessary to execute the endpoint, however, the plumber service can be used with any API query method (e.g. httpuv, httr, curl, etc.). The `api_endpoint` function accepts any server address and path (if the path includes the address it will ignore the `server_addr` argument), accepts any additional arguments needed to use the endpoint with the ellipsis (including advanced parameters for other APIs), will return the constructed call rather than executing it with `execute = FALSE`, will return values to the session (the default) or in a browser window with `open_in_browser = TRUE`, and can return results in a variety of formats. Plumber APIs communicate in JSON strings. Set the `return_type` and `return_format` appropriately for what you expect back from the server; the default is a vector.

The environment for the API is defined at `inst/plumber/env_plumb.R`; package requirements are resolved as part of the environment. When launched as a background process the name of the resulting object is drawn from the `env_plumb.R` file, by default `plumber_service`. During launch the server will add several R scripts from defined files in the project directory; more can be added by modifying the `r_scripts` variable. Logging is turned on by default for the API as no interactive session is available. Finally, `rdkit` integration is enabled and a connection is established with the project database as defined elsewhere. Any of these settings can be customized for use.

Certain options which may also be changed are set at the project level in `"config/env_R.R"`, which is necessary to share values and keep sessions connected appropriately. Those options include the `PLUMBER_VERSION` (if you are iterating for development, testing, or deployment), `PLUMBER_HOST`, `PLUMBER_PORT`, and `PLUMBER_FILE`. These should generally not be changed, but certain network configurations may require alterations to the host and port. It is recommended in that case to set options (e.g. `getOption("plumber.host")` and `getOption("plumber.port")`) in the project or user `.Renviron` file or in the session prior to running the compliance script to set other values; the compliance script will honor those settings. The default listening port is

`http://127.0.0.1:8080`; Swagger documentation will be available at `http://127.0.0.1:8080/__docs__` until the server stops.

To restart the API server from an R session, use the `api_reload` function. This will close and relaunch the service in a background process and return control to your session. Call it with `background = FALSE` to launch it directly within the current session, which will automatically launch a browser pointing to the Swagger documentation. By default `api_reload` pulls its parameters from the current compute environment. To launch a second API server with different endpoints under a different name, use `api_reload` with different parameters, e.g.:

```
api_reload(
  pr = "api_service2",
  plumber_file = "new_plumber.R",
  on_host = "127.0.0.1",
  on_port = 8085
)
```

The background process is launched using `callr::r_bg` and the health of resulting object can be checked with the `api_service2$is_alive()` property for the example above. With the server running, call `api_open_doc` with the API URL (e.g. `http://127.0.0.1:8085` for the example above) to launch a browser with the documentation.

Plumber endpoints are decorated R functions available to a routing server managing transactions for request and response. Endpoint specifications that ship with the project are defined in the `inst/plumber/plumber.R` file. Plumber uses the OpenAPI specification and provides interactive Swagger documentation once the server is active. Any R function can be turned into a plumber endpoint, or an endpoint can be defined that uses an existing R function if that function is available in the environment. The latter case is the one used most often in this project; a modified version of the function is created and then redirected as needed. This allows flexibility to be able to handle any preprocessing necessary (e.g. unpacking JSON or argument verification) before calling the underlying function while keeping that function available for interactive use. Several endpoints are provided in the project and are described in the following paragraphs.

Two filters are defined that will ensure a database connection is alive and log incoming requests prior to forwarding to their destination path.

Three health endpoints are defined:

- **"_ping"** checks to make sure the server is ready to respond to requests and returns "Ok" if so; if the server is not yet ready (e.g. it is still resolving its environment) calling with api_endpoint("_ping") will try a number of times (20 is the default) before it times out;
- **"db_active"** and **"rdkit_active"** return Boolean values for whether the API can communicate with the database and **rdkit,** respectively.

Four inspection endpoints are defined to assist with debugging:

- "**version**" returns the value defined by `PLUMER_VERSION` to ensure the running version meets expectations;
- **"support_info"** endpoint returns a nested list of system and project properties drawn from the environment files and server information; it is roughly equivalent to the `support_info` project function intended to facilitate support tasks during deployment;
- **"exists"** returns a Boolean value for whether a function exists in the server environment;
- **"formals"** returns the names of formal arguments for a function by name as defined in the server environment and is roughly equivalent to the base R function **"formals"** when called as a list.

Both **"exists"** and **"formals"** must be called with the function name appended to the path (e.g. **"formals/build_db"**).

The remaining twelve endpoints perform database queries or actions:

- **"compound_data"** returns mass spectral data for a compound by its internal ID number;
- **"list_tables"** and **"list_views"** lists tables and views, respectively, in the database;
- **"method_narrative"** returns the mass spectroscopic method narrative for a peak, sample, or method by its database primary key id;
- **"molecular_model/file"** and **"molecular_model/png"** use `rdkit` to generate and return either (**"/file"**) a file path to a molecular ball-and-stick plot of a compound or fragment in portable-network-graphics (png) format or (**"/png"**) the graphic itself. If notation is provided, it must match the `notation_type` provided;
- **"peak_data"** returns mass spectral data for a peak by its internal ID number;
- **"sample_narrative"** returns the plain text narrative for a sample by its database primary key id;
- **"search_compound"** uses `search_precursor` or `search_all` to find matching compounds for a processed mass spectrum object in JSON notation. This does no preprocessing of the `search_ms` item and only executes the defined search on the database. The serialized version of the object created from `create_search_ms` is much smaller than that of serializing the entire mzML object;
- **"search_fragments"** uses `get_compound_fragments` and additional processing to find matching fragments from the database for a list of fragment mass-to-charge ratios;
- **"table_search"** is equivalent to `build_db_action` for SELECT queries and is the most flexible way to query the database.

More information about these endpoints is available using the Swagger documentation, which includes live testing for endpoints. New endpoints can be defined easily, and the server can be quickly relaunched at any time with `api_reload`.

Python

Python is an open-source general programming language with similar aspects to R. Packages are controlled by environment management programs, the most common of which is the "conda" package manager available through installing either Anaconda or Miniconda. Integrating R and Python requires the `reticulate` R package which allows for miniconda to be installed independently. If no installation can be identified, the compliance script of this project will install miniconda through R. Python environments are sets of packages tied to a python distribution and must be established prior to use; when using `reticulate` to integrate R and Python that environment must include the `r-reticulate` python library. Environment resolution should in most cases be left to the compliance script but is described below for both completeness and to support and inform non-standard installations of python and package management solutions. There are several settings in the "`inst/rdkit/env_py.R`" environment script that may be changed to comply with systems using advanced or non-standard python package management approaches.

Setting	Type	Description
PYENV_NAME	String	The name of the python environment (`conda` preferred) to use. It must contain `rdkit` and `r-reticulate` and can be solved using the provided .yml file; defaults to `"nist_hrms_db"` but can be easily customized for any installation. This is only a name reference to activate and, if necessary, build a python environment (be default a `conda` environment).
PYENV_REF	String	The name of the R session object which will be tied with `rdkit` functions; defaults to `"rdk"`.
USE_PY_VER	Numeric	The version of Python to install into the `PYENV_NAME` environment; defaults to 3.9.
INSTALL_FROM	String	For stability, this should be `"local"` in most cases which will build from the file located at `INSTALL_FROM_FILE`.
INSTALL_FROM_FILE	String Path	The file path to a local environment.yml file to be used to build the python environment using functions provided in this project.
PYENV_LIBRARIES	String Vector	A fall-back list of python packages required if `INSTALL_FROM_FILE` fails on your system, assists with manual creation using `reticulate`.
PYENV_MODULES	String	Module names that must be present and available in

	Vector	the environment; defaults to `"rdkit"` which is the only required library for this project.
PYENV_CHANNELS	String Vector	Channels from which to install python libraries; defaults to `"conda-forge"` for consistency. Other channels can be added for customization and non-conda package distributions.
CONDA_PATH	String Path	Under most circumstances, this should always be left as the default `"auto"` though advanced set ups may have other requirements and, in that case, (e.g. multiple installations) this should be a file path to a `conda` executable that will be used for this project.

Several project functions exist to create, activate, and manage python environments through reticulate. These are housed in the "`inst/rdkit/py_setup.R`" file and documentation is available with `fn_help(fn)` where `fn` is the name of the function either quoted or unquoted. The most user friendly of these is: `rdkit_active` which wraps the other function calls in a flexible arrangement pulling from environment settings and has the following arguments inheriting the above settings:

rdkit_ref	defaults to PYENV_REF (see above)
rdkit_name	defaults to PYENV_NAME (see above)
log_ns	the logging namespace to use, defaults to "rdk"
make_if_not	must be a TRUE/FALSE value indicating whether an rdkit environment as defined in the table above should be installed if it is not already available (defaults to FALSE)

Chemometric operations are performed by the `rdkit` package (as built into this python environment) which is a wrapper for the underlying C library and is used here primarily by the plumber server with the `reticulate` package. This ROA does not include a tutorial for rdkit.

The first time the project is used, sourcing the `"R/compliance.R"` script should automatically set up the compute environment. The python module is published to the conda registry as `"rdkit"` in the `"conda-forge"` channel. To minimize package conflicts, it is recommended that it be installed with packages only from the `"conda-forge"` channel and forced to python 3.9 for the current version of this project. With conda installed and available in your PATH environment, there are two options to manually create the required environment. (If miniconda is installed through R this will not be available.) In a terminal prompt open at the project directory, this can be created from the provided environment file with:

```
conda create -n nist_hrms_db -f inst/rdkit/environment.yml
```

or created directly from any directory with:

```
conda create -n nist_hrms_db -c conda-forge python=3.8 reticulate=1.24
rdkit=2021.09.4
```

Again, functions in the project should take care of the set up as part of the environment resolution but may fail on certain systems. Note that the environment name "nist_hrms_db" is only a recommended environment name and could be anything, but in either case must match that provided in the configuration file at inst/rdkit/env_py.R as variable PYENV_NAME. There is no need to activate this environment, but testing it is considered good practice. Once the compliance script is run and rdkit is available (typically as the R object rdk), all rdkit functionality can be used (e.g. create a mol file from the SMILES string for hexane using rdkit$Chem$MolFromSMILES("CCCCCC").

Importing Data

For now, importing data is only supported at the command line using JSON mzML files generated by the NTA MRT tool. For other uses, import routines will need to be developed to translate source data into the database schema. Several import file examples are provided in the example directory. Import requirements are defined in the file "config/NIST_import_requirements.json" and each file is screened against this list. During the import process, files are parsed with the import map defined in "config/map_NTA_MRT.csv"

The easiest way to demonstrate the import routines (which are largely described in Importing Data) from within an R session (the following requires that packages magrittr and stringr be loaded) is to create a list of files to import (e.g.)

```
f_dir     <- file.path("example")
# here "example" could also be a network path
f_names   <- list.files(f_dir, ".JSON$", full.names = TRUE)
to_import <- lapply(
  f_names,
  function(x) fromJSON(read_file(x))
) %>%
  setNames(
    str_remove_all(
      f_names,
      sprintf("(example%s|.JSON$)", .Platform$file.sep)
    )
  )
```

verify these files meet import requirements and expectations:

```
obj_check <- verify_import_requirements(to_import)
cat(
  sprintf("Required data are %spresent.\n",
          ifelse(all(obj_check$has_all_required), "", "not ")),
  sprintf("Full detail data are %spresent.\n",
          ifelse(all(obj_check$has_full_detail), "", "not ")),
```

```
  sprintf("Extra data are %spresent.\n",
          ifelse(any(obj_check$has_extra), "", "not "))
)
```

and examine the resulting `obj_check` session object, a data frame of verification checks which can be filtered and searched as normal Figure 17. The missing_requirements, missing_detail, and extra_cols columns in the resulting data frame are list columns detailing any number of verification failures and which elements were missing or not in the import mapping definition.

Figure 17. Screenshot of the results from importing a directory of data files in JSON format and verifying they meet import expectations.

This verification check is run automatically as part of the `full_import` pipeline, with the following parameterized options to determine behavior based on verification of individual components of the import object:

- *exclude_missing_required*
  - defaults to `FALSE`
  - Rather than the import pipeline failing, set this to TRUE to ignore and exclude (and get a list of failures) files that do not meet import requirements.
- *stop_if_missing_required*
  - defaults to `TRUE`
  - Stops the import process if files do not include all requirements. This is forced to `TRUE` when `exclude_missing_required` is `FALSE` to ensure integrity.
- *include_if_missing_recommended*
  - defaults to `FALSE`
  - By default, files missing recommended information are skipped from the import process, under the assumption that data may be missing. Set to `TRUE` to include these.
- *stop_if_missing_recommended*
  - defaults to `TRUE`

    - Stops the import process if files do not include all recommended data.
- *ignore_extra*
    - defaults to `TRUE`
    - Rather than the import pipeline failing, set this to `TRUE` to ignore and exclude (and get a list of failures) data in files that are not expected given defined import requirements.
- *ignore_insert_conflicts*
    - defaults to `TRUE`
    - Rather than the import pipeline failing, set this to `TRUE` to ignore and exclude (and get a list of failures) insert conflicts during database write operations.

This user guide does not completely detail all of the parameters for the `full_import` function as most are schema specific and passed to underlying action functions. See the full function documentation if interested. The import pipeline resolves each of the SQL Nodes in order and makes liberal use of console feedback during the process when it cannot automatically resolve:

1. **Contributors** are resolved under the expectation that this may be often repeated for a given set of import files; a temporary data frame is created that maps provided contributor identifiers to known contributors and, where resolution fails, prompts the user to either create a new contributor record or map that value to an existing contributor which is then maintained for the remainder of the import process.
2. **Methods** are resolved by examining the mass spectrometric method properties (i.e. ionization type, voltage, polarity, collision energy, etc.) and matching against known methods, allowing for deep linking of data that were generated using the same mass spectrometry method; additional instrument properties are also recorded in a linked table allowing for an unlimited number of records to be attached.
3. **Descriptions** are resolved for the models and properties for any number of mass spectrometer and chromatographic separation system configurations used to generate data; conceptually, this could result in conflicts if the exact same method settings are used across multiple systems, but collisions are highly unlikely.
4. **Sample** properties and the mzML file used to generate data are resolved and can have attached any number of aliases for a single sample including external identifiers and links to those data sources; similarly to the methods node, sample properties are first matched against known samples to enable deep linking and for data from samples to be reanalyzed while maintaining linkages to previous data.
5. **Chromatography** condition descriptions for sample and method are resolve to an open-ended collection of carrier information including flow rates and carrier mixes with individual components and additives.
6. **Quality control** descriptions resolve to QC methods and open-ended result data used for a given sample and method with references to defined QC.
7. **Peak** information about features of interest measured in a sample is resolved including retention time, precursor ions, the number of points measured across a peak, and any identification confidence associated with it; conversion software

settings (e.g. from processing with msconvert) are also linked as are optimal uncertainty mass spectra parameters to assist with spectral matching; measured m/z and intensity values are housed here, typically in a format describing all m/z and intensity values in a single row; for highly trusted spectra needing more advanced or rapid search capabilities, these data can also be unpacked into a long form table of the same data;

8. **Compound** reference information is resolved for known analytes including where data originated, known or calculated exact mass and molecular other properties, their common and machine-readable aliases, and optionally chemical categorization; when data from an identified compound are submitted, compounds are matched by all known aliases prior to being added into the table and resolved to the known internal identifier.
9. **Fragments** annotated in the data submission either with or without a known structure are resolved for known fragments and is matched to known aliases to firmly identify fragment identity and limit size creep and are normalized (fragments with a known structure are kept separate from those with only elemental formula notations); inspection records are also available for future annotation notes of known fragments.
10. **Peak**, **fragments**, and **compound** linkages are resolved in the `compound_fragments` table providing flexible linkages for peaks representing only known compounds (no fragment annotation), peaks with only annotated fragment information (no compound identification), and compounds with known annotated fragments but no peak data yet in the system, allowing for any combination of known information to be populated *a priori* mass spectrometric data are collected.

Each node is resolved using a specific function that is called during the `full_import` pipeline. Normalization resolution is accomplished using `resolve_normalization_value` which checks known values in a normalization table and, when encountering an unknown entry, prompts the user to either map it to a known value or to add a record to that table. This is leveraged heavily throughout and is the main reason the import process (for now) is only available for interactive console sessions. This is what allows the pipeline to maintain identifiers throughout the import process while populating individual data tables.

Future Development

Both the R/Shiny and python code are fully extensible and community stakeholder feedback will be important for the future success of this project. Future development may include deployment of and to a Shiny server to host shiny applications, extending the python code to analyze data of various formats from different instruments, and adding analysis features and functionality (e.g. high resolution plot generation and download, or supporting the full workflow from instrument through import and to report generation) requested by stakeholders.

Conclusions

The Database Infrastructure for Mass Spectrometry (DIMSpec) project provides a new way to rapidly create and iterate databases to combine data from mass spectrometry experiments and associated metadata about the analytical methods and samples used to generate those data. It is hoped that this tool set makes a standard schema easily accessible to both internal and external stakeholders and that it can potentially be reused for any projects within the Chemical Sciences Division that generate mass spectrometric data; by structuring such data in a common schema the Division will be one step closer to a shared data environment allowing for federated data management while supporting customized add on tools for data processing and display. For one example of how this structure can be used, we highlight the "Mass Spectral Match (MSMatch) for Non-Targeted Analysis" application, which demonstrates how new data in mzML format can be used to interrogate a version of this database populated with high resolution non-targeted mass spectral data of per- and polyfluorinated alkyl substances to match both compounds and annotated fragments and accelerate NTA projects.

Shiny Web Applications

Table Explorer

To facilitate visual exploration of the DIMSpec database schema, a web application was written in Shiny. It served as proof-of-concept for the database/API/shiny approach and was used as the basic skeleton of the template app that ships with the project.

Table Explorer is a simple entity viewer for the attached database. Combining the comment decorations in DIMSpec and reading of entity definitions from the database (see Inspecting Database Properties) allows for R to expose a wealth of information about the underlying schema and quickly change which entity is being viewed. See Shiny Applications for details of how to launch this app, but the easiest method is after the `compliance.R` script has been executed, use `start_app("table_explorer")` to launch it in your preferred browser.

Table Viewer

There is only one page for interactive content, named "Table Viewer" (Figure 1). A navigation bar on the left controls the current page being viewed; collapse the bar using the "hamburger" icon (≡) at the top next to the NIST logo. Click the drop down box (Figure 2 - left) to change the database table or view being displayed. This will update the definition narrative immediately below the selection box (Figure 2 - right) and display the contents of that table (Figure 3).

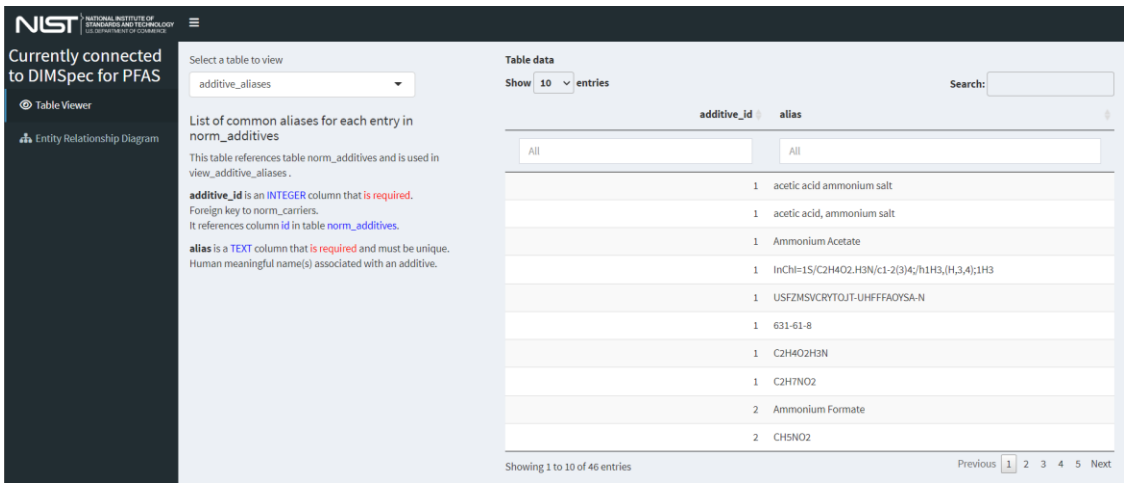


Figure 1. The Table Explorer main page.

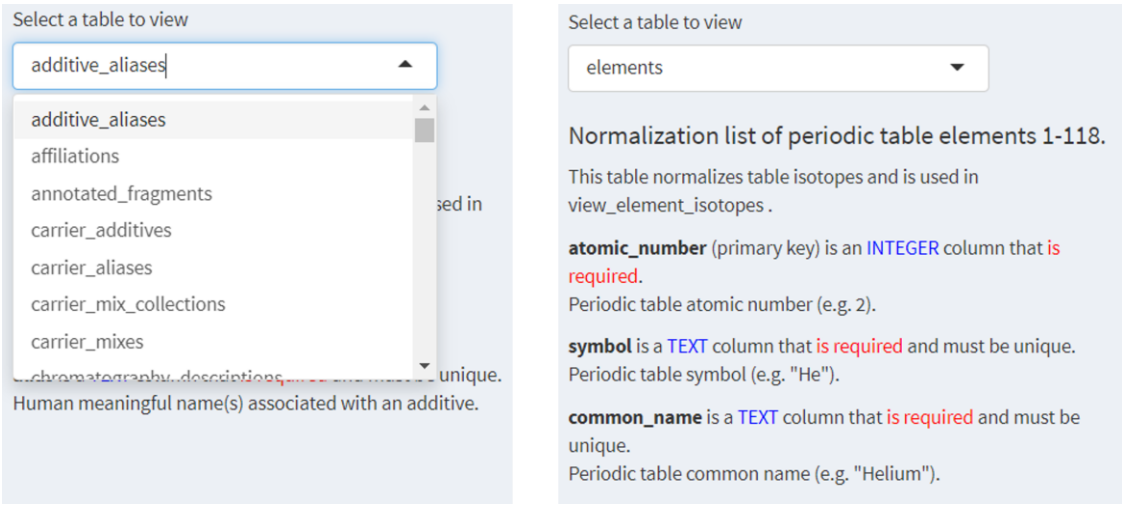


Figure 2. Choose a database entity (left) for information about its definition (right).

Table data

Show 10 entries Search:

atomic_number	symbol	common_name
All	All	All
1	H	Hydrogen
2	He	Helium
3	Li	Lithium
4	Be	Beryllium
5	B	Boron
6	C	Carbon
7	N	Nitrogen
8	O	Oxygen
9	F	Fluorine
10	Ne	Neon

Showing 1 to 10 of 118 entries Previous 1 2 3 4 5 … 12 Next

Figure 3. Data held in the selected entity.

Entity Relationship Diagram

A full graphical representation of the entity relationship diagram is also provided. A full resolution version of this graphic is available from inside the app by right-clicking it and opening it in a new tab.

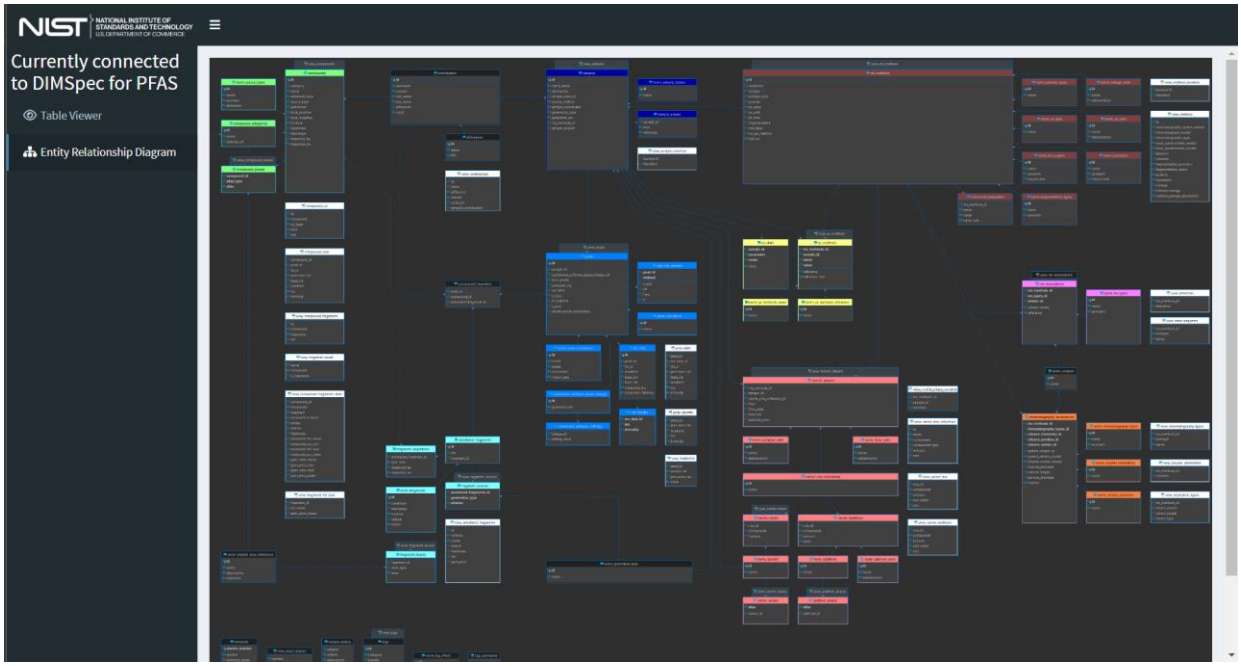


Figure 4. Entity Relationship Diagram

Mass Spectral Quality Control (MSQC)

Introduction

One goal of the Data Infrastructure for Mass Spectrometry (DIMSpec) project is to provide a database that can be easily retasked to support individual projects within the Chemical Sciences Division to manage data coherently and accelerate analyte identification, screening, and annotation processes for non-targeted analysis projects. Databases built and managed with DIMSpec are SQLite files used within a distributed R Project. A DIMSpec mass spectral database incorporates empirical mass spectral data from analytical standards and complex mixtures with relevant analytical method metadata and mass spectral annotation. Algorithms have been developed in R to validate the quality of new experimental data.

For ease of use, the Mass Spectral Quality Control (MSQC) application was developed that incorporates the R functions into a R/Shiny application. Scripts for automated setup are included. For this initial release, DIMSpec is distributed with data populated for per- and polyfluorinated alkyl substances (PFAS); that effort has been primarily supported by the Department of Defense Strategic Environmental Research and Development Program (DOD-SERDP), project number ER20-1056.

Set Up Instructions

The MSQC application installs alongside DIMSpec. If there is continued (or expanded) interest, the project could be turned into an R package installable directly from GitHub with additional development or this tool can be deployed to a shiny server for use by those connected to the NIST network without the need for launching or maintaining it locally. For now, this application is distributed for demonstration and evaluation with an implementation of NIST DIMSpec containing high resolution accurate mass spectrometry data for per- and polyfluorinated alkyl substances (PFAS). The R project can be opened in RStudio[iv] which may be downloaded and installed free of charge if not already installed. Initial set up does require an internet connection to install dependencies; on a system which does not contain any software components this can take a considerable amount of time.

Refer to the System Requirements section for installation details.

Input File Format Requirements

To use MSQC, raw data files produced by a mass spectrometer must be converted into mzML format (Deutsch 2010) using Proteowizard's msConvert software (Adusumilli, Ravali and Mallick, Parag 2017). There are specific parameters that must be used during conversion.

```
Filter:           Threshold peak filter
Threshold type:   absolute
Orientation:      most intense
Value:            1
Filter:           Peak picking
Algorithm:        vendor
MS levels:        1-2
```

A more detailed user guide for converting these files is provided as a vignette in the project directory.

Non-Targeted Analysis Method Reporting Tool

A macro-enabled Microsoft Excel workbook, called the Non-Targeted Analysis Method Reporting Tool (NTA-MRT), is used for the systematic collection of sample, method, and compound information related to chemicals identified in a sample. The most up-to-date version of NTA-MRT is publicly available at GitHub.

The instructions for completing the NTA-MRT are contained within the tool itself. In order to use the MSQC application, a `sample.JSON` file must be generated using the "Export to JSON file output" button on the first tab of the NTA-MRT.

The file name entered in the NTA-MRT under the Sample tab must exactly match (case-sensitive) the paired mzML file name to be used for the MSQC.

Launching MSQC

Launch this tool similarly to other "shiny"-based tools as part of DIMSpec. In brief, this can be done from a terminal or the R console, though the preferred method is to use RStudio (RStudio Team 2020). The following commands are typical given an existing installation of R or RStudio and should always be run from the project directory. The shiny package (Chang et al. 2021) and other necessary R packages will be installed if it not already available by running the script at `R/compliance.R`, but shiny is the only package required to start the application. When first run, it may take a moment to install necessary dependencies and launch the application programming interface (API) server.

Terminal[v]

R.exe "shiny::runApp('inst/apps/msqc')"

R console

```
shiny::runApp('inst/apps/msqc')
```

RStudio

Open the ".Rproj" project file in RStudio, navigate to the "inst/apps/spectral_match" directory, open one of the "global.R," "server.R," or "ui.R" files, and click the "Run App" button. Files open in an RStudio project will remain open by default when RStudio is closed, allowing users to quickly relaunch by simply loading the project. For best performance, ensure "Run External" is selected from the menu "carrot" on the right to launch the application in your system's default web browser. This application has been tested on Chrome, Edge, and Firefox.

Alternatively, once the compliance script has been executed MSMatch can be launched using `start_app("msqc")`.

Once launched the API server will remain active until stopped, allowing users to freely launch, close, and relaunch any shiny apps dependent upon it much more quickly. The application is fluid and will dynamically resize to fit the dimensions of a browser window. By default, the server does not stop when the browser is closed. This means that, once started, it is available by navigating a web browser back to the URL where it launched until the server is shut down.

If anything is needed from the user, interactive feedback will occur in the console from which it was launched. Install any packages required if prompted by the application. Once the package environment requirements have been satisfied and the server has spun up, which may take a moment, the tool will launch and display the "About" screen (Figure 1) either in the RStudio viewer or the browser. The navigation panel on the left will control which page is currently being viewed; click an entry to navigate to that page.

While this version of DIMSpec includes analytical data for PFAS, MSQC can be tailored for any given database name. See Technical Details>Application Settings for customization options.

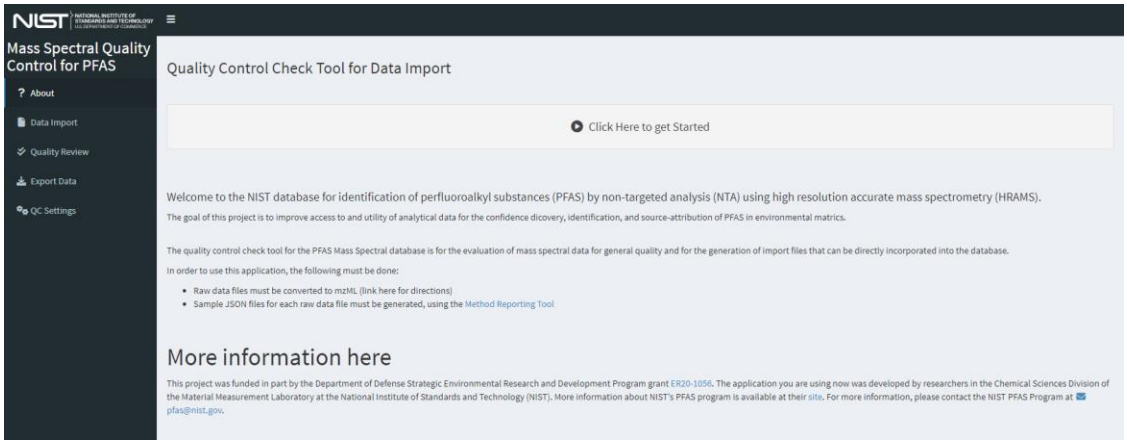


Figure 1. The home page for the MSQC web application contains basic information about the application and can be tailored easiliy for each use case.

Using MSQC

Every effort has been made to make using the MSQC application as intuitive to use as possible. Generally the user interface will adjust to the current needs and highlight the next step. Some steps (e.g. 2 and 3) can be used interchangeably.

Step 0 - Modify quality control settings (optional)

Prior to file processing, verify or set parameters to be used for quality control analysis can be modified. On the left menu, select "QC Settings" to navigate to the settings dashboard shown in Figure 2. The default settings are recommended, but can be modified if needed.

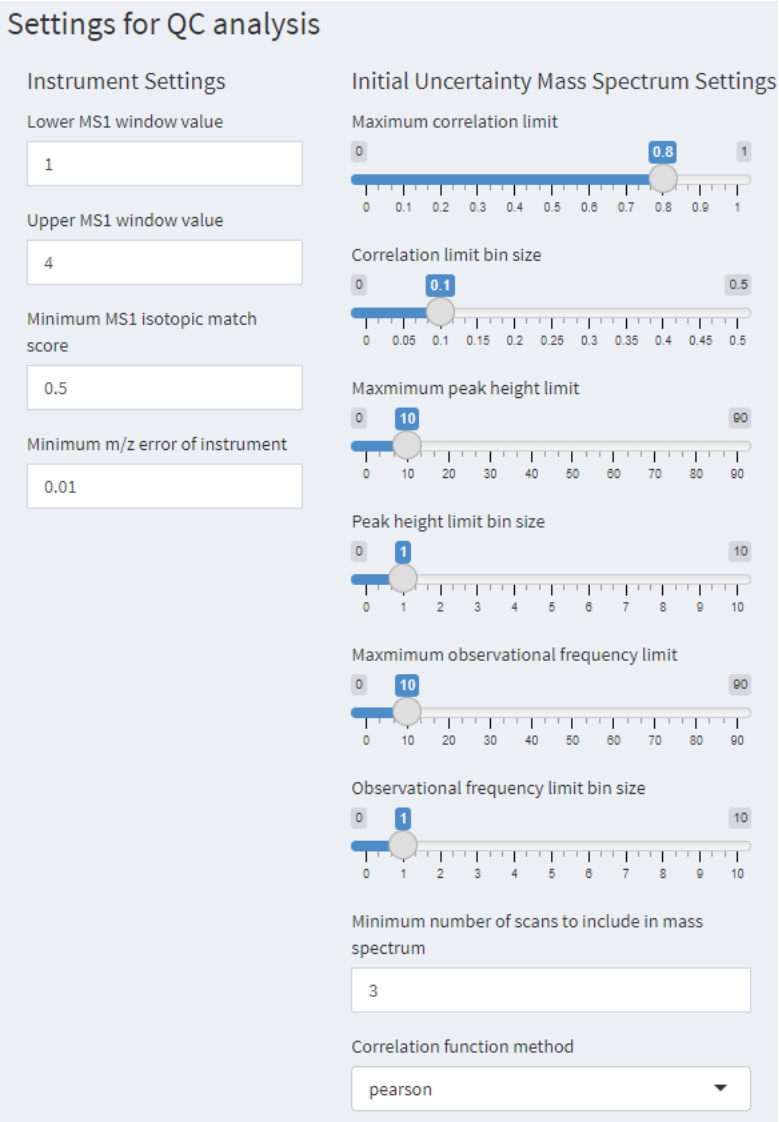


Figure 2. The QC Settings window that has settings that can be adjusted for quality control analysis, the uncertainty mass spectrum settings are described in Place 2021.

Step 1 - Import mzML and Sample JSON Files

Upload paired mzML and sample JSON files produced by the NTA-MRT macros (conventionally named "filename_mzml_sample.JSON"). First, select "Data Import" on the left menu or the "Click Here to get Started" button from the "About" page to bring up the Data Import page Figure 3. Multiple files may be uploaded; the limit of number and size of files is dependent on system memory By default the maximum size of a single file is 250 MB, but this limit can be increased by changing the setting of `file_MB_limit` in the `global.R` file.

First, load the mzML files of interest using the top "Load" button or by dragging the data file(s) to the input widget labeled *1) Load raw mzML file(s)*.

Second, load the paired sample JSON files using the second "Load" button or by dragging the JSON file(s) to the input widget labeled *2) Load Sample JSON file(s)*. The sample JSON files do not need to be selected in the same order as the mzML files and will be matched by name when processed.

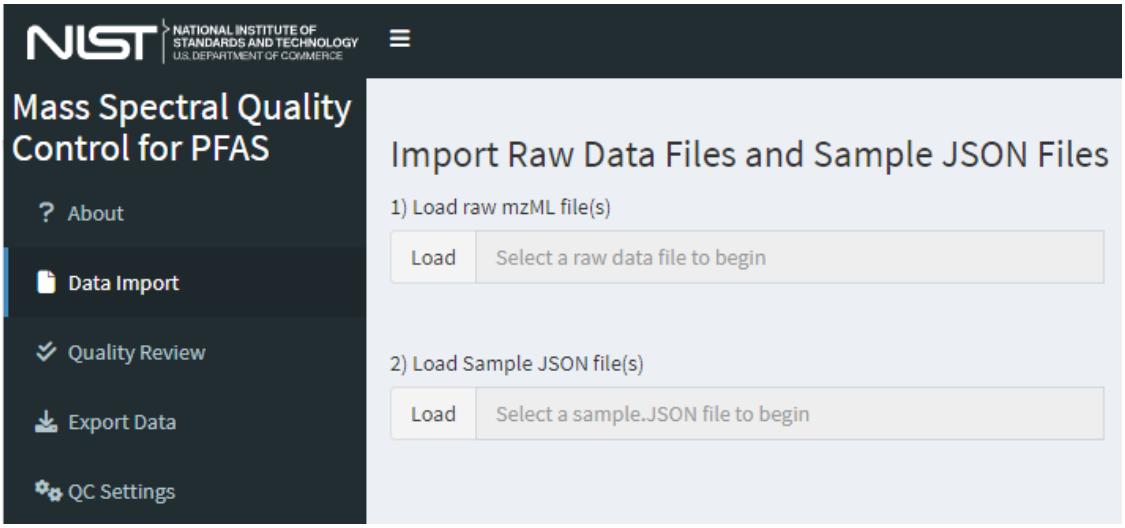


Figure 3. The data import page of MSQC where data files, experiment parameters, and features of interest are identified. Workflow guidance options become available once data are provided.

Once mzML and Sample JSON files are uploaded, the application will automatically check to see if there are valid pairs of mzML and Sample JSON files. A successful upload will give a screen similar to Figure 4. If multiple files are loaded, only files with verified matches will be included in the table and available for further processing.

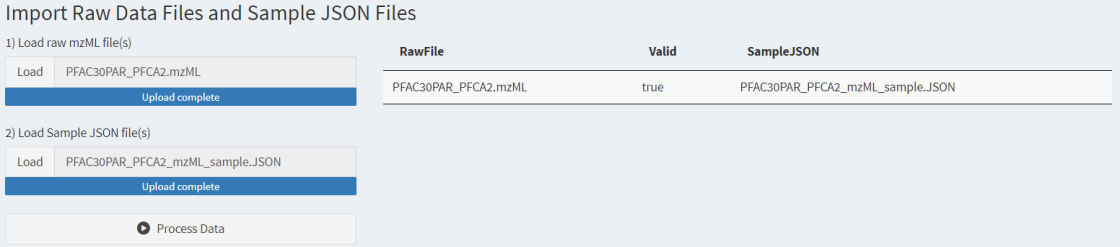


Figure 4. The data import page after data files have been loaded.

Step 2 - Process Data

After files have been loaded and matched, a button will appear on the "Data Import" window that is labeled "Process Data". Click the "Process Data" button and it will sequentially process each mzML file. This can take up to 5 minutes per raw file depending on the number of compounds per file, so a large number of files may take a long time to process. Progress indicators are provided. Once complete, the text under "QC Data Import Status" will read: "Data processing complete."

If a raw file does not have a valid Sample JSON, the files can still be processed, but the invalid rows will be excluded.

Step 3 - Review QC Results

Once processed, the QC results can be reviewed by selecing "Quality Review" in the left menu or clicking the "Quality Review" button that appears below the "Process Data" button.

The top table, which is the only visible table when starting a new review, shows the raw files that have been processed and the respective quality control check results ("PassCheck"). If all compounds in the raw file passed all QC checks, the PassCheck result will be true. If any compound in the raw file failed any of the QC checks, the PassCheck result will be false.

To review the compounds within a single raw file, select the row of the raw file you want to review in the table labeled *1) Click a row to select an mzML file*. This will display a second table of all compounds within the selected raw file. The PassCheck result for each compound is displayed in this table. If all QC checks for each compound in the raw file passed all QC checks, the PassCheck result will be true for that compound. If a compound in the raw file failed any of the QC checks, the PassCheck result will be false for that compound.

To review the individual QC checks (described in the Technical Details>Quality Control Evaluation section), select a row for the peak to review in the table labeled *2) Click a row to see metrics for that peak*. This will display boxes to the right containing all QC checks for that compound. Expand a specific QC check by clicking on the box header to display the results of the QC check as a table. An example view of the quality control review page is

shown in Figure 5. A note below the two tables on the left will indicate whether any QC checks failed.

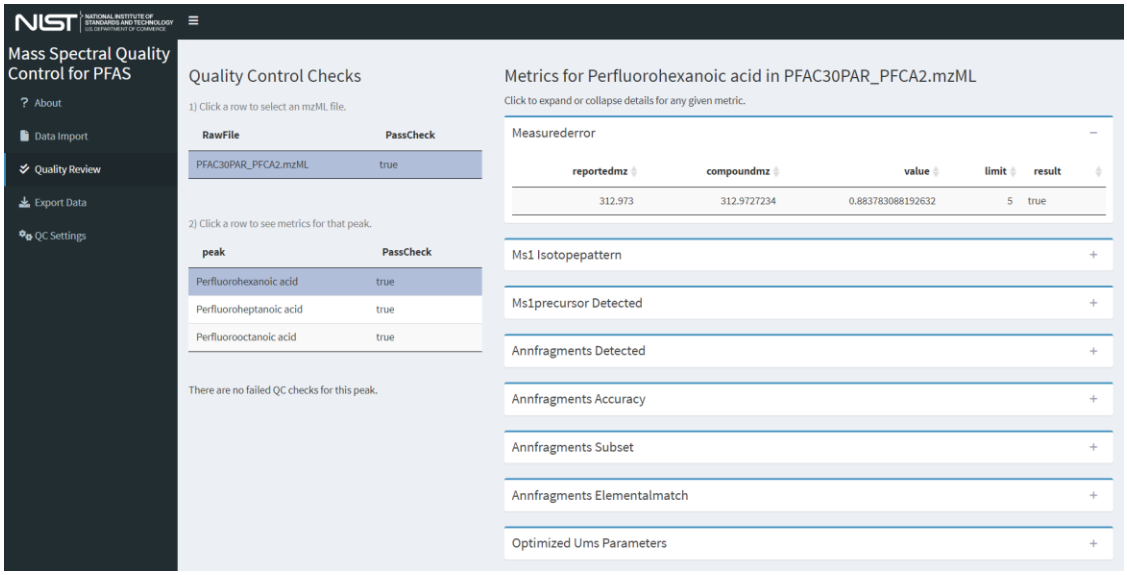


Figure 5. Screenshot of the quality review page with both an mzML file and a peak object selected.

Step 4 - Export Data

Once data are processed, all data can be exported (regardless of quality review status) by selecting "Export Data" in the left menu. Additional options may be added in the future to refine the export process such as selecting only peaks and files that pass all defined quality checks.

Clicking the button labeled "Export all data", will write the peak JSON files and download them in a single .zip file. This file can be unzipped and the peak JSONs can be directly incorporated into the DIMSpec database using the import routine described in the Importing Data section.

Step 5 - Closing Down

When finished using the application, typing the escape key at the R console is the simplest way to stop the server and exit the application. If using RStudio there is a "stop sign" button at the top right of the console pane that will also stop it. When finished completely with the project, users also need to shut down the API server.

- Loading the entire project from the compliance script (i.e. MSMatch was launched using `start_app("spectral_match")`) provides additional actions and includes a live database connection with the ability to read data into tables and preserve them for further analysis. Use the function `close_up_shop()` with the argument

`back_up_connected_tbls` set to `TRUE` to preserve these, or the default FALSE to simply close all connections including the API server).

- If launching the app directly and using the default settings there will be a session object named `plumber_service` connected to that server. To stop it, use the `api_stop` function from the console or stop the service directly using `plumber_service\$kill()`; it will also generally stop when the calling R process closes (e.g. when RStudio is closed), but it is highly recommended to stop it manually to prevent hanging connections.
- After closing all connections, a hanging connection may be indicated by the presence of "-shm" and "-wal" files in the project directory. Flushing these hanging connections is not required but is recommended.
  - If launching MSMatch with the compliance script, run [`close_up_shop()`] again.
  - Otherwise flush those connections by directly connecting and disconnecting with the DBI package:

```
con <- DBI::dbConnect(
            RSQLite()::SQLite,
            "nist_pfas_nta_dev.sqlite")
DBI::dbDisconnect(con)
rm(con)
```

Feature requests, suggestions, and bug reports are most conveniently submitted as issues via GitLab but may also be submitted by contacting the authors of this ROA. New functionality suggestions are encouraged as the project tooling develops. Likewise, if the functionality demonstrated here is of interest to projects outside of PFAS, this is only one example implementation of the underlying technology stack (i.e. DIMSpec); contact the authors to see if your mass spectrometry data would be amenable to that framework as other implementation suggestions are encouraged and a larger goal of the project to cohesively manage mass spectrometry data for non-targeted analysis within the Chemical Sciences Division.

This concludes the User Guide for the Mass Spectral Quality Control (MSQC) web application. The following section contains technical details about the implementation and user customization of this digital assistant.

Technical Details

Implementation and environment details for the MSQC application largely follow those for DIMSpec. See the sections on System Requirements, Environment Resolution, Shiny Applications, Plumber for the API implementation which is required for this application, and Python Integration for chemometrics support.

Technical details in this section will describe only the MSQC application found in the `inst/apps/msqc` directory and, unless otherwise noted, all files referred to hereafter refer to that directory.

JSON Schema

There are two types of JavaScript Object Notation (JSON) files used for the data import and quality control process:

1. *Sample JSON*: these files (conventionally labeled "[mzmlfilename]_mzml_sample .JSON") are exported from the Non-Targeted Analysis – Method Reporting Tool (NTA-MRT) and contain sample, method, and compound information related to a paired mzML file. A visual representation of the Sample JSON schema is shown in Appendix A.

2. *Peak JSON*: these files are exported from the MSQC application (conventionally labeled "[mzmlfilename]_mzml_cmpd[compoundreferencenumber].JSON") and contain sample, method, compound information with mass spectral data related to the specified compound. The Peak JSON schema is shown in Appendix B.

Quality Control Evaluation

Upon running the quality control data processing there are seven individual checks that are performed. Within the MSQC app, the different checks are represented by parameter names. The QC checks with their named parameters are as follows. Most settings for these checks may be changed in the QC Settings page.

- *measurederror*: is the reported precursor ion *m/z* value within the reported instrumental error of the calculated precursor ion *m/z* of the designated compound? This calculation uses the instrument relative mass error contained in Sample JSON file, the absolute minimum mass error (default: **0.01 Da** - see the setting labeled *Minimum m/z error of instrument*), and the monoisotopic mass of the designated compound in the DIMSpec database.

- *ms1_isotopepattern*: does the MS1 isotopic pattern of the submitted data match the calculated isotopic pattern with a match score above an expected value? This calculation uses the Minimum MS1 isotopic match score (default: **0.5** - see the setting labeled *Minimum MS1 isotopic match score*), the lower MS1 window value (default: **1** - see the setting labeled *Lower MS1 window value*), and the upper MS1 window value (default: **4** - see the setting labeled *Upper MS1 window value*).

- *ms1precursor_detected*: is the reported precursor ion *m/z* value present in the MS1 mass spectrum of the submitted data? This calculation uses the instrument relative mass error contained in Sample JSON file and the absolute minimum mass error (default: **0.01 Da** - see the setting labeled *Minimum m/z error of instrument*).

- *annfragments_detected*: are the reported annotated fragment ion *m/z* values present in the MS1 mass spectrum of the submitted data? This calculation uses the instrument relative mass error and annotated fragment ion list contained in Sample JSON file and the absolute minimum mass error (default: **0.01 Da** - see the setting labeled *Minimum m/z error of instrument*).

- *annfragment_accuracy*: are the reported annotated fragment ion *m/z* values value within the reported instrumental error of the fragment ion *m/z* of the designated fragment, calculated from the elemental formula? This calculation uses the instrument relative mass error and fragment elemental formulas contained in the Sample JSON file and the absolute minimum mass error (default: **0.01 Da** - see the setting labeled *Minimum m/z error of instrument*).

- *annfragments_subset*: are the reported annotated fragment elemental formulas a subset of the elemental formula of the designated compound? For example, is the fragment "C3F7" a subset of the designated compound elemental formula "C8F15O2H"? In this example, the result would be true. This calculation uses the elemental formula contained in the Sample JSON file and the elemental formula of the designated compound in the DIMSpec database.

- *annfragment_elementalmatch*: if there is a SMILES structure provided for an annotated fragment, does the elemental formula of the SMILES structure match the elemental formula provided for the same annotated fragment? This calculation uses the SMILES structure and the elemental formula contained in the Sample JSON file.

- *optimized_ums_parameters*: this is not a quality check, but occurs during the same data processing step. Optimized settings for the uncertainty mass spectrum of the MS1 and MS2 data is calculated using the function `optimal_ums` for import into the DIMSpec database.

At any time necessary, these settings may be changed on the QC Settings page and QC checks run again by clicking "Process Data".

Application Settings

Many global application settings are customizable by modifying the `global.R` file. Changes to those listed here should not cause issues, but other settings in this file may result in instability. Anywhere a `TRUE` or `FALSE` value is indicated should only be `TRUE` or `FALSE`. The most germane user settings include:

Future Development

Both the R/Shiny and python code bases are fully extensible for future functionality needs, as is the underlying database infrastructure for custom tables and views. Future development may include deployment of and to a Shiny server to serve this as a hosted

web application, extending the python code to analyze data of various formats from different instruments, and adding analysis features and functionality (e.g. high resolution plot generation and download or supporting the full workflow from instrument through import and to report generation) requested by stakeholders.

This concludes the technical details section for the Mass Spectral Quality Control (MSQC) application.

Conclusions

The Mass Spectral Quality Control application provides a new way to make NTA tools developed at NIST on top of the Database Infrastructure for Mass Spectrometry accessible to both internal and external stakeholders. It is the first demonstration of tools that can be built on top of databases conforming to the DIMSpec project which can be repurposed for any class of chemicals or project of interest.

Appendices

Example Sample JSON Schema

The javascript object notation (JSON) schema describing samples is minimal for flexibility and extensibility as it is produced by visual basic for applications (VBA) scripts in the Non Targeted Analysis Method Reporting Tool (NTA-MRT), and is not fully defined in a machine readability sense allowing for automatic schema verification.

This definition is intended only to facilitate transfer and assessment of data through the NTA-MRT into the DIMSpec schema, and sufficient for that purpose.

Any number of schema harmonization efforts could connect DIMSpec with larger schema development efforts within the community to increase machine readability and transferability in line with the FAIR principles. This is an area where the DIMSpec project can be improved and schema mapping efforts can serve to connect data with larger projects outside of this project. NIST welcomes collaborative efforts to harmonize schema with larger efforts; reach out with an email to the PFAS program at NIST to start a collaboration.

```
{
  "sample": {
    "name",
    "description",
    "sample_class",
    "data_generator",
    "source"
  },
  "chromatography": {
    "ctype",
    "cvendor",
```

```
    "cmodel",
    "ssolvent",
    "mp1solvent",
    "mp1add",
    "m2solvent",
    "mp2add",
    "mp3solvent",
    "mp3add",
    "mp4solvent",
    "mp4add",
    "gcolvendor",
    "gcolname",
    "gcolchemistry",
    "gcolid",
    "gcollen",
    "gcoldp",
    "colvendor",
    "colname",
    "colchemistry",
    "colid",
    "collen",
    "coldp",
    "source"
  },
  "massspectrometry": {
    "msvendor",
    "msmodel",
    "ionization",
    "polarity",
    "voltage",
    "vunits",
    "massanalyzer1",
    "massanalyzer2",
    "fragmode",
    "ce_value",
    "ce_desc",
    "ce_units",
    "ms2exp",
    "isowidth",
    "msaccuracy",
    "ms1resolution",
    "ms2resolution",
    "source"
  },
  "qcmethod": [
    {
      "name",
      "value",
      "source":
    }
  ],
  "peaks": {
    "peak": {
      "count":,
      "name",
```

```
      "identifier",
      "ionstate",
      "mz",
      "rt",
      "peak_starttime",
      "peak_endtime",
      "confidence"
    }
  },
  "annotation": {
  "compound": {
      "name",
      "fragment": {
          "fragment_mz",
          "fragment_formula",
          "fragment_SMILES",
          "fragment_radical",
          "fragment_citation"
          }
      }
  }
}
```

Example Peak JSON Schema Extension

The javascript object notation (JSON) schema describing peak data and quality control metrics is a minimal extension of the sample schema and is produced by the MSQC tool and associated R functions used as part of the QC evaluation process. It is not fully defined in a machine readability sense allowing for automatic schema verification.

MSQC uses this extension to split the provided sample schema by peak, maintaining the sample metadata, attach the "msdata" element containing anlytical results, and attach resulting QC data. This results in one file per peak for import into the DIMSpec schema, and is sufficient for that purpose.

Any number of future schema harmonization efforts could connect DIMSpec with larger schema development efforts within the community to increase machine reading and transferability in line with the FAIR principles. This is an area where the DIMSpec project can be improved and schema mapping efforts can serve to connect data with larger projects outside of this project. NIST welcomes collaborative efforts to harmonize schema with larger efforts; reach out with an email to the PFAS program at NIST to start a collaboration.

```
{
  ...,
  "msdata": [
    {
      "scantime",
      "ms_n",
      "baseion",
      "base_int",
```

```
      "measured_mz",
      "measured_intensity"
    }
  ],
  "qc": [
    [
      {
        "parameter",
        ...,
        "value",
        "limit",
        "result"
      }
    ]
  ]
}
```

Mass Spectral Match (MSMatch)

Introduction

One goal of the Data Infrastructure for Mass Spectrometry (DIMSpec) project is to provide a database that can be easily built to support individual projects within the Chemical Sciences Division to manage data coherently and accelerate analyte identification, screening, and annotation processes for non-targeted analysis projects. Toward that end, the Mass Spectral Match for Non-Targeted Analysis (MSMatch) application was built to accelerate non-targeted analysis projects by searching experiment result data in mzML format for matches against a curated mass spectral library of compounds and annotated fragments. MSMatch is a web application built using the Shiny package in R and installs alongside DIMSpec and is one example of a tool that can built on top of the DIMSpec tool set. Databases built and managed with DIMSpec are SQLite files used within a distributed R Project. Scripts for automated setup are included. For this initial release, DIMSpec is distributed with data populated for per- and polyfluorinated alkyl substances (PFAS); that effort has been primarily supported by the Department of Defense Strategic Environmental Research and Development Program (DOD-SERDP), project number ER20-1056.

This section serves as a user guide for using the MSMatch application.

Set Up Instructions

The MSMatch application installs alongside DIMSpec. If there is continued (or expanded) interest, the project could be turned into an R package installable directly from GitHub with additional development or this tool can be deployed to a shiny server to avoid the need for launching or maintaining it locally. For now, this application is distributed for demonstration and evaluation with an implementation of NIST DIMSpec containing high resolution accurate mass spectrometry data for per- and polyfluorinated alkyl substances (PFAS). The R project can be opened in RStudio[vi] which may be downloaded and installed free of charge if not already installed. Initial set up does require an internet connection to install dependencies; on a system which does not contain any software components this can take a considerable amount of time.

Refer to the System Requirements section for installation details.

Input File Format Requirements

To use MSMatch, raw data files produced by a mass spectrometer must be converted into mzML format (Deutsch 2010) using Proteowizard's msConvert software (Chambers et al. 2012). There are specific parameters that must be used during conversion.

```
Filter:           Threshold peak filter
Threshold type:   absolute
Orientation:      most intense
```

```
Value:            1
Filter:           Peak picking
Algorithm:        vendor
MS levels:        1-2
```

A more detailed user guide for converting the files is provided as a vignette.

Launching MSMatch

Launch this tool similarly to other "shiny"-based tools provided as part of DIMSpec. In brief, this can be done from a terminal or the R console, though the preferred method is to use RStudio (RStudio Team 2020). The following commands are typical given an existing installation of R or RStudio and should always be run from the project directory. The shiny package (Chang et al. 2021) and other necessary R packages will be installed if it not already available by running the script at `R/compliance.R`, but shiny is the only package required to start the application. When first run, it may take a moment to install necessary dependencies and launch the application programming interface (API) server.

Terminal[vii]

R.exe "shiny::runApp('inst/apps/spectral_match')"

R console

```
shiny::runApp('inst/apps/spectral_match')
```

RStudio

Open the ".Rproj" project file in RStudio, navigate to the "inst/apps/spectral_match" directory, open one of the "global.R," "server.R," or "ui.R" files, and click the "Run App" button. Files open in an RStudio project will remain open by default when RStudio is closed, allowing users to quickly relaunch by simply loading the project. For best performance, ensure "Run External" is selected from the menu "carrot" on the right to launch the application in your system's default web browser. This application has been tested on Chrome, Edge, and Firefox.

Alternatively, once the compliance script has been executed MSMatch can be launched using `start_app("spectral_match")`.

Once launched the API server will remain active until stopped, allowing users to freely launch, close, and relaunch any shiny apps dependent upon it much more quickly. The application is fluid and will dynamically resize to fit the dimensions of a browser window. By default, the server does not stop when the browser is closed. This means that, once started, it is available by navigating a web browser back to the URL where it launched until the server is shut down.

If anything is needed from the user, interactive feedback will occur in the console from which it was launched. Install any packages required if prompted by the application. Once the package environment requirements have been satisfied and the server has spun up, which may take a moment, the tool will launch (Figure 1) either in the RStudio viewer or the browser

.

Using MSMatch

Every effort has been made to make MSMatch as intuitive for users as possible. Set up may require a bit of effort on certain systems, but once the application launches it should be straightforward; please contact the authors directly or email pfas@nist.gov for support, or with any questions or suggestions.

Hints in the form of tooltips are provided throughout; hover over question mark icons or controls to see them. These can be toggled on and off at any time using the "Show Tooltips" toggle button at the bottom left of the application window (see Figure 1 inset at bottom right). If enabled, advanced search settings can be similarly toggled on and off for the session (see Application Settings for instructions on how to set default accessibility and settings for tooltips and advanced settings). The "hamburger" (≡) icon at the top left of the screen will collapse the left-hand navigation panel to provide more horizontal room on smaller screens, though the application will rearrange itself when screens are smaller than a minimum width.

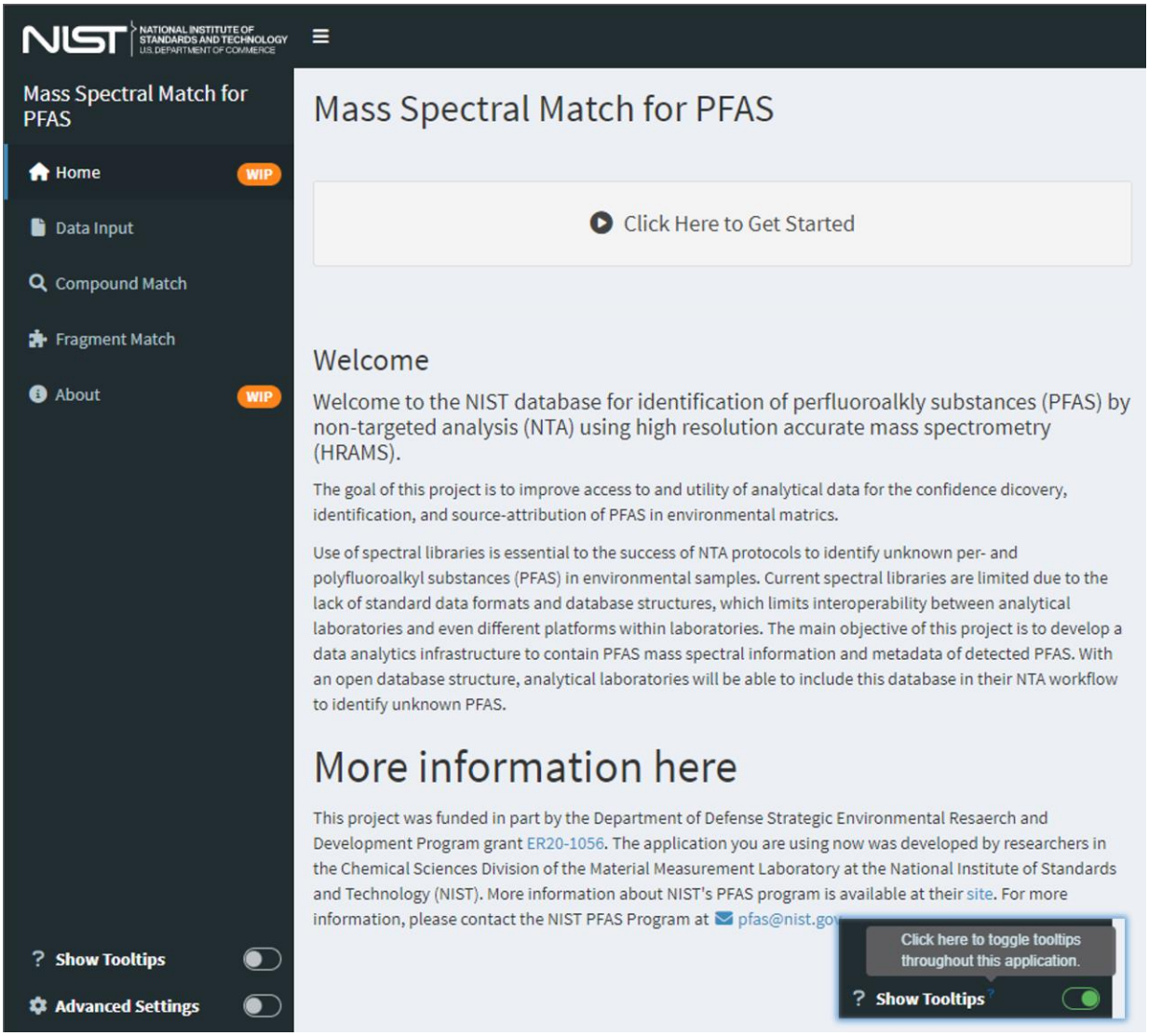


Figure 1. The home page for the MSMatch web application contains basic information about the application and can be tailored easily for each use case for reuse.

Click the "Click Here to Get Started" button to begin. This will activate the "Data Input" page. Example data files are provided in the project directory or upon request ("example/PFAC30PAR_PFCA2.mzML" and "example/example_peaklist.csv").

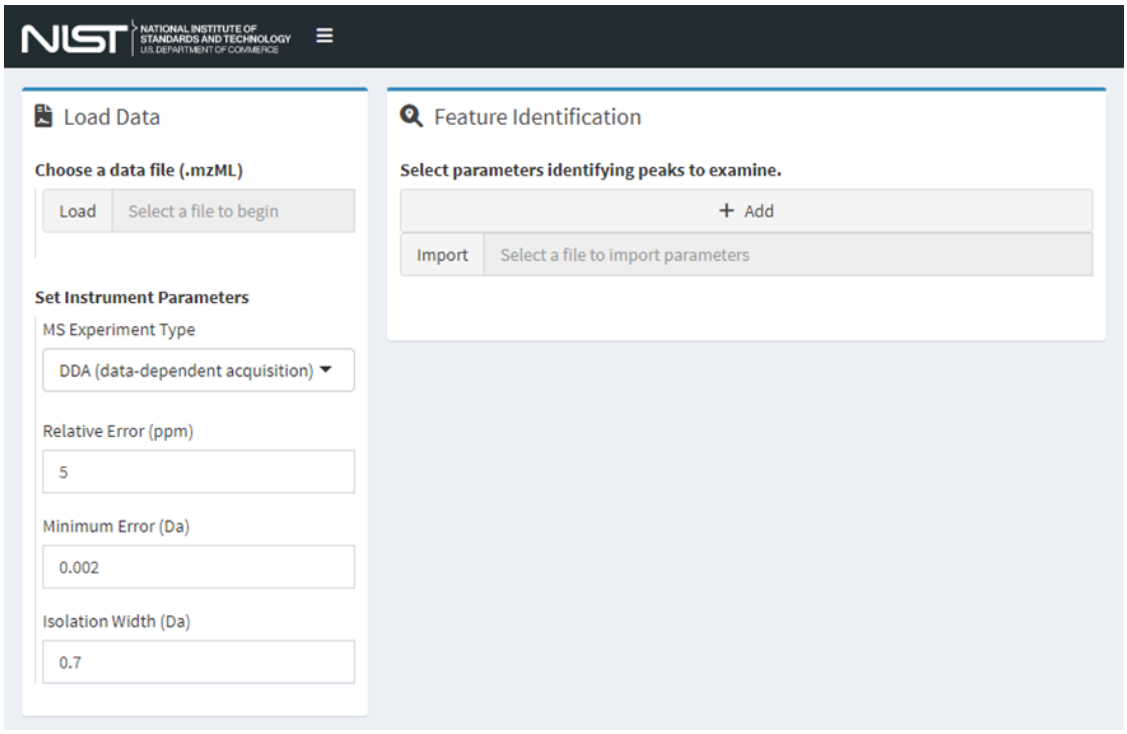


Figure 2. The data input page of MSMatch where data files, experiment parameters, and features of interest are identified. Workflow guidance options become available once data are provided.

All input values are validated against expectations and will flag the user if invalid values are used.

Step 1. Load an mzML Data File

MSMatch accepts files in the mzML format, see the previous section Input file format requirements for more details. Either click the "Load" button to select a local file or click and drag one from your file system to that widget. Only .mzML files are accepted using this release. Set instrument parameters to match those used in the experiment using the controls provided.

Step 2. Identify Features of Interest

Two methods (Figure 3) are supported to identify features of interest by mass-to-charge ratio and retention time properties. Either use case is fully supported. Users may:

1. import a file (either .csv, .xls, or .xlsx, though workbooks should have relevant data in the first worksheet) and identify which columns contain the correct information (Figure 3, left).

- Click “Import” and select a file of interest from your local computer or drag and drop a file to this input.
- Use this method if you have a file containing features of interest from other procedures or software outputs to quickly import many feature properties.
- Select a column that corresponds to each property.
- To append to the current list, keep the checkbox checked. To overwrite, uncheck this box.
- Click “Load Parameters” to validate and add parameters or “Cancel” to abort this operation.
- Repeat until all files are imported. or

2. click the “Add” button and enter search parameters one at a time (Figure 3, left); repeat this process to add more.
   - Add numeric values for all items.
   - Click “Save Parameters” to validate and add or “Cancel” to abort this operation

- Users receive feedback on the form if values are left blank or if they do not meet expectations (e.g. centroid is after peak start and before peak end).
- Values should all be numeric in nature.
- This list may be edited after import (Figure 4).

Data are ready to be processed once features of interest are added. Selecting any row in the resulting table makes two additional functions available (Figure 4, right). With a row selected, click “Remove” to delete it or “Edit” to bring up the same form as above (Figure 3, right), change the values, and click “Save Parameters.” All records remaining in the feature of interest list will be available to search widgets on subsequent pages.

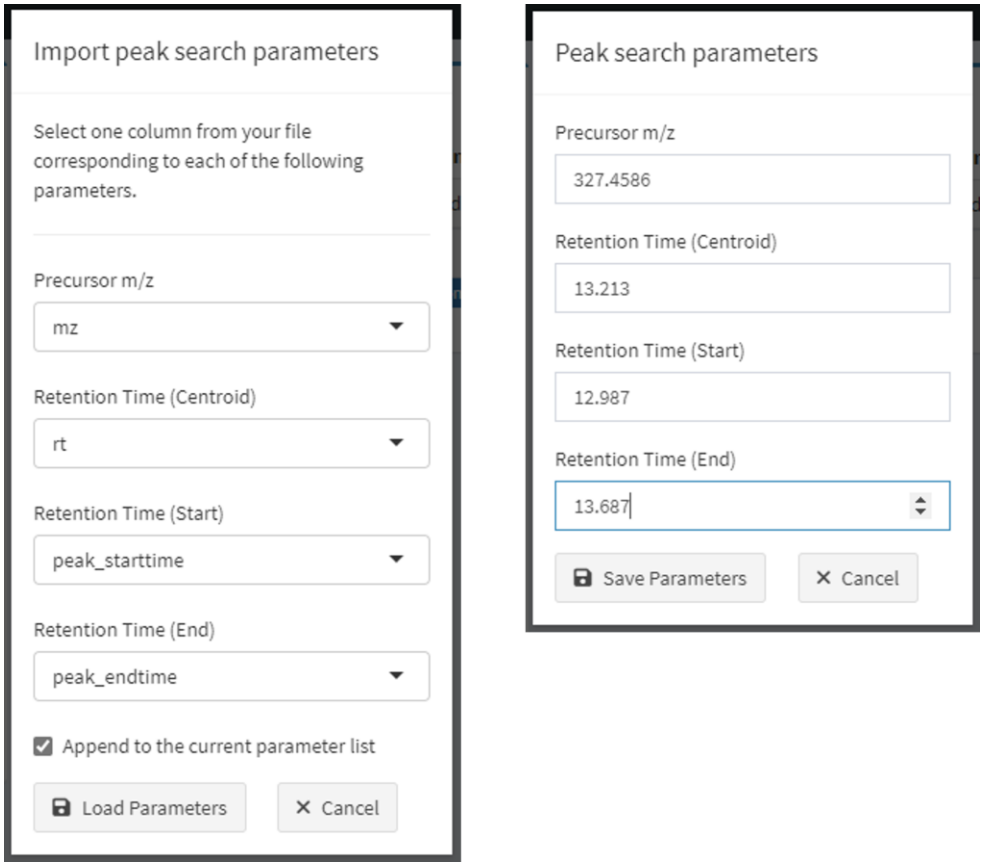


Figure 3. Dialogs to identify features of interest by upload (left) or manually by clicking the Add button (right).

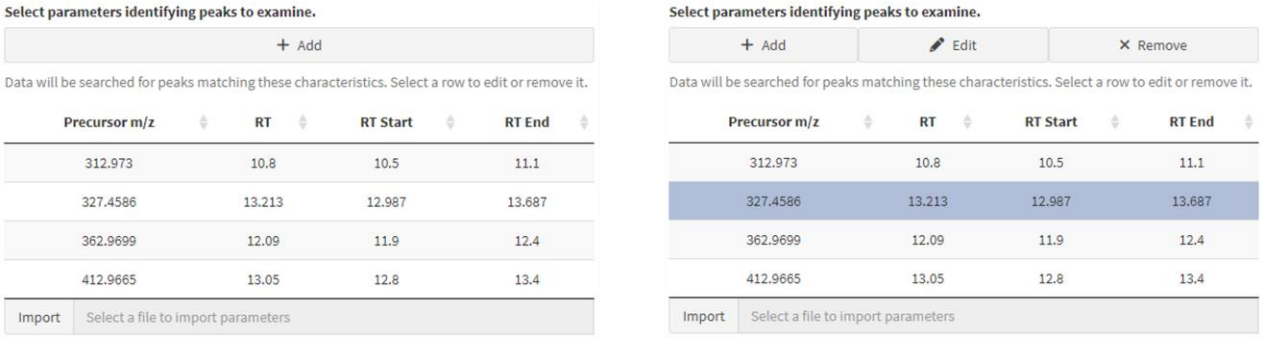


Figure 4. Manage the feature identification list (left) interactively by adding, editing, or removing features as needed (right) by selecting a row from the table and clicking the appropriate button.

Step 3. Generate the Search Object

Click the "Process Data" button (Figure 5) to filter and recast data in the .mzML file according to defined feature properties. (In further screenshots the manually added row [m/z 327.4586] has been removed. This will unlock mass spectral matching actions; click one of the buttons or navigate to the desired page using the navigation panel on the left.

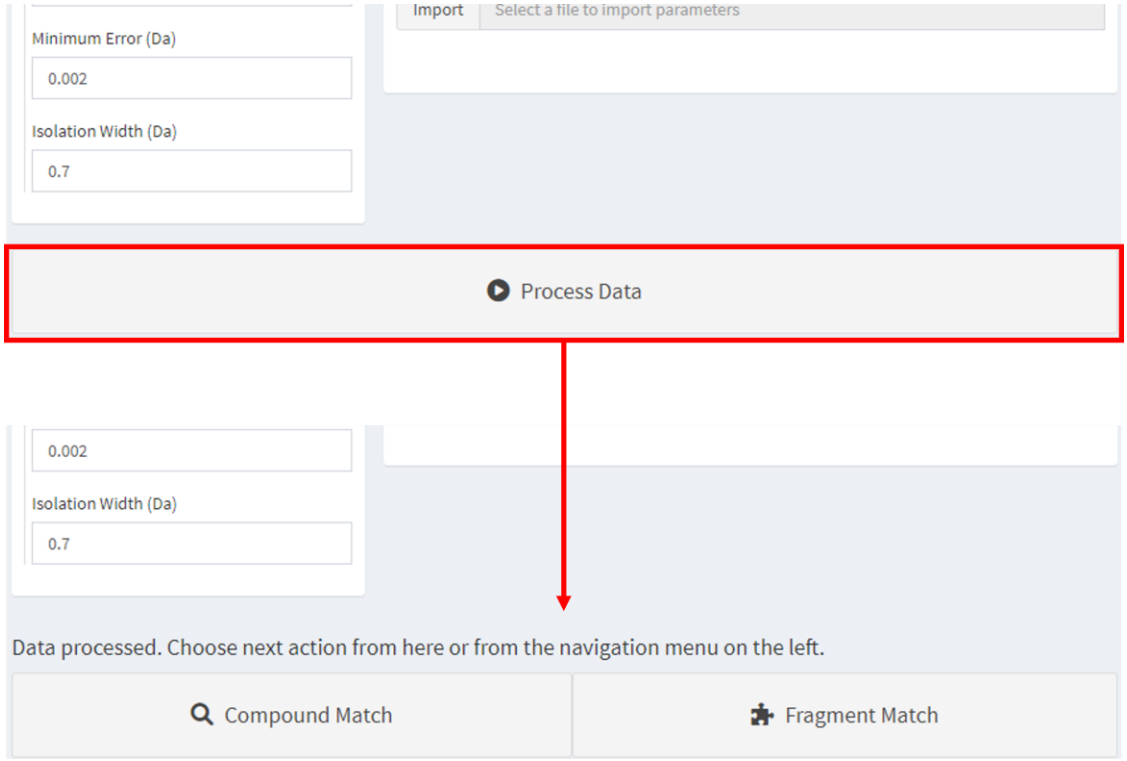


Figure 5. Click "Process Data" to validate experiment data and choose one of the options that appear or navigate to the desired page using the left-hand navigation menu.

Step 4. Explore Results

Algorithmic matching of provided mass spectral data for features of interest are matched against data stored in the attached database. Matching algorithms are described in detail in Compound and Fragment Match Algorithms. In brief, data meeting properties of a feature of interest are extracted from the provided .mzML file given the reported mass accuracy settings of the experiment and mass-to-charge ratios for known compounds and fragments are searched within an uncertainty boundary range and returned from the database. Results are then stored in the application server and displayed to the user.

Match Compounds

Click the "Compound Match" button from the previous page or select "Compound Match" from the left-hand navigation menu to match features of interest from the mzML file to known compounds in the database.

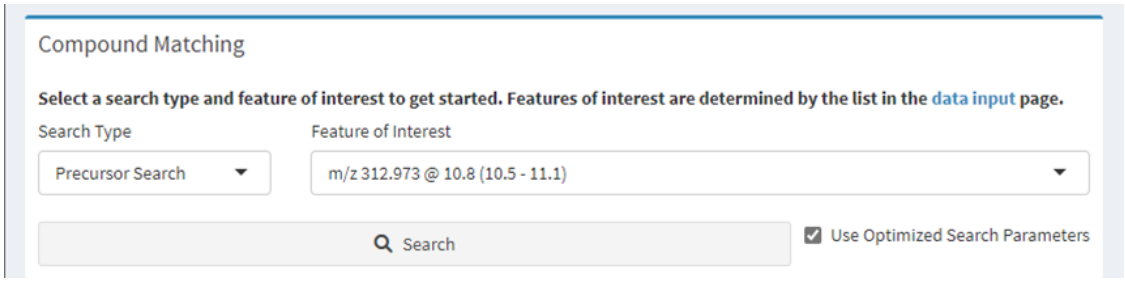


Figure 6. Compound matching options. Select a feature of interest and search type, then click the 'Search' button.

Select a feature of interest from the drop-down box and click the "Search" button.

In most cases the "Precursor Search" option should remain selected; the other option is "All" which takes a considerable amount of time and may yield poor matches. The "Use Optimized Search Parameters" checkbox will utilize a set of predefined properties for known compounds to accelerate the search; uncheck this box to perform a wider search.

Narrative results are provided regarding the top match and the match currently being viewed, including a method summary for how the reference was measured. The spectral comparison is visualized in a "butterfly plot" showing measurements in black and the comparison (database) spectrum in red; toggle the different fragmentation levels (e.g. MS1 vs MS2) to view those independently (Figure 7).

The table at the right displays compound match identities and their match scores. Click the green plus icon to expand any given row of the table or click a different row to examine that match and update the plot and method narrative (Figure 8). This table may be downloaded using the buttons at the bottom left of the table.

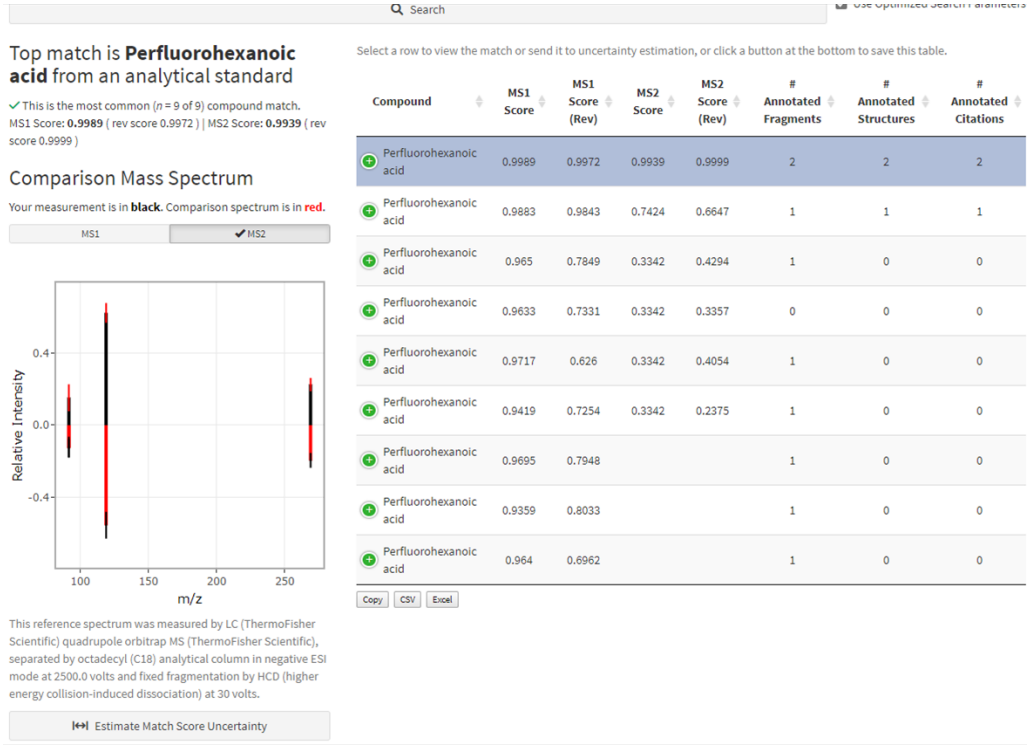


Figure 7. Results of a compound match for the selected feature of interest. (Inset: Download results using the buttons at the bottom left of the match table.)

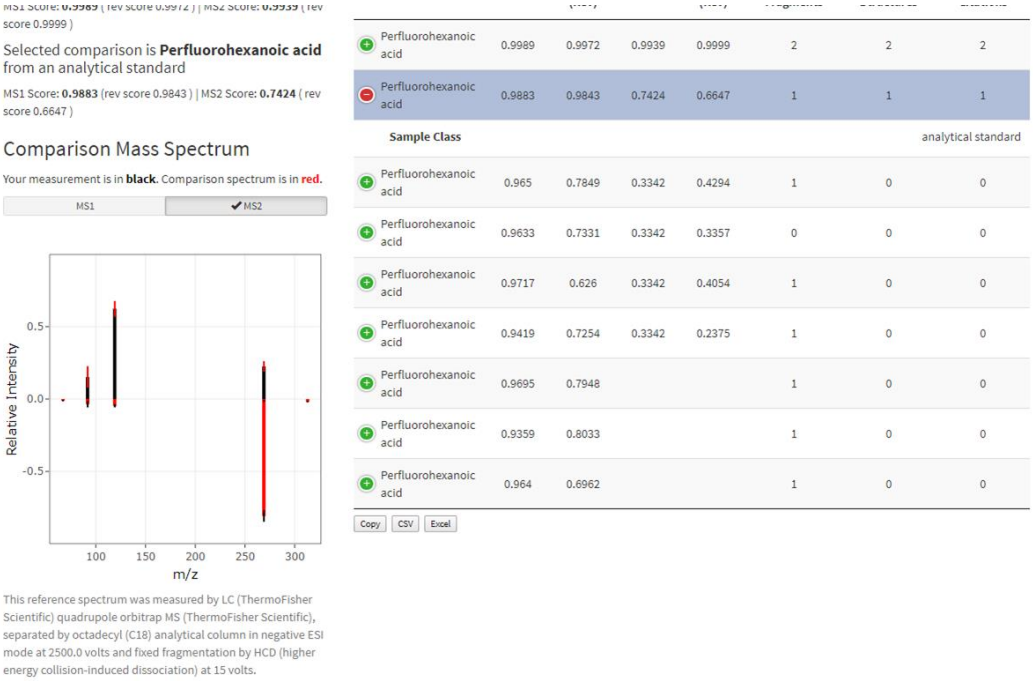


Figure 8. Results change in real time when different rows are selected from the table, updating the narrative, butterfly plot, and method narrative (compare with Figure 6).

Evaluation of match score uncertainty is also provided. Click the "Estimate Match Score Uncertainty" button below the butterfly plot (Figure 7) to evaluate the spread in match scores for the currently selected match (Figure 9). Results from a bootstrapped version of the match algorithm are displayed as boxplots for both forward and reverse matches. Toggle the different fragmentation levels (e.g. MS1 vs MS2) to view each. Change the number of bootstrap iterations to use and click "Calculate Uncertainty" to run the estimation again. Click the "Close" button to return to the compound match screen and change the match being evaluated. The calculation of mass spectral uncertainty and estimation of the distribution of the match scores is described in Place, Benjamin J. (2021a).

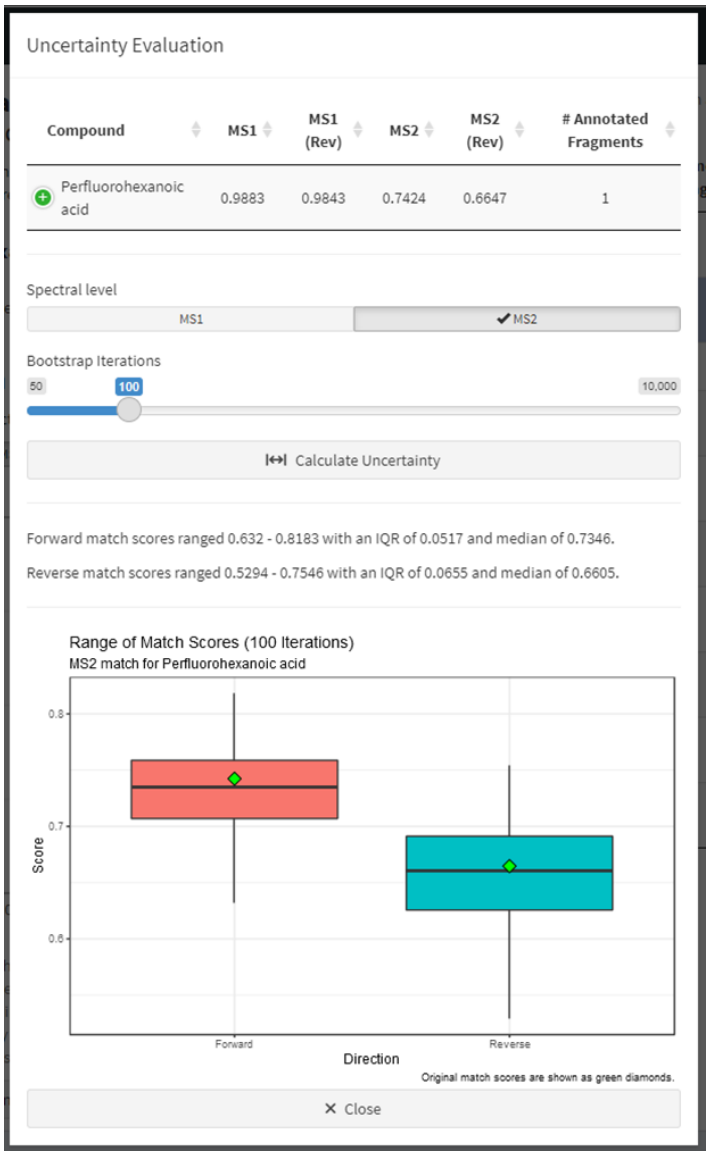


Figure 9. Estimation of match score uncertainty for any selected match candidate.

The match result table (Figures 7 and 8) offers several options.

- Sort using the column headers (results are by default ordered by MS1 score and MS2 score, with ties being decided by reverse match scores and the number of annotated fragments).
- Download the resulting match table by either copying it to the clipboard ("Copy") or downloading in either CSV or XLS file formats using the buttons at the bottom left of the table (see Figure 6).
- Select any row in the table to update the narratives and plots or evaluate match score uncertainty for that match.

If no compound matches are found, users are flagged to that effect. Proceed to Match Fragments using the navigation menu to identify fragments matching known annotations.

Match Fragments

Click the "Fragment Match" button from the data input page (Figure 5) or select "Fragment Match" from the left-hand navigation menu to match analytical fragments within features of interest from the .mzML file with previously annotated fragments in the database.

Select a feature of interest from the drop-down box just as with compounds (features are defined in Step 2: Identify Features of Interest) and click the "Search" button (Figure 10).

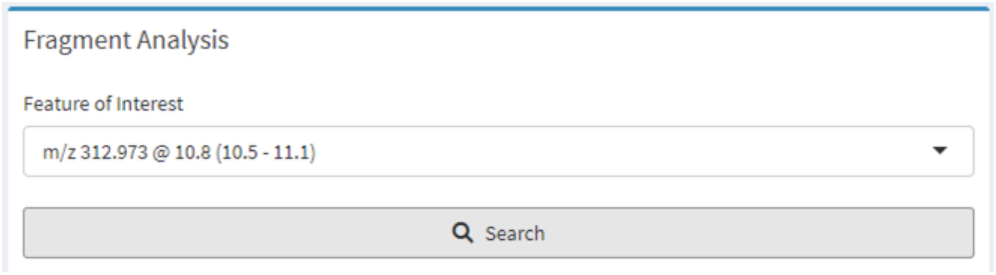


Figure 10. Fragment matching options. Select a feature of interest and click the 'Search' button.

Fragments measured within the feature of interest will be matched against database fragments with known annotations. Complicated spectra may take a moment but generally completes within 30 seconds yielding a variety of results to indicate possible compound identity. Results are presented as an annotated mass spectral uncertainty plot (Figure 11 left) and additional information about measured spectra are provided in an expanding table (Figure 11 right).

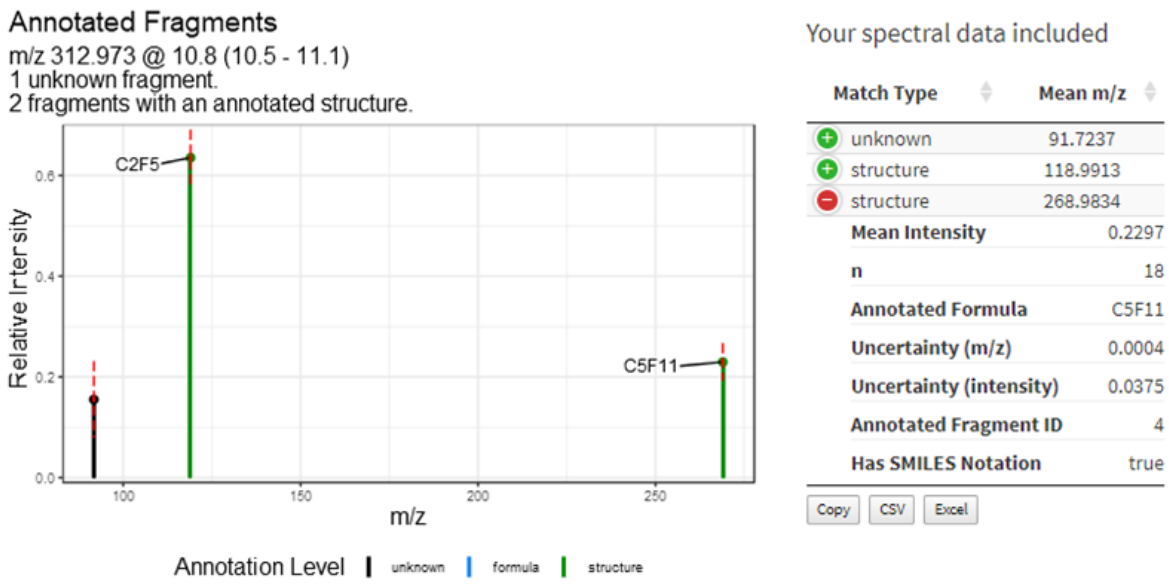


Figure 11. Graphical display of fragment annotations on an uncertainty mass spectral plot and the measured data provided to MSMatch.

Matched fragment annotations and associated metadata are provided below this output (Figure 12). Match records are located in the expandable table to the left. As matches may have structural annotation or not, these are separated to indicate confidence and annotations with structural notation are displayed at the top. Select a row in the table (the top record is selected by default) to update the following displays.

- A human readable measurement narrative about the known fragment.
- If structural notation is present a molecular model is displayed (requires rdkit to be active in the API server)
- Compounds and peaks within which this fragment has been previously annotated appear in the table to the right. Select the tab to switch between compounds and peaks.

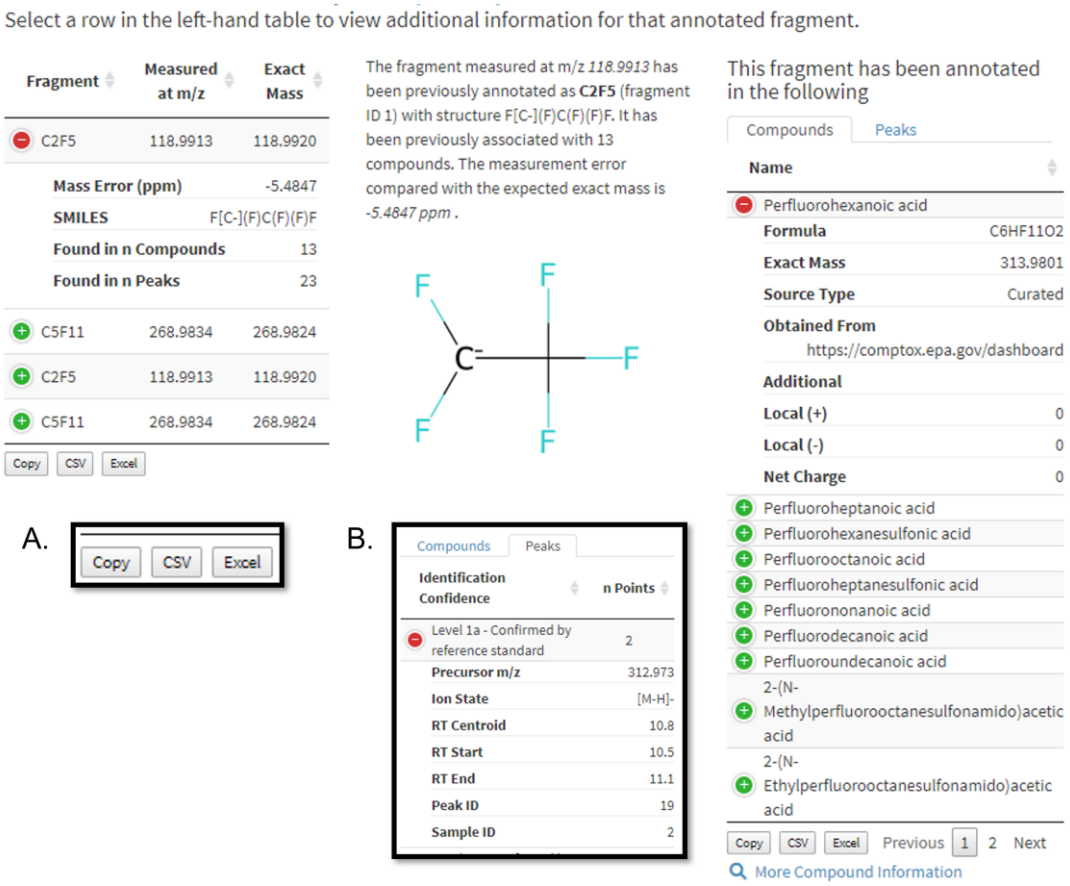


Figure 12. Results from matching user provided data to annotated fragments with associated metadata. Selecting a row in the left-hand table will update data to the right in real time. Inset A: click a button to save that table's results. Inset B: an example of peak information available.

Two options are available for more contextual information regarding compounds and peaks.

- Click **More Compound Information** to list other known or generated aliases for a compound and provides links to those resources if available. These aliases have either been collated from existing locations or, in the case of most machine-readable identifiers, generated using rdkit.

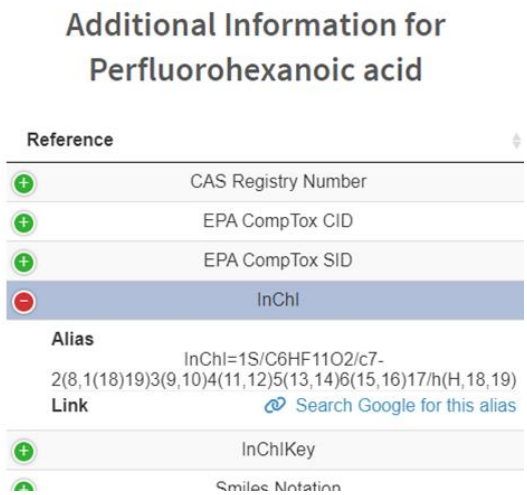


Figure 13. Additional information available for compounds.

- Click **More Peak Information** to provide a human readable narrative regarding measurement methods and sample information provided as part of the database accession process. Narratives are constructed directly from the underlying linked data tables by the database and stored as a database view.

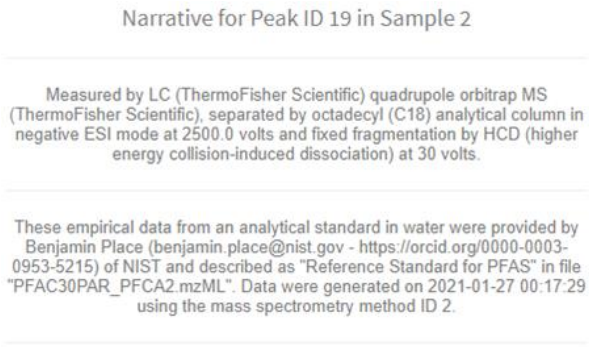


Figure 14. Additional information available for peaks.

Step 5. Closing Down

When finished using the application, typing the escape key at the R console is the simplest way to stop the server and exit the application. If using RStudio there is a "stop sign" button at the top right of the console pane that will also stop it. When finished completely with the project, users also need to shut down the API server.

- Loading the entire project from the compliance script (i.e. MSMatch was launched using `start_app("spectral_match")`) provides additional actions and includes a live database connection with the ability to read data into tables and preserve them for further analysis. Use the function `close_up_shop()` with the argument `back_up_connected_tbls` set to `TRUE` to preserve these, or the default FALSE to simply close all connections including the API server).
- If launching the app directly and using the default settings there will be a session object named `plumber_service` connected to that server. To stop it, use the `api_stop` function from the console or stop the service directly using `plumber_service\$kill()`; it will also generally stop when the calling R process closes (e.g. when RStudio is closed), but it is highly recommended to stop it manually to prevent hanging connections.
- After closing all connections, a hanging connection may be indicated by the presence of "-shm" and "-wal" files in the project directory. Flushing these hanging connections is not required but is recommended.
  - If launching MSMatch with the compliance script, run `close_up_shop()` again.
  - Otherwise flush those connections by directly connecting and disconnecting with the DBI package:

```
    con <- DBI::dbConnect(
        RSQLite()::SQLite,
        "nist_pfas_nta_dev.sqlite")
    DBI::dbDisconnect(con)
    rm(con)
```

Feature requests, suggestions, and bug reports are most conveniently submitted as issues via GitHub but may also be submitted by contacting the authors of this ROA. New functionality suggestions are encouraged as the project tooling develops. Likewise, if the functionality demonstrated here is of interest to projects outside of PFAS, this is only one example implementation of the underlying technology stack (i.e. DIMSpec); contact the authors to see if your mass spectrometry data would be amenable to that framework as other implementation suggestions are encouraged and a larger goal of the project to cohesively manage mass spectrometry data for non-targeted analysis within the Chemical Sciences Division.

This concludes the User Guide for the Mass Spectral Match (MSMatch) web application. The following section contains technical details about the implementation and user customization of this digital assistant.

Technical Details

Implementation and environment details for the MSMatch application largely follow those for DIMSpec. See the sections on System Requirements, Environment Resolution, Shiny Applications, Plumber for the API implementation which is required for this application, and Python Integration for chemometrics support.

Technical details in this section will describe only the MSMatch application found in the `inst/apps/spectral_match` directory and, unless otherwise noted, all files referred to hereafter refer to that directory.

Mass Spectral Search Object

In order to use the functions related to the database searching, library matching, and fragment matching, a *mass spectral search object* must be generated. A *mass spectral search object* is a nested list in the R environment that contains the following:

- Peak search parameters input in Step 1, including instrument performance parameters, in a `search_df` dataframe
- MS1 and MS2 (if available) mass spectra peak tables, nested lists with names `ms1_pt` and `ms2_pt` respectively. Each peak table contains:
  - `peaktable_mass`: a dataframe of *m/z* values with rows representing binned *m/z* values and columns representing individual scans

- `peaktable_int`: a dataframe of intensity values with the same shape and organization as the `peaktable_mass` dataframe
- `EIC`: dataframe containing paired `time` and `int` (intensity) values corresponding to the MS1 extracted ion chromatogram (EIC) of the precursor ion stated in the peak search parameters
- `ms1scans`: integer vector that indicates the scan number of the MS2 scans
- `ms2scans`: integer vector that indicates the scan number of the MS2 scans

The *Mass Spectral Search Object* is converted into uncertainty mass spectra (MS1 and MS2, if available) through the `get_ums` function for subsequent analysis.

Compound and Fragment Match Algorithms

All relevant functions for compound and fragment match algorithms are in the `R/spectral_analysis` directory. Currently, the compound search tool of MSMatch can search the database using the precursor ion *m/z* of the unknown compound (`search_precursor` function) to reduce the processing time and compare only reference mass spectra with the same (defined below) precursor ion *m/z* or compare the mass spectra of the unknown compound to all compounds in the database (`search_all` function). The `search_precursor` function first identifies all peaks that have precursor ion m/z values that are within a range, as dependent on the type of MS2 experiment used:

Data-Dependent Acquisition (DDA, TopN), reference precursor ion *m/z* range:

$$\left(m_p - \max\left(error,\ min.error\right),\ m_p + \max\left(error,\ min.error\right)\right)$$

Sequential Windows of All Theoretical Masses (SWATH, SONAR), range:

$$\left(m_p - \frac{1}{2}\,error,\quad m_p + \frac{1}{2}\,error\right)$$

Data-Independent Acquisition (DIA, AIF), range:

$$(0,\quad \infty)$$

Where mp is the precursor ion *m/z* of the unknown compound, *error* is the instrument relative error (in ppm) for DDA and the width of the window in SWATH, and *min.error* is the instrument absolute minimum error (in Da). For DIA, the range is all possible positive

masses so that, functionally, the `search_precursor` function is no different than the `search_all` function.

To determine the match score between the unknown compound mass spectra (MS1 and MS2) and the reference mass spectra, the dot product of the mean mass spectra is calculated using the `compare_ms` function, which uses the respective mass errors and minimum mass errors of the unknown mass spectra and reference mass spectra. To determine the uncertainty distribution of the match score, the `bootstrap_compare_ms` function is used. The application of this function is described in Place, Benjamin J. (2021a).

For the fragment search tool, the function `get_annotated_fragments` is used, where all *m/z* values in the MS2 mass spectrum of the unknown compound are searched against all annotated fragments in the `norm_fragments` view using the instrument relative mass error and absolute minimum mass error as boundary conditions. Annotated fragments are fragment *m/z* values that have elemental formulas assigned; SMILES structural notation can also be assigned but is not required for annotation.

Application Settings

Many global application settings are customizable by modifying the `global.R` file within the application directory. Changes to those listed here should not cause issues, but other settings in this file may result in instability. Anywhere a `TRUE` or `FALSE` value is indicated should only be `TRUE` or `FALSE`. The most germane user settings include:

Table 4. Example application settings in the `global.R` file.

Setting	Default Value	Description
APP_TITLE	"Mass Spectral Match for PFAS"	Set the application title which will appear in the browser tab when running.
default_title	`APP_TITLE`	The application name that will display inside the page. Overrides the default of APP_TITLE if desired.
app_ns	`paste0("app_", app_name)`	The namespace to direct logging messages.
dev	`FALSE`	Whether to launch the application in development mode, which will allow for interactive interrogation of application state by unlocking a button, provide the opportunity to pre-load data for testing, and display most hidden elements for inspection.
enable_adv_use	`TRUE`	Whether to launch the application with a toggle to display advanced controls available to the user.

advanced_use	FALSE	Whether to launch the application with advanced controls visible by default.
enable_more_help	TRUE	Whether to launch the application with a toggle to display tooltips to the user.
advanced_use	FALSE	Whether to launch the application with tooltips by default.
tooltip_text	"www/tooltips.csv"	A data source for populating tooltips. This should point to a readable file (e.g. www/tooltips.csv) that is coerced to a named vector.
toy_data	FALSE	Whether to launch the application with "toy" data for testing purposes. This loads data from "src_toy_data" and "src_toy_parameters" during application spin up so the tester does not have to manually upload files or add data.
src_toy_data	"toy_data.RDS"	Name of the RDS file housing "toy" data from a previously processed mzML file.
src_toy_parameters	"toy_parameters.RDS"	Name of the RDS file housing "toy" data describing features of interest to use for testing.
need_files	Character vector	File paths for files containing application functions.
app_settings	Named list	Properties used to populate controls within the application, by name (e.g. min, max, value, etc.).
jscode	Text/HTML/Javascript	Custom javascript functions which either execute automatically or by calling the function server side.
file_MB_limit	Numeric	The maximum size in megabytes for a single uploaded file. Shiny's built-in default is 5 MB.

Future Development

Both the R/Shiny and python code bases are fully extensible for future functionality needs, as is the underlying database infrastructure for custom tables and views. Future development may include deployment of and to a Shiny server to serve this as a hosted web application, extending the python code to analyze data of various formats from different instruments, and adding analysis features and functionality (e.g. high resolution

plot generation and download or supporting the full workflow from instrument through import and to report generation) requested by stakeholders.

Conclusions

The Mass Spectral Match for Non-Targeted Analysis application provides a new way to make NTA tools developed at NIST using the Database Infrastructure for Mass Spectrometry accessible to both internal and external stakeholders. It is the first demonstration of tools that can be built on top of databases conforming to the DIMSpec project, which can be repurposed for any class of chemicals or project of interest using mass spectrometry data.

References

Adusumilli, Ravali and Mallick, Parag. 2017. "Data Conversion with ProteoWizard msConvert." *Methods in Molecular Biology (Clifton, N.J.)* 1550: 339–68. https://doi.org/10.1007/978-1-4939-6747-6_23.

Allaire, JJ, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, Winston Chang, and Richard Iannone. 2022. *Rmarkdown: Dynamic Documents for r*. https://CRAN.R-project.org/package=rmarkdown.

Bache, Stefan Milton, and Hadley Wickham. 2022. *Magrittr: A Forward-Pipe Operator for r*. https://CRAN.R-project.org/package=magrittr.

Barthelme, Simon. 2022. *Imager: Image Processing Library Based on CImg*. https://CRAN.R-project.org/package=imager.

Chambers, Matthew C., Brendan Maclean, Robert Burke, Dario Amodei, Daniel L. Ruderman, Steffen Neumann, Laurent Gatto, et al. 2012. "A Cross-Platform Toolkit for Mass Spectrometry and Proteomics." *Nature Biotechnology* 30 (10): 918–20. https://doi.org/10.1038/nbt.2377.

Chang, Winston, Joe Cheng, JJ Allaire, Carson Sievert, Barret Schloerke, Yihui Xie, Jeff Allen, Jonathan McPherson, Alan Dipert, and Barbara Borges. 2021. *Shiny: Web Application Framework for r*. https://shiny.rstudio.com/.

Deutsch, Eric W. 2010. "Mass Spectrometer Output File Format mzML." In *Proteome Bioinformatics*, edited by Simon J. Hubbard and Andrew R. Jones, 319–31. Methods in Molecular Biology™. Totowa, NJ: Humana Press. https://doi.org/10.1007/978-1-60761-444-9_22.

Gagolewski, Marek. 2021. "Stringi: Fast and Portable Character String Processing in r." *Journal of Statistical Software*.

Gagolewski, Marek, Bartek Tartanus, others; Unicode, Inc., et al. 2021. *Stringi: Character String Processing Facilities*. https://CRAN.R-project.org/package=stringi.

Grolemund, Garrett, and Hadley Wickham. 2011. "Dates and Times Made Easy with lubridate." *Journal of Statistical Software* 40 (3): 1–25. https://www.jstatsoft.org/v40/i03/.

Henry, Lionel, and Hadley Wickham. 2020. *Purrr: Functional Programming Tools*. https://CRAN.R-project.org/package=purrr.

Hester, Jim, and Jennifer Bryan. 2022. *Glue: Interpreted String Literals*. https://CRAN.R-project.org/package=glue.

Müller, Kirill. 2020. *Here: A Simpler Way to Find Your Files*. https://CRAN.R-project.org/package=here.

Müller, Kirill, and Hadley Wickham. 2021. *Tibble: Simple Data Frames*. https://CRAN.R-project.org/package=tibble.

Müller, Kirill, Hadley Wickham, David A. James, and Seth Falcon. 2021. *RSQLite: SQLite Interface for r*. https://CRAN.R-project.org/package=RSQLite.

Ooms, Jeroen. 2014. "The Jsonlite Package: A Practical and Consistent Mapping Between JSON Data and r Objects." *arXiv:1403.2805 [Stat.CO]*. https://arxiv.org/abs/1403.2805.

———. 2022. *Jsonlite: A Simple and Robust JSON Parser and Generator for r*. https://CRAN.R-project.org/package=jsonlite.

Place, Benjamin J. 2021a. "Development of a Data Analysis Tool to Determine the Measurement Variability of Consensus Mass Spectra." *Journal of the American Society for Mass Spectrometry* 32 (3): 707–15. https://doi.org/10.1021/jasms.0c00423.

———. 2021b. "Suspect List of Possible Per- and Polyfluoroalkyl Substances (PFAS)." National Institute of Standards; Technology. https://doi.org/10.18434/MDS2-2387.

R Core Team. 2021. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

R Special Interest Group on Databases (R-SIG-DB), Hadley Wickham, and Kirill Müller. 2021. *DBI: R Database Interface*. https://CRAN.R-project.org/package=DBI.

RDKit: Open-Soure Cheminformatics (version 2021.09.4). n.d. https://doi.org/10.5281/zenodo.5835217.

RStudio Team. 2020. *RStudio: Integrated Development Environment for r*. Boston, MA: RStudio, PBC. http://www.rstudio.com/.

Spinu, Vitalie, Garrett Grolemund, and Hadley Wickham. 2021. *Lubridate: Make Dealing with Dates a Little Easier*. https://CRAN.R-project.org/package=lubridate.

Temple Lang, Duncan. 2021. *XML: Tools for Parsing and Generating XML Within r and s-Plus*. http://www.omegahat.net/RSXML.

Urbanek, Simon. 2015. *Base64enc: Tools for Base64 Encoding*. http://www.rforge.net/base64enc.

Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. https://ggplot2.tidyverse.org.

———. 2019. *Stringr: Simple, Consistent Wrappers for Common String Operations*. https://CRAN.R-project.org/package=stringr.

———. 2021a. *Forcats: Tools for Working with Categorical Variables (Factors)*. https://CRAN.R-project.org/package=forcats.

———. 2021b. *Tidyverse: Easily Install and Load the Tidyverse*. https://CRAN.R-project.org/package=tidyverse.

Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, et al. 2019. "Welcome to the tidyverse." *Journal of Open Source Software* 4 (43): 1686. https://doi.org/10.21105/joss.01686.

Wickham, Hadley, Jennifer Bryan, and Malcolm Barrett. 2021. *Usethis: Automate Package and Project Setup*. https://CRAN.R-project.org/package=usethis.

Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, and Dewey Dunnington. 2021. *Ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. https://CRAN.R-project.org/package=ggplot2.

Wickham, Hadley, Romain François, Lionel Henry, and Kirill Müller. 2021. *Dplyr: A Grammar of Data Manipulation*. https://CRAN.R-project.org/package=dplyr.

Wickham, Hadley, and Maximilian Girlich. 2022. *Tidyr: Tidy Messy Data*. https://CRAN.R-project.org/package=tidyr.

Wickham, Hadley, Maximilian Girlich, and Edgar Ruiz. 2021. *Dbplyr: A Dplyr Back End for Databases*. https://CRAN.R-project.org/package=dbplyr.

Wickham, Hadley, Jim Hester, and Jennifer Bryan. 2022. *Readr: Read Rectangular Text Data*. https://CRAN.R-project.org/package=readr.

Xie, Yihui. 2014. "Knitr: A Comprehensive Tool for Reproducible Research in R." In *Implementing Reproducible Computational Research*, edited by Victoria Stodden, Friedrich Leisch, and Roger D. Peng. Chapman; Hall/CRC. http://www.crcpress.com/product/isbn/9781466561595.

———. 2015. *Dynamic Documents with R and Knitr*. 2nd ed. Boca Raton, Florida: Chapman; Hall/CRC. https://yihui.org/knitr/.

———. 2016. *Bookdown: Authoring Books and Technical Documents with R Markdown*. Boca Raton, Florida: Chapman; Hall/CRC. https://bookdown.org/yihui/bookdown.

———. 2022a. *Bookdown: Authoring Books and Technical Documents with r Markdown*. https://CRAN.R-project.org/package=bookdown.

———. 2022b. *Knitr: A General-Purpose Package for Dynamic Report Generation in r*. https://yihui.org/knitr/.

Xie, Yihui, J. J. Allaire, and Garrett Grolemund. 2018. *R Markdown: The Definitive Guide*. Boca Raton, Florida: Chapman; Hall/CRC. https://bookdown.org/yihui/rmarkdown.

Xie, Yihui, Christophe Dervieux, and Emily Riederer. 2020. *R Markdown Cookbook*. Boca Raton, Florida: Chapman; Hall/CRC. https://bookdown.org/yihui/rmarkdown-cookbook.

[i] NIST-developed software is provided by NIST as a public service. You may use, copy, and distribute copies of the software in any medium, provided that you keep intact this entire notice. You may improve, modify, and create derivative works of the software or any portion of the software, and you may copy and distribute such modifications or works. Modified works should carry a notice stating that you changed the software and should note the date and nature of any such change. Please explicitly acknowledge the National Institute of Standards and Technology as the source of the software.

NIST-developed software is expressly provided "AS IS." NIST MAKES NO WARRANTY OF ANY KIND, EXPRESS, IMPLIED, IN FACT, OR ARISING BY OPERATION OF LAW, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND DATA ACCURACY. NIST NEITHER REPRESENTS NOR WARRANTS THAT THE OPERATION OF THE SOFTWARE WILL BE UNINTERRUPTED OR ERROR-FREE, OR THAT ANY DEFECTS WILL BE CORRECTED. NIST DOES NOT WARRANT OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF THE SOFTWARE OR THE RESULTS THEREOF, INCLUDING BUT NOT LIMITED TO THE CORRECTNESS, ACCURACY, RELIABILITY, OR USEFULNESS OF THE SOFTWARE.

You are solely responsible for determining the appropriateness of using and distributing the software and you assume all risks associated with its use, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and the unavailability or interruption of operation. This software is not intended to be used in any situation where a failure could cause risk of injury or damage to property. The software developed by NIST employees is not subject to copyright protection within the United States.

[ii] Any mention of commercial products within NIST web pages is for information only; it does not imply recommendation or endorsement by NIST.

[iii] This release was tested on a fresh VMWare build of Ubuntu 20.04 LTS which carries several additional system requirements. Prior to running DIMSpec, install or make sure the following are available using:apt install -y build-essential libcurl4-openssl-dev libxml2-dev zlib1g-dev libssl-dev libsodium-dev ffmpeg libtiff-dev libpng-dev libblas-dev liblapack-dev libarpack2-dev gfortran libcairo2-dev libx11-dev libharfbuzz-dev libfribidi-dev libudunits2-dev libgeos-dev libgdal-dev libfftw3-3 libmagick++-devAfter following the R installation instructions for Ubuntu, ensure additional requirements using:apt install -y –no-install-recommends r-cran-tidyverse r-cran-shiny

[iv] Any mention of commercial products within NIST web pages is for information only; it does not imply recommendation or endorsement by NIST.

[v] Requires R.exe is available in your system PATH

[vi] Any mention of commercial products within NIST web pages is for information only; it does not imply recommendation or endorsement by NIST.

[vii] Requires R.exe is available in your system PATH