DIMSpec Quick Guide - Importing Data

Jared M. Ragland*

2023-03-03

Importing Data

Having a database infrastructure is fine. Having one with data is better. The DIMSpec project ships with a database of per- and polyfluorinated alkyl substances (PFAS) for evaluation and use, but one major goal is to be able to easily reuse it. For now, importing data is most easily accomplished by following the data submission and quality assessment workflow established for PFAS. This guide discusses an example import workflow for non-targeted analysis (NTA) of PFAS data using the NIST Non-Targeted Analysis Method Reporting Tool (NTA-MRT) That workflow entails:

1. Complete data collection according to your laboratory procedures.
2. Convert raw data files to the mzML format (see the quick guide for file conversions).
3. Complete your NTA peak characterization protocols, with compound identification and fragment annotation where possible.
4. Use the NTA-MRT to generate import files in javascript object notation (JSON) format, which will include data from step 2.
5. Using files produced in step 4, use the DIMSpec Mass Spectral Quality Control (MSQC) web application [1] to assess quality metrics for new data.
6. Download import files in JSON format from the MSQC application. These will be the files used to import into your database.

The rest of this guide will discuss only the steps using import format established for these files. There are a few aspects to be aware of that will determine import performance and execution. For best results, the following files MUST match your import format:

Creating Import Requirements for Import Validation

The file provided in the project at `/config/NIST_import_requirements.json` is a JSON file defining the elements expected of an import file. It is a list object that includes a lists named for expected elements with their class, internal names, and whether or not the element is required. The entry for the `sample` element includes, for instance

```
{
  "sample": {
    "class":["list"],
    "names":["name","description","sample_class","data_generator","source","starttime"],
    "required":[true]
  }
}
```

*NIST | MML | CSD | jared.ragland@nist.gov

[1] With the compliance file sourced, run `start_app("msqc")` or navigate to a hosted version if available.

By default this file is used in checking files for import for the presence of all required and recommended elements. To use a different import format, develop a similar import requirements object and store it in (or read into) your session as `import_requirements` (though the `full_import` workflow allows specification of which import requirements object to use via the `requirements_obj` argument, which should be the character name of the session object).

Mapping Between Import Files and the Database Schema

The file provided in the project at `/config/map_NTA_MRT.csv` is a comma-separated value file containing elements mapped by name from the NTA-MRT output fields to the database schema (e.g. . If using a different import format, map elements and element components to their corresponding schema tables and columns. To use a different import map, develop a similar map object and store it in (or read into) your session as `IMPORT_MAP` (though the `full_import` workflow allows specification of which import mapping object to use via the `import_map` argument, which should be a `data.frame` session object for your import map). For mapping between the NTA-MRT `sample` element and the database `samples` table, for instance, it looks like the following, where the `import_category` and `import_parameter` columns determine which import elements are mapped to, respectively, which database table (`sql_table`) and column (`sql_column`). Other columns determine other behaviors in certain cases for this import format, for example the `sql_normalization` column should refer to which normalization table to use for a given column, if any, though these values can also be mapped programmatically from the entity relationship map (`er_map`) function. This is a selection of the full import map; for more detail examine the .csv file directly.

import_category	import_parameter	sql_table	sql_parameter	sql_normalization
sample	name	samples	mzml_name	NA
sample	description	samples	description	NA
sample	sample_class	samples	sample_class_id	norm_sample_classes
sample	data_generator	samples	sample_contributor	contributors
sample	source	samples	source_citation	NA
NA	NA	samples	generation_type	norm_generation_type
sample	starttime	samples	generated_on	NA
NA	NA	samples	ms_methods_id	NA
chromatography	ssolvent	samples	sample_solvent	norm_carriers

Schema Map and Data Dictionary

Import routines also leverage the entity relationship map produced by `er_map` which is a list object with one element for each database table detailing its object name, type, which tables it references and the references themselves (if any), which tables it normalizes (if any), and which views it is used within. When calling `full_import` if the object `db_map` (the default session name for the result of `er_map`) does not exist it will be created. This is to facilitate automatic normalization of values, when a value is provided in an import for a foreign key field, the relationship is used to identify the corresponding integer key value. A data dictionary produced by default as object `db_dict` is also available for sessions where the compliance script has been run. This is the result of a call to `data_dictionary` and is stored on disk as a JSON file

Importing Data

The default `full_import` function is the workflow simplification function for imports in this format. Other formats are not supported, but the transparency of the schema should allow relatively easy development of new import routines. This function first checks for missing elements according to the import requirements file via the `verify_import_requirements` function. It allows for certain overrides of missing information via function arguments (i.e. recommended aspects such as “annotation” may be left blank if `include_if_missing_recommended = TRUE`). Files for which missing information is listed as required are excluded; to continue with a batch import when such files exist, set `stop_if_missing_required = FALSE`. Information about this assessment is included in the console when the `full_import` function is called, but the recommended practice is to directly call `verify_import_requirements` ahead of this step and evaluate your import files prior to beginning import. Evaluation is the first step in the import workflow and is a catch for bad import files that may slip through other evaluation steps. The default argument parameters are the strictest, only allowing files with all required and recommended elements to proceed. **It is also considered best practice to create a backup of your database file as a safety check prior to beginning the import process.**

Imports can be done for a single file (the NTA MRT and MSQC outputs will split to one peak per file) or for a batch of files. When the import process starts the checks defined by the import requirements document are performed. Users will receive feedback in the console, and if those checks pass the import will begin. If a data generator is supplied but not identified in the database, users will have the option at the console, to associate that generator with an existing contributor or add a new one; that association will then persist for the rest of the import process. Logs will be generated describing the import progression by default; it is recommended that logging always be turned on for this step.

During the import process itself, each file in the batch will be processed in order of requirements for each SQL nodes. Unnecessary (if any) nodes will be skipped. These are described in the Technical Details > Importing Data section of the DIMSpec User Guide. The following is an example workflow with a bit more detail than that provided in the User Guide. The import routine will automatically verify normalization values and identify key relationships where present, and will prompt at the console if additional information is necessary (e.g. a normalization value is not recognized). Each node has its own custom import function for the NIST NTA MRT format. This format is not intended to be a universal format and therefore, `full_import` is not intended to be a universal import routine.

Your workflow may need to change, or the workflow here adapted, but if using the NTA MRT and MSQC applications provided, this workflow may be perfectly adequate to import data in your use case. Other use cases and import paradigms are of interest to the developers. Eventually, the import process may be simplified through a shiny application, but for now imports are available only from the console. The following example is only that, one example of an import process from the console. Example PFAS JSON files produced by the MSQC app are provided for reference in the `/example` directory.

```r
# Source the compliance file.

source("R/compliance.R")

# Add a file parser to pull files into the session; as source files may be coming from a
# different workflow, this is not included in the project functions.

pull_json <- function(file_list) {
  stopifnot(is.character(file_list), length(file_list) > 0, all(file.exists(file_list)))
  out <- lapply(
    file_list,
    function(x) fromJSON(read_file(x)))
  names(out) <- file_path_sans_ext(basename(file_list))
  return(out)
}

# Obtain a list of files to be imported

src_dir <- here::here("example")
```

```r
f_list <- list.files(path = src_dir, pattern = ".JSON$", full.names = TRUE)

# Convert them to list objects in the session

to_import <- pull_json(f_list)

# Check suitability using the verify_import_requirements function to check against any
# requirements that may have been imposed.

ver <- verify_import_requirements(to_import)
cat(
  sprintf("Required data are %spresent.\n",
          ifelse(all(ver$has_all_required), "", "not ")),
  sprintf("Full detail data are %spresent.\n",
          ifelse(all(ver$has_full_detail), "", "not ")),
  sprintf("Extra data are %spresent.\n",
          ifelse(any(ver$has_extra), "", "not ")),
  sprintf("Mismatched names are %spresent.\n",
          ifelse(any(ver$has_name_mismatches), "", "not "))
)

# We will assume a full rebuild of the database and populate it with PFAS default
# lists of chemicals and normalization values from the /config/data/pfas directory.

# Skip the next two steps to import into an existing database. If making a new
# database it is HIGHLY recommended that you first make a backup copy of any existing
# database prior to performing bulk imports. This is done here by setting archive = TRUE.

build_db(
  populate_with = "populate_pfas.sql",
  archive = TRUE,
  connect = TRUE)
)

# Create an install name for this database - use your institution or project name here.

make_install_code(new_name = "Your organization or project name")

# Here a connection to the database already exists in the session because build_db was
# called with connect = TRUE. If not building a new database, the easiest way to connect is
# to set INIT_CONNECT = TRUE in the global environment settings (at /config/env_glob.txt).
# If a connection does not exist, use the manage_connection function; its arguments will
# by default pull from the session environment which was set during the compliance script.

manage_connection()

# These files all include the NIST PFAS ID as an integer, but for consistency we want
# to instead match the alias held in the database: template is "NISTPFAS00XXXX"

to_import <- lapply(to_import,
  function(x) {
    x$compounddata$id[1] <- x$compounddata$id[1] %>%
      as.character() %>%
      str_remove_all("NISTPFAS") %>%
      as.integer() %>%
      as.character() %>%
      str_pad(6, "left", "0") %>%
      str_c("NISTPFAS", .)
      return(x)
  })

# There may be other naming idiosyncracies from your workflow (e.g. "micro ampere" is
# recognized, but it may be provided with a greek "mu" which may have character
# encoding issues) that should be addressed here.

# Begin import (can test with the first member of the file list (e.g. to_import[1])
# if you like.)

full_import(
  import_object = to_import,
  stop_if_missing_recommended = FALSE,
  include_if_missing_recommended = TRUE
)
```

Last Words

To reiterate, the current version of DIMSpec does not include a graphical user interface for manipulating, importing, or exporting data. It is best to work with the database in such a manner using a database manager program. All functions provided as part of DIMSpec leverage R for database manipulations, including import files. The import routine described here is very much tailored to the project workflow pipeline described in the first section of this quick guide.

Workflows are expected to differ between research groups. The DIMSpec specifications should capture most of the information necessary to support NTA exploration and data mining. The import requirements and mapping files must match your import files exactly. Import functions are fully parameterized to leverage different requirement and mapping files, but care should always be used when importing data.