YAADA

Software Toolkit to Analyze Single-Particle Mass Spectral Data

Reference Manual
Version 1.0

December 31, 2001

Jonathan O. Allen

Chemical & Materials Engineering
Civil & Environmental Engineering
Arizona State University
Tempe, AZ 85287-6006

Abstract

Researchers are now able to measure the size and composition of single aerosol particles using instruments like the Aerosol Time-of-Flight Mass Spectrometry (ATOFMS) instruments developed by Prof. Kimberly Prather and her research group at the University of California. Complete mass spectra are collected on individual particles at a rate of approximately one per second. Thus very large data sets can be collected during a multi-day, multi-instrument experiment. These data sets are too large for *ad hoc* data analysis techniques. YAADA is a package of data management and analysis functions written for Matlab which are designed to process these large data sets. YAADA is available as free software. Users can write Matlab functions to extend YAADA in order to develop novel analyses of ATOFMS data.

YAADA includes functions to import, query, plot, and quantitatively analyze ATOFMS data. The import module rapidly converts data from the common ATOFMS data acquisition software and performs quality control checks on the data. ATOFMS data in YAADA are organized in a hierarchical object-oriented database with objects to uniquely identify each instrument, particle, spectrum, and spectral peak. The search module implements a query language to find sets of particles based on their size and composition. For example, to find particles with aerodynamic diameter (D_a) between 1.0 and 1.8 microns which also have mass spectral peaks near m/z = 23, one would use the query "Da = [1.0 1.8] AND PeakMZ = [22.5 23.5]". The plotting programs include differential concentration versus differential D_a and digital mass spectral plots. The quantification module includes functions to scale up ATOFMS data for missing times, instrument busy time, and particle detection efficiency.

Copyright Notices

YAADA
Software Toolkit to Analyze Single-Particle Mass Spectral Data

YAADA
Software Toolkit to Analyze Single-Particle Mass Spectral Data

Acknowledgements

The development of YAADA has been part of a very rewarding research collaboration between Prof. Cass's and Prather's research groups. This work has been encouraged by Prof. Glen R. Cass at the California Institute of Technology and later at the Georgia Institute of Technology. Prof. Kimberly A. Prather at the University of California, Riverside, and subsequently at the University of California, San Diego, supported the testing and use of YAADA in her research group. Particular thanks are owed to David P. Fergenson, Sylvia H. Pastor, Prakash V. Bhave, and Lara S. Hughes who made important contributions to the development and testing of YAADA. David championed the initial development of YAADA; he contributed to the initial design then tested and debugged the rough beta versions of YAADA. Sylvia systematically tested beta versions and trained her colleagues in its use. Prakash spotted more bugs and made many useful suggestions on the evolution of YAADA. Lara codeveloped and tested the xhatch plotting routines.

Finally, I thank Glen Cass who supported the initial development of YAADA. His energy, vision, and wisdom are sorely missed.

Typographic Conventions

Function and variable names are shown in sanserif font as VariableName. Variable text and text to be entered by the user are shown in slanted sanserif font as *Type me*.

Release Notes for v 1.00a (15 Dec 01)

1. This version will work on databases generated with version 0.93. However to reimport data in version 1.00a, the .inst must be updated for the column name changes listed below.
2. The INST columns ExperimentName, ExperimentDesc, StudyID, and StudyName have been replaced with ExpName and ExpDesc. The column OpCode has been replaced with OpName and OpDesc.
3. The performance of many functions has been significantly improved due to improvements in low-level functions find_chunk and intersect.
4. The performance of run_query for the SPEC and PEAK tables has been greatly improved. The performance of get_children also has been improved.
5. The aggregation operators count, mean, median, sum, min, and max treat spectra with none of the requested peaks as zero.
6. The get_spectrum function can return selected columns from PEAK. The former behavior, returning data from all the default columns, is the default behavior.
7. The function get_int_spectrum replaces get_area_table. The new function is more efficient and offers more control over the data returned. A get_area_table function is provided that calls get_int_spectrum; this function may be discontinued in a future version.
8. The union and intersect functions for the identification objects now can operate on more than two sets of identifiers.
9. A new function combine creates inclusive and exclusive combinations of object sets.
10. A new function xlabel_timedate provides more control over temporal x-axis legends.
11. The msview plot is slightly smaller to display better on XGA screens.
12. The import routines now create chunks of sizes between 1 and 32 MB. The init function requests the chunk size in megabytes.

13. The import routines now create InstIDs for each unique InstCode-OpName-ExpName combination.

14. The check_part function now checks for unique particles only within each chunk; the revised function can be run on large datasets without generating an 'OUT OF MEMORY' error.

15. The D_a to D_p and D_p to D_a conversion programs had truncated trailing NaNs in a vector. This is fixed.

16. Floating point data in the default data tables are stored as double not single since double and single data appear to take the same storage space in Matlab 6.

17. Relative searches like sum(Area{23}) > sum(Area{38.5 40.5}) are not yet implemented.

18. The quantification package, which will include functions to scale up ATOFMS data for missing times, instrument busy time, and particle detection efficiency, is not included in this release.

Contents

List of Figures

List of Tables

1 Introduction

One long-standing goal of atmospheric aerosol science has been to simultaneously measure the size and composition of individual airborne particles. Prof. Kimberly Prather and her research group developed the aerosol time-of-flight mass spectrometry (ATOFMS) instruments which measure the size and composition of individual airborne particles [1, 2, 3]. In 1996 the Prather group and Prof. Glen Cass's group at the California Institute of Technology conducted the first field study with the ATOFMS instruments and colocated reference samplers [4, 5, 6, 7]. During this study, three ATOFMS instruments measured the size of approximately 3×10^6 particles and the composition of 3×10^5 particles. YAADA was developed to manage and analyze this and other large data sets as part of the research collaboration between the Prather and Cass groups.

ATOFMS data in YAADA are organized in a hierarchical object-oriented database with objects to uniquely identify each instrument, particle, spectrum, and spectral peak. The YAADA software includes functions to import, query, plot, and quantitatively analyze these data. YAADA is written in the Matlab programming language and is available as free software. Users can extend and automate analyses in YAADA by modifying or creating functions in the Matlab programming language. YAADA is provided as free software by the California Institute of Technology and Arizona Board of Regents. Users are requested to cite [8] in works for which YAADA was used.

This manual documents the data organization, data management functions, and data analysis functions in YAADA. Some familiarity with Matlab is assumed. Matlab is available from the Mathworks (Natick, MA, www.mathworks.com). In addition to the complete Matlab documentation, excellent third-party texts on Matlab are available [9, 10].

1.1 ATOFMS Instrument Operation

ATOFMS instruments operate by sampling aerosol at atmospheric pressure and directing the flow through an expansion nozzle and skimmers (see Figure 1) [1, 2]. During the expansion, particles are accelerated to a velocity characteristic of their aerodynamic size, with the smallest particles traveling at the highest speeds. After the last skimmer, velocities (hence aerodynamic size) of individual particles are measured in the sizing chamber by detecting scattered light from two timing lasers positioned a known distance apart.

The rarefied aerosol is subsequently directed into the particle ablation/ionization chamber of the ATOFMS instrument. The arrival time of a specific particle is predicted based on the velocity measured in the sizing chamber and an ablation/ionization laser is fired to intercept the

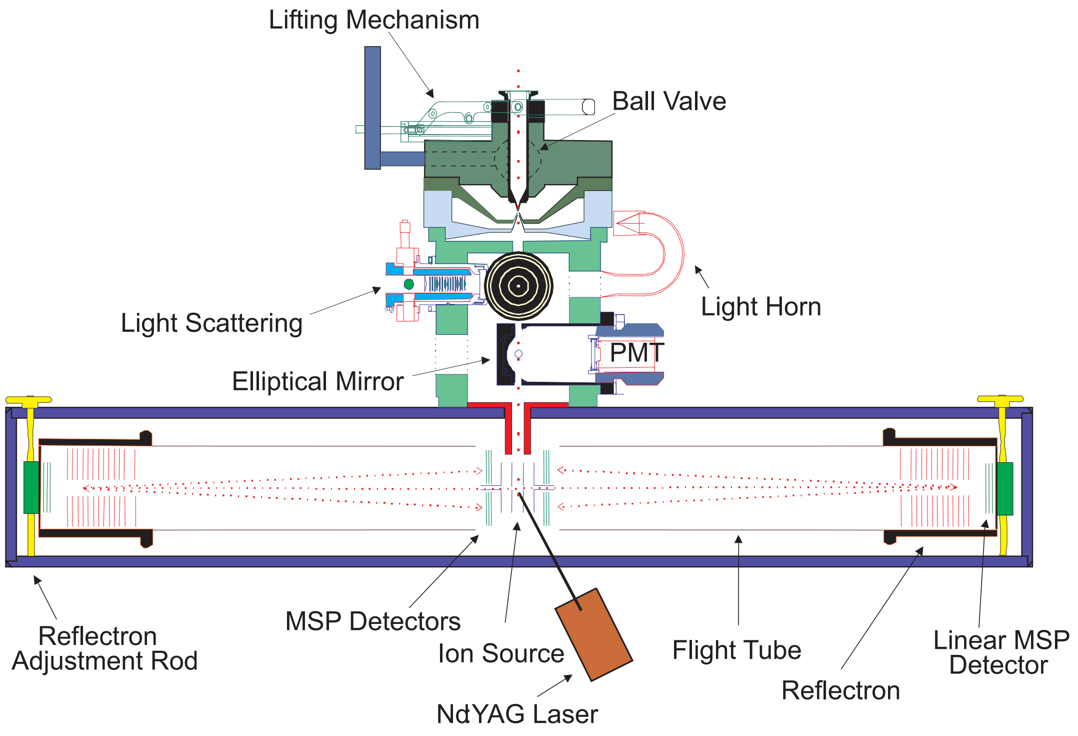


Figure 1: Schematic diagram of Aerosol Time-of-Flight Mass Spectrometry instrument developed by Prof. Prather and her research group [1].

moving particle. Ionized fragments from the particle are directed to positive and negative polarity time-of-flight mass spectrometers. The particle size and, if present, mass spectra are then recorded on a data acquisition computer. Particles which are detected by both timing lasers are said to have been *sized*; those which are also ablated and ionized by the third laser to produce mass spectra are said to have been *hit*. The ATOFMS instruments detect approximately 2 particles per second and hit approximately 15% of these particles. Thus, an ATOFMS instrument can collect size data on 200,000 particles and composition data on 25,000 of these particles in 24 hours. The total quantity of data collected during a multi-day, multi-instrument experiment is therefore too large for *ad hoc* data analysis techniques. YAADA provides a framework and tools to analyze these large datasets.

ATOFMS instruments collect data on time of acquisition, delay between light scattering events, and mass spectrometer signal versus time data for both the positive and negative ions. These raw data are analyzed and converted into terminal velocities and mass spectral peak data by proprietary software specific to the instrument hardware. The resulting data is referred to as *preprocessed*. These preprocessed data are stored as text files. YAADA includes programs to convert some preprocessed data file formats into the standard PK2 format.

2 Getting Started Tutorial

This section describes how to start using YAADA with a small set of demonstration data. A brief tutorial follows which demonstrates how to search for particles with characteristic mass spectra and plot data for these particles.

In order to run YAADA, you need Matlab (version 5.3 or later) and Perl (version 5.6 or later) programs installed on your computer. Details are available in Appendix A. This section assumes that you have a basic familiarity with Matlab, including the concepts of scripts, functions, and variables which are explained in the *Getting Started with Matlab* manual. In addition to the complete Matlab documentation, excellent third-party texts on Matlab are available [9, 10]. No familiarity with Perl is needed to run YAADA.

2.1 Import Data

To begin, start Matlab then move to the directory where YAADA was installed in your computer with the `cd` command.

Approximately 15 minutes of demonstration data provided by the Prather group are distributed with YAADA. The demonstration data are in three text files in the *PK2* format in the `demo/pk2` directory. Note that these data are not representative of actual ambient aerosol measurements and are provided for demonstration only.

There are four steps to create a new database and import data into it, they are

1. Convert data generated by an ATOFMS instrument into PK2 files
2. Set up the database structure
3. Convert data from PK2 files to the YAADA data format
4. Set up the Matlab workspace

The first step, creation of PK2 files, has been done for the demonstration data. Each of the remaining steps are automated in programs. The `init` program sets up the database structure; this function will request information about where files are located with the prompts

```
Study name? [demo]
Main YAADA directory? [c:/yaada/v100a]
Perl program directory? [c:/perl/bin]
User program directory? [c:/yaada/v100a/user]
```

```
Processed data directory? [c:/yaada/v100a/demo/final]
Temporary file directory? [c:/temp]
Clear database in c:/yaada_100a/demo/final? (y/n)
Chunk size (MB)? [5.000000]
```

For each prompt, enter a value or return to accept the default value shown in square brackets. You should create the requested directories before running init. Now run init for the demo study; be sure to clear the database and set Chunk Size to 5.

Next use the make_demo program to convert data from PK2 files to the YAADA internal data format. This will take a few minutes, during which YAADA will display messages to mark progress in loading and verifying the data; for example

```
Digesting c:\yaada\demo\pk2\demo1.pk2
Digesting c:\yaada\demo\pk2\demo2.pk2
Digesting c:\yaada\demo\pk2\demo3.pk2
Loading c:\yaada\demo\final\P000001.mat
Loading c:\yaada\demo\final\S000001.mat
Loading c:\yaada\demo\final\K000001.mat
CHUNK_CHECK for CL_PART finished

CHUNK_CHECK for CL_SPEC finished

CHUNK_CHECK for CL_PEAK finished

Found 2963 unique PartID in chunk P000001, 0 duplicates
Found 744 unique SpecID in chunk S000001, 0 duplicates
Found 91109 unique PeakID in chunk K000001, 0 duplicates
Found 2958 unique physical particles in database
List of 5 duplicates written to c:\temp\check_part.log
A small number (<1%) of duplicates in c:\temp\check_part.log is normal
```

Next run the startup script to set up the Matlab workspace for YAADA as

```
>> startup
— Copyright Display —
Study name? demo
```

Startup loads the first data files and reports

```
Loading d:\yaada\demo\final\P000001.mat
Loading d:\yaada\demo\final\S000001.mat
Loading d:\yaada\demo\final\K000001.mat
```

Congratulations, you have created the demonstration database! List the files in your study directory with the command

```
dir(YAAADA.StudyDir)
```

The study directory should contain files like these:

```
    .             P000001.mat    datadef.mat
    ..            S000001.mat    inst.mat
    K000001.mat   chunklist.mat  pk2list.mat
```

Files starting with K contain *chunks* of mass spectral peak data. YAADA stores large data tables as a group of chunks; the demo data set is small enough that all the peak data fit in a single chunk. Files starting with P and S contain chunks of particle and spectral data, respectively. Information about these chunks is stored in chunklist. The remaining files (datadef, inst, pk2list, and yaada) store the database structure and instrument data.

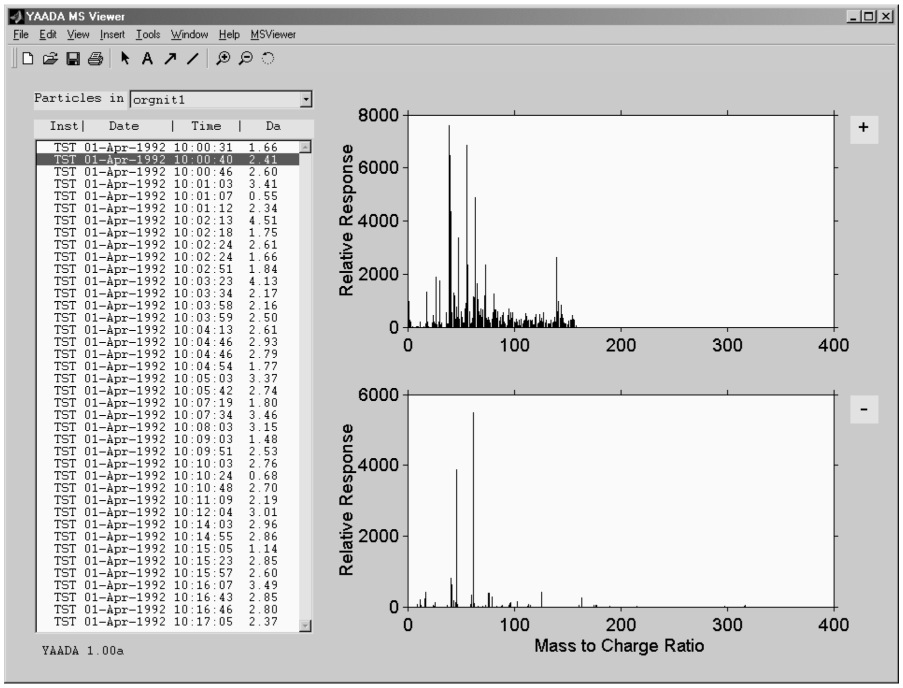
After you have successfully created the demonstration database, you can quit Matlab at any time with

>> *quit*

When you restart Matlab, move to the main YAADA directory then run the startup script to access the demonstration database again.

2.2 Find Particle Types

Once you have created a YAADA database, you can search to find particles based on their size and composition. In this section we demonstrate YAADA's query functions and find sets of particles based on their composition using the criteria developed by Noble and Prather [2]. Noble and Prather call the most common particle type for Southern California "Organic/Nitrate". These are most likely particles originally emitted by combustion sources on which ammonium nitrate has condensed. These particles have peaks at m/z equal to 12 (C^+), 18 (NH_4^+), 24 (C_2^+), 30 (NO^+), and 36 (C_3^+). To find this type of particle in the demonstration data use (*enter the command on one line*):



```
Inst|    Date     |   Time   |  Da
TST 01-Apr-1992 10:00:31  1.66
TST 01-Apr-1992 10:00:40  2.41
TST 01-Apr-1992 10:00:46  2.60
TST 01-Apr-1992 10:01:03  3.41
TST 01-Apr-1992 10:01:07  0.55
TST 01-Apr-1992 10:01:12  2.34
TST 01-Apr-1992 10:02:13  4.51
TST 01-Apr-1992 10:02:18  1.75
TST 01-Apr-1992 10:02:24  2.61
TST 01-Apr-1992 10:02:24  1.66
TST 01-Apr-1992 10:02:51  1.84
TST 01-Apr-1992 10:03:23  4.13
TST 01-Apr-1992 10:03:34  2.17
TST 01-Apr-1992 10:03:58  2.16
TST 01-Apr-1992 10:03:59  2.50
TST 01-Apr-1992 10:04:13  2.61
TST 01-Apr-1992 10:04:46  2.93
TST 01-Apr-1992 10:04:46  2.79
TST 01-Apr-1992 10:04:54  1.77
TST 01-Apr-1992 10:05:03  3.37
TST 01-Apr-1992 10:05:42  2.74
TST 01-Apr-1992 10:07:19  1.80
TST 01-Apr-1992 10:07:34  3.46
TST 01-Apr-1992 10:08:03  3.15
TST 01-Apr-1992 10:09:03  1.48
TST 01-Apr-1992 10:09:51  2.53
TST 01-Apr-1992 10:10:03  2.76
TST 01-Apr-1992 10:10:24  0.68
TST 01-Apr-1992 10:10:48  2.70
TST 01-Apr-1992 10:11:09  2.19
TST 01-Apr-1992 10:12:04  3.01
TST 01-Apr-1992 10:14:03  2.96
TST 01-Apr-1992 10:14:55  2.86
TST 01-Apr-1992 10:15:05  1.14
TST 01-Apr-1992 10:15:23  2.85
TST 01-Apr-1992 10:15:57  2.60
TST 01-Apr-1992 10:16:07  3.49
TST 01-Apr-1992 10:16:43  2.85
TST 01-Apr-1992 10:16:46  2.80
TST 01-Apr-1992 10:17:05  2.37
YAADA 1.00a
```

Figure 2: Mass Spectral Viewer showing "Organic/Nitrate" particles.

```
>> OrganicNitrate1 = run_query('mz = 12 and mz = 24 and mz = 36 and
                                        mz = 18 and mz = 30');
```

YAADA will find 40 particles that match this criterion. You can view the particles and their mass spectra with the msview program. Note that msview requires a screen resolution of at least 1024 by 768 pixels. To run msview

```
>> msview
```

The sets of particles in the Matlab workspace are shown in a pull-down menu in the upper left corner of the YAADA MS Viewer figure. Select OrganicNitrate and the particles will be listed in the left-hand column. The particles are listed in the left-hand column ordered by the three letter instrument code, date, time, and aerodynamic diameter in μm. Click on one of these particles to view its mass spectra (see Figure 2). The mass spectra are shown as lines at integral mass-to-charge ratios. Some particles have both positive and negative mass spectra, others have only one mass spectrum.

You can zoom and annotate spectra using the Matlab figure tools. To activate these select from the menu View/Figure Toolbar. Click on the tool of interest to edit the spectra. To zoom in on portions of the spectra, select the magnifier tool then drag the cursor over the area of interest. To return to the original magnification double click on the spectra.

Search criteria can also specify the mass spectral response, “area”, for each peak. A more complex search for Organic particles is

```
>> OrganicNitrate2 = run_query('Area{12} > 50 and Area{24} > 75 and
                                Area{36} > 50 and Area{18} > 50 and
                                Area{30} > 75');
```

Here Area{12} > 200 finds those spectra which have peaks in the mass-to-charge range 11.5 to 12.5 Daltons that have a total response greater than 200. Only 12 particles in the demonstration data set match this criterion. To view these with msview select from the menu MSViewer/Refresh PartID List, then select OrganicNitrate2.

Another common class of compounds identified by Noble and Prather is the “Marine” particle class identified by ion peaks at $m/z = 23$, 39, 81, and 83 Da, representing Na^+, K^+, $Na_2{}^{35}Cl^+$ and $Na_2{}^{37}Cl^+$, respectively. Search for these with the command

```
>> Marine = run_query('Area{23} > 1000 and Area{39} > 200 and
                       Area{81} > 50 and Area{83} > 50');
```

This demonstration of searching in YAADA used only the basic query syntax and a small part of the data available in YAADA; these are discussed in detail in Sections 4 and 6.

2.3 Use Identifier Objects

YAADA creates unique identifier objects for instruments, particles, spectra, and peaks. Sets of particles, for example, are identified by partid objects. These objects are the basis for data management and analysis in YAADA; using them you can create novel and sophisticated data analysis procedures. Here is a brief introduction to partid objects.

Identifier objects behave similarly to numeric variables in Matlab. To display an object, type its name at the Matlab prompt without a trailing semicolon; for example display the OrganicNitrate2 partid objects as

```
>> OrganicNitrate2
00001 00000 00118
00001 00000 00356
00001 00000 00383
00001 00000 00588
00001 00000 00831
00001 00000 00992
```

```
00001 00000 01729
00001 00000 01817
00001 00000 02329
00001 00000 02475
00001 00000 02648
00001 00000 02813
```

The partid objects are displayed as 3 integers. Note arbitrary unique numbers are assigned as identifiers by YAADA; the numbers may not exactly match those in your demonstration data set. Subsets of partid objects can be selected using indices. For example display the first two partid in OrganicNitrate2

```
>> OrganicNitrate2(1:2)
00001 00000 00118
00001 00000 00356
```

Sets of identifier objects can be combined using the Matlab set functions. To find particle that are *either* marine or organic/nitrate, combine the Marine and OrganicNitrate2 sets of particles

```
>> MarineOrOrganicNitrate = union(Marine,OrganicNitrate2);
```

To find particle that are marine particles with an organic/nitrate coating, find particles with *both* marine and organic/nitrate signatures as

```
>> MixedMarine = intersect(Marine,OrganicNitrate2);
```

For these small sets you can verify the logical operations by displaying the partids.

2.4 Plot Data

A few basic plotting programs are included in this version of YAADA. It is expected that users will develop (and contribute) their own plots using the Matlab plotting functions and Handle Graphics. Examples of publication quality plots generated with Matlab can be found in [6, 7].

A useful overview of an ATOFMS data set is obtained by plotting the frequency of hit and missed particles. To make this plot for the demonstration period (*comments follow "%"; you do not need to type these*).

```
>> figure;   % open a new figure
```

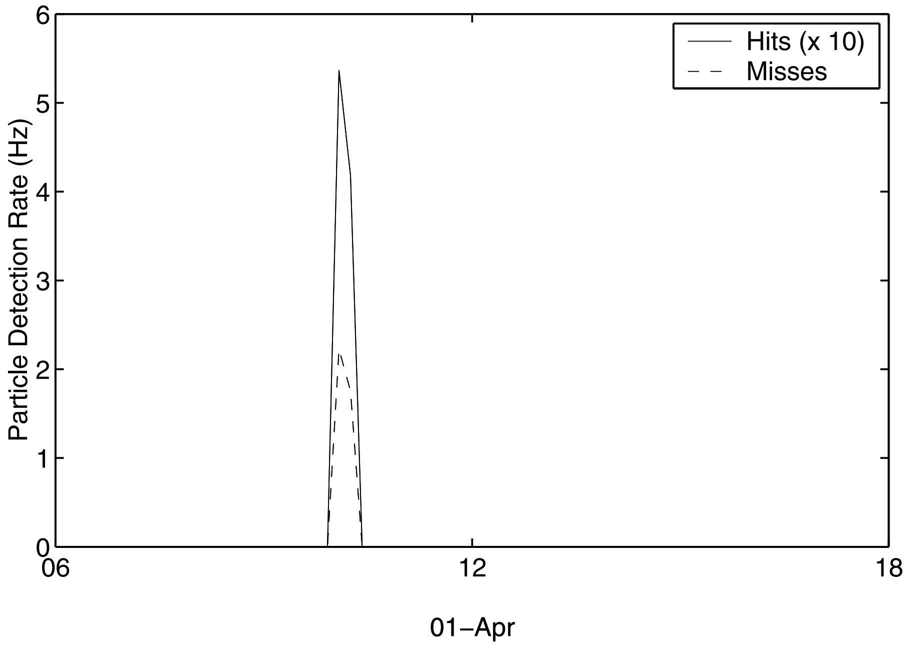


Figure 3: Detection Rate for Hit and Missed Particles.

```
>> clf;      % clear the new figure
>> plot_hit_miss('TST', 01-Apr-92 06:00, 01-Apr-92 18:00, 144)
```

This plots the frequency of hit and missed particles for instrument TST during April 1, 1992. The first date is the start of the period, the second is the end of the period. The last parameter, 144, is the number of bins into which the time period is divided; in this case, 5 minute intervals.

The hit and missed particle plot shows that the demonstration data cover only a short time in the morning (see Figure 3). The detection rate of missed particles is approximately 2 Hz and 25% of the particles are hit. A similar plot for a larger data set would reveal particle concentrations and instrument performance.

To analyze many mass spectra, it is useful to aggregate the mass spectra and plot the aggregate. The digital mass spectrum aggregates mass spectra by collecting for each integral m/z value the fraction of spectra that have a peak [11]. Plot digital mass spectrum of the Marine particles to see the aggregate mass spectrum as

```
>> figure;   % open new figure
>> clf;      % clear figure
>> digital_ms(Marine,1,200);
```

The peaks in the selection criterion, $m/z = 23$, 39, 81, and 83 Da, are present in nearly all the

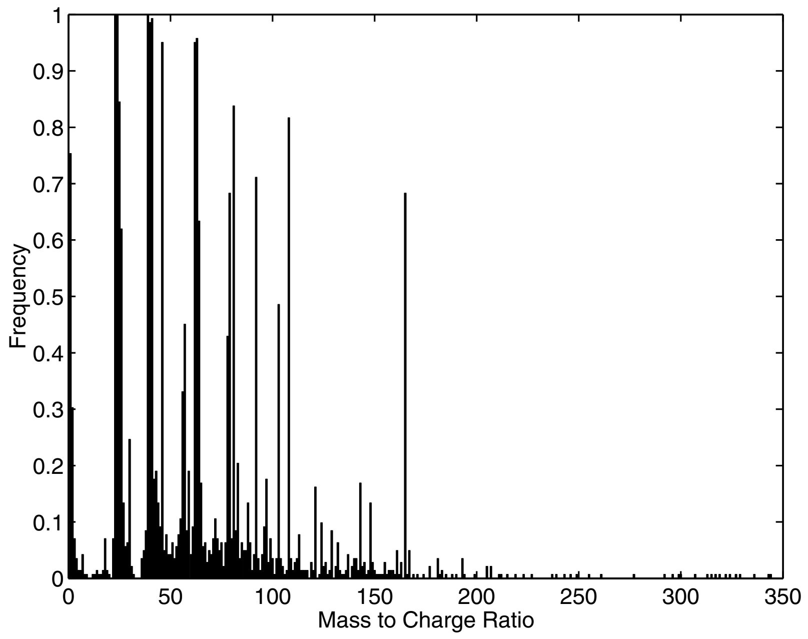


Figure 4: Digital Mass Spectrum of "Marine" Particles

spectra as expected. The digital mass spectrum shows that many of the spectra also contain distinctive peaks, for example $Na_2NO_3^+$ (m/z = 108) and $Na_3SO_4^+$ (m/z = 165).

This concludes the tutorial. The remaining chapters of this manual cover Data Objects, Database Structure, Importing Data, and the Query Language. You should skim the chapters on Data Objects and Database Structure since a basic understanding of these is need to effectively use YAADA. You can skip the Importing Data chapter if you are working with an existing database. The Query Language chapter is essential for the effective use of YAADA.

3 Data Objects

3.1 Identifier Object Classes

The identifier objects, `instid`, `partid`, `specid`, and `peakid`, uniquely identify elements of the ATOFMS data. These objects have an inherent hierarchy which establishes relations between the data elements. The identifier objects are composed of fields as shown in Table 1.

A single identifier object contains identifiers for one or more thing, i.e. a partid object generally refers to a *set* of particles. We have already seen how `run_query` returns a set of `partids` that match a search criterion. Identifier objects can also have null or empty values. Null objects have fields set to 0; these should be used as place holders. Empty objects do not

Table 1: Identifier Objects

Object	Parent	Field
instid	—	instrument serial number (0–65535)
partid	instid	particle serial number (0–4,294,967,295)
specid	partid	polarity (0–1)
peakid	specid	peak serial number (0–65535)

Table 2: Column Object Structure

column.name	name of column data
.desc	description of column data
.units	units of column data
.type	type of data
.sorted	true if data are sorted
.makeindex	true if data are to be indexed
.renewindex	true if index is out of date
.data	column data
.index	index to sorted column data

contain any data.

YAADA includes a suite of functions to create and manipulate the identifier objects; these functions are called *methods* (see Section B.1). Objects lower in the hierarchy, called *children*, inherit information from their parent object, e.g. each partid inherits its parent's instid. Identifier object methods can be called with different object types, in this case, child objects are *promoted* to match their parent object. Thus, if a set of particles and peaks are intersected, the peakids are promoted to their parent partids, then the two sets of partids are intersected. This intersection would return the set of particles from the original particle set which also contain a peak in the peak set.

3.2 Column Object Class

Column objects contain one type of data, for example the aerodynamic diameter a particle. We have already used column names like Da and Area to create search criteria. The column object contains its own description and optional sorted index (see Table 2). Data stored in a column can be of one of the types listed in Table 3. Time data are stored in the Matlab date format in which dates are the number of days since 01-Jan-0000 (try *help datenum* for more explanation). Integer date numbers correspond to midnight on that day. User-written objects, including the identifier objects, are also valid data types.

Table 3: Column Data Types

Data Type	Description
time	days since 01 Jan 0000, stored as double
single	single precision floating point number
double	double precision floating point number
boolean	true (1) or false (0)
word	character string without spaces, stored as a cell vector
text	character string with spaces, stored as a cell vector
cell	Matlab cell

Table 4: Data Tables

DATADEF	Columns that make up the database structure
INST	Data on instrument sampling conditions
PART	Data on particles
SPEC	Data on spectra
PEAK	Data on peaks which make up spectra
CL_PART	Chunk list for PART virtual table
CL_SPEC	Chunk list for SPEC virtual table
CL_PEAK	Chunk list for PEAK virtual table

3.3 Table Object Class

Data in YAADA are stored in *tables* which are organized into *columns* and *rows*. Each column corresponds to a type of data, e.g. the aerodynamic diameter of particles; each row corresponds to a single element in the database, e.g. a particle. Tables contain a column which uniquely identifies each row; this is called a *primary key*. The primary keys are a convenient way to extract data from tables. For the default data structure, the identifier objects are primary keys of the respective tables.

4 Database Structure

Data in YAADA are stored as table objects. The default data structure contains eight tables which are created by DATA_DEF and filled with imported data (see Table 4). These tables are stored *global* variables, that is the information in these variables is available to any program.

Table 5: Data Definition Table

Column	Type	Description
Table	Word	Name of table
Column	Word	Name of column
Desc	Text	Description of column data
Units	Text	Units of column data
Type	Word	Type of data
Sorted	Boolean	True if column is sorted
MakeIndex	Boolean	True if column is indexed
PrimaryKey	Boolean	True if column is table's primary key

4.1 Data Definition Table (DATADEF)

A description of the database structure is stored in the DATADEF table. This table is saved in the datadef.mat file in the study directory. Each study must have exactly one data definition. The data definition can only be altered before the study data files are built.

4.2 Instrument Table (INST)

Instrument data describe the instrument conditions during a study or portion of a study (see Table 6). Unique instrument operating conditions are assigned unique identifiers, instid. Physical instruments are designated with an InstCode, a code of three uppercase letters; only letters (A–Z) are valid. Required columns are shown in bold. Columns generated by YAADA are shown in bold italics. DaCalibFunction names a function to calculate aerodynamic diameter (D_a) in μm as a function of velocity (v) in m/s. The calibration functions distributed with YAADA are described in Section B.4.1. The BusyTimeFunction names a function to calculate instrument busy time, and hence actual on-line time (see Section B.10.1).

4.3 Particle Table (PART)

Many particles are detected by an instrument in an experiment, thus there is a many-to-one relationship between particles and the instrument. The particle table includes data on all sized particles, both hit and missed particles. Data on particles include their detection times and terminal velocities (see Table 7). Required columns are shown in bold. Particles are uniquely identified by partids which are assigned during data import. The maximum number of particles for an instrument and operation mode is approximately 4×10^9 ($2^{32} - 1$).

Table 6: Default Instrument Table.

Column	Type	Description
InstID	Object	Identifier for instrument operating condition
InstCode	Word	Three uppercase letter code for instrument
InstName	Word	Instrument name
InstDesc	Text	Instrument description. Include version, important adjustments or changes to instrument
OpName	Word	Name of instrument operating condition
OpDesc	Text	Description of instrument operating condition
ExpName	Word	Experiment name
ExpDesc	Text	Experiment description
AvgLaserPower	Double	Average ionization laser power
DaCalibFunction	Word	Function to calculate D_a from velocity
DaCalibParam	Cell	Parameters for D_a calibration function
BusyTimeFunction	Word	Function to calculate time to process particle data
BusyTimeParam	Cell	Parameters for busy time function
SampleFlow	Double	Sample flow rate (m^3s^{-1})
MinHeight	Double	Minimum signal height for peak identification
MinArea	Double	Minimum signal area for peak identification
PosDefaultZero	Double	Default zero level for positive mass spectra
NegDefaultZero	Double	Default zero level for negative mass spectra
PosDefaultVoltage	Double	Default voltage level for positive mass spectra
NegDefaultVoltage	Double	Default voltage level for negative mass spectra
PreprocDate	Time	Date of preprocessing
PreprocDesc	Text	Description of data preprocessing
LastPartID	Object	Last PartID written for instrument

Table 7: Default Particle Table

Column	Type	Description
PartID	Object	Particle identifier
Time	Time	Particle detection time
Velocity	Double	Particle velocity (m s^{-1})
Da	Double	Particle aerodynamic diameter (μm)
PositionInFolder	Double	Order in which particle spectrum saved
FastScatter	Boolean	True particle detected in fast scatter mode
Hit	Boolean	True if particle has an associated mass spectrum

Table 8: Default Spectrum Table

Column	Type	Description
SpecID	Object	Spectrum identifier
Polarity	Boolean	True if positive spectrum
FileNameLength	Double	Length of file name in which spectra originally stored
AreaIntegral	Double	Area of all peaks in this spectrum
Noise	Double	Average noise of spectrum
BaseLine	Double	Baseline for TOFMS signal
FitVoltage	Double	Voltage level for TOFMS
FitZero	Double	Zero level for TOFMS

Table 9: Default Peak Table

Column	Type	Description
PeakID	Object	Peak identifier
MZ	Double	Mass to charge ratio at peak (Daltons)
Area	Double	Area of peak (arbitrary units)
RelArea	Double	Relative area of peak
Height	Double	Height of peak (arbitrary units)
Width	Double	Width of peak at half height (arbitrary units)
BlowScale	Boolean	True if peak exceed instrument dynamic range

4.4 Spectrum Table (SPEC)

Mass spectra are recorded for the *hit* particles. ATOFMS instruments can record both positive and negative spectra for particles, thus a particle can have 0, 1 or 2 associated spectra. Spectra are uniquely identified by specids. The spectrum table stores data that apply to a whole mass spectrum (see Table 8). Required columns are shown in bold. These include how the data were originally stored (FileNameLength), overall spectral data (AreaIntegral, Noise, BaseLine), and how the spectral data were processed (FitVoltage, FitZero).

4.5 Peak Table (PEAK)

Each mass spectrum is composed of many peaks. The peak data include the mass/charge ratio, area, width, and height (see Table 9). Required columns are shown in bold. Peaks are uniquely identified by peakids which are assigned during data import. The maximum number of peaks that can be assigned in a spectrum is 65535.

4.6 Virtual Tables and Chunk Lists

Matlab is very efficient at processing objects that are stored in physical memory. However, Matlab was not designed to store objects which are larger than a computer's physical memory, and cannot handle large data sets. Once physical memory is full Matlab uses virtual memory, also known as disk swap space, and its performance deteriorates dramatically. Once physical and virtual memory are full, Matlab aborts with an `OUT OF MEMORY` error.

YAADA stores large data tables as *virtual tables* which can be as large as a computer's available disk space. Virtual tables are split into *chunks* which are stored as separate files and loaded into memory as needed. Only one chunk at a time is loaded for each virtual table. Like ordinary tables, virtual tables are sorted on their primary key, e.g. particle identifiers. A chunk must contain *contiguous* rows of a virtual table so that YAADA can locate and load the chunk that contains data for a specific primary key. Since identifier objects for the same instrument are assigned in order of acquisition time, the virtual tables and chunks are also sorted by time.

During normal use of YAADA, you need not worry about chunks and virtual tables since functions like `run_query`, `get_column`, and `get_spectrum` load the necessary chunks as needed. However, if you want to write programs that access chunks directly, you need to understand the rest of this section.

YAADA contains three virtual tables `PART`, `SPEC`, and `PEAK`, each of which is typically 1 GB or larger for a single study. The global variables `PART`, `SPEC`, and `PEAK` contain *only* the current chunk of the larger virtual table. Thus a command like `idx = find(PART(:,'Da') > 2)` finds only particles larger than 2 μm in the *current* chunk. To find particles in the entire virtual table use the `run_query` function discussed below.

Functions like `run_query` are optimized to load only chunks as needed. Programs that access chunks directly should first call *find_chunk* to get a list of chunks that contain certain identifier objects, or cover a time period. Next these functions should load each chunk, in turn with `load_chunk` and process each chunk attempting to minimize the reloading of chunks.

Information about the chunks is stored in *chunk lists* tables. `CL_PART`, `CL_SPEC`, and `CL_PEAK` are the chunk lists for their respective virtual tables. Chunk lists record the range of identifier objects and acquisition times stored in each chunk (see Table 10).

4.7 Changing the Database Structure

This section has described the default YAADA database structure in detail. One can modify the content of these tables by editing the `data_def` program before the study data files are built. In this program, columns are defined with code that fills cells with information about each

Table 10: Chunk List Structure

Column	Description
First	First identifier object in chunk
Last	Last identifier object in chunk
Start	Acquisition time of first identifier object in chunk
Stop	Acquisition time of last identifier object in chunk
ChunkName	Name of chunk file in study directory

column like

```
i = i + 1;
Table{i} = 'PART';
Column{i}  = 'Da';
Desc{i}  = 'Particle Aerodynamic Diameter';
Units{i} = 'um';
Type{i}  = 'Double';
Sorted{i} = 0;
MakeIndex(i) = 1;
PrimaryKey{i} = 0;
```

New columns can be added to the existing tables as needed. The YAADA import and query packages expect some columns in the database; these are shown in bold in Tables 6, 7, 8, and 9. Changes to or removal of these columns will require reprogramming of portions of these packages. New tables can be added to store user data using the table class creation methods. For example, the quant package creates new tables to store busy time data.

5 Data Import

This chapter describes the creation of a YAADA database from text files created by ATOFMS instrument data acquisition software. If you have been given data in the PK2 format, skip to Section 5.2. You can skip this chapter if you have been given a YAADA database stored in Matlab (*.mat) data files.

Users can import data into YAADA from a number of text file formats. These preliminary steps are common to all input data file formats:

1. Create an empty directory where the database will be stored; make subdirectories for the preprocessed, PK2, and final data files.
2. Initialize the database using the `init` script.
3. Create the PK2 files from the preprocessed data.
4. Create the final data from the PK2 files.

The last two steps can be automated using a program modeled on the `make_demo` script.

5.1 PK2 Creation

ATOFMS data are initially recorded by a data acquisition program. These raw data are then analyzed to identify mass spectral peaks and otherwise process the data. There are a number of data acquisition and preprocessing programs that are specific to ATOFMS instrument designs. These programs write data in a number of different formats. The data acquisition and preprocessing programs are not part of YAADA.

The preprocessed data are converted to the PK2 format using Matlab and Perl programs included in YAADA. The PK2 format is a flexible data format from which YAADA reads data. This file format is designed as a human-readable archival format which includes metadata to describe the data in the file. The PK2 format is defined in Appendix C.2.

YAADA can create PK2 formatted data files from data files created by

- TasWare (1997 and 2000 versions), a data preprocesing program developed by Tas Dienes
- TSI data preprocesing programs

These data formats are briefly described below.

5.1.1 TasWare

TasWare is a data preprocesing program developed by Tas Dienes in Prof. Kim Prather's group at the University of California, Riverside. Hit particle data and their spectra are recorded in files with the extension `pkl`. Missed particle data are recorded in file with the extensions `sem` and `sef`. The formats of these files are described in `pkl2pk2.pl`.

The TasWare data files do not record instrument operating conditions, so the first step to import TasWare data into YAADA is to create instrument data files. For each `*.pkl` file or directory containing `*.pkl` files, create a text file with the extension `.inst` to store instrument data. These files have comment lines and data lines (see Section C.1). The comment lines begin with %. There is one data line for each instrument data column in the form *ColumnName = Value*. Note that InstID values are assigned by YAADA during import.

Raw data directories can contain one instrument file for the entire directory, or one instrument file for each PKL file. If there is one instrument file in a directory, these data are copied to every PK2 file and the base file name is arbitrary. If there one instrument file for each .pkl, the base file name must match the PKL base file name.

Next, create a program to digest the raw data files with a program modeled after the make_demo script in the main directory. Modify the program as described in the comments. This program calls digest_tw97 or digest_tw00 which reads TasWare data files and writes PK2 files. It is convenient to keep TasWare files segregated in subdirectories by experiment. Digest_* programs halt if an error is found in the raw data files. In this case fix the errors, and restart the make_*study* program starting with the directory which contained the bad file. Sometimes make_*study* cannot be restarted, then redigest the data from the beginning by running startup followed by make_*study*.

5.1.2 TSI Data Acquisition Software

Similar to TasWare. Use the program digest_tsi00 to digest the preprocessed data.

5.2 PK2 Digestion

The last step in database creation is to digest PK2 format data files. Create a program modeled after the make_demo script in the main directory. Modify the program as described in the comments and run the program. The make_*study* program creates data table chunks from PK2 data files where *Pk2Dir* is the directory containing PK2 files. All files in *Pk2Dir* and its subdirectories will be processed.

It is common for a few PK2 files to have errors which prevent incorporation of the data. Digest_pk2 is designed so that these errors do not require that the entire database be redigested. Digest_pk2 saves its important data to a file after each PK2 file is successfully digested. If an error, typically an error during file loading, occurs, the user can fix the problematic PK2 file and start the digestion process where it left off by calling digest_pk2 again. To discard intermediate results and restart the digestion process, reinitialize the database with the init script, then rerun make_*study*.

Once all the PK2 files are digested, run check_all to update and check the data. The updates include calculation of the Da, Hit, AreaIntegral, and RelArea columns. The data checks verify that identifiers are unique, chunks are contiguous, and that physical particles are unique in the entire database. “Physical particles” are those with a unique combination of InstCode—Time—Velocity. A small number of duplicate InstCode—Time—Velocity combinations are expected since time and velocity data are discretized. Duplicate combinations of InstCode—Time—Velocity can be ignored if they are not continuous and are less than 1% of the particles in the database. Duplicate particle information is written to check_part.log in the temporary directory.

6 Query Language

6.1 Query Elements

You can find sets of instruments, particles, spectra, or peaks which match search criteria with YAADA. The columns to search, search conditions, and combinations are written in the YAADA query language. The elements of the language are *column names*, *aggregation operators*, *relative operators*, *values*, and *set operators* which users combine to define a query. An elementary query is a column name, relative operator, and value. Every query must contain at least one elementary query. For example, to find particles with aerodynamic diameter (D_a) less than 1.0 μm, use the elementary query

```
Da < 1.0
```

where Da is the column name, < is the relative operator, and 1.0 is the value. The results of elementary queries can be combined using set operators and parentheses. For example, to find particles with D_a less than 1.0 μm which also have mass spectral peaks near 23, use the query

```
Da < 1.0 and MZ = [22.5 23.5]
```

The remainder of this section discusses the query elements in detail.

6.1.1 Column Names

Most of the columns in the database are searchable directly. The exceptions are text, cell, and object columns which are *not* searchable. Columns with single, double, and time type data can be queried for relative and range matches. NaN, “not a number”, values in numeric columns are ignored during query execution. Columns with word and logical data can be queried for exact matches.

Table 11: Aggregation Operators

Operator	Description
Count	Number of rows matching m/z condition
Mean	Mean of rows matching m/z condition
Median	Median of rows matching m/z condition
Sum	Sum of rows matching m/z condition
Min	Minimum of rows matching m/z condition
Max	Maximum of rows matching m/z condition

Columns in the PEAK table can also be queried using *m/z conditions*. The m/z condition is given inside curly braces to further limit the search. For example `Area{23} > 1000` finds *any* peaks with both MZ in the range 22.5 to 23.5 and Area greater than 1000. Note that this is equivalent to running the query `mz = 23 and Area > 1000` to return peakids.

6.1.2 Aggregation Operators

Conditions in curly braces can match multiple rows in the PEAK table. With aggregation operators you can collect peak data over a spectrum as part of a query. Aggregation operators specify how to collect data in these rows into a single value for the parent spectrum. The aggregated data are then compared with the search criterion (see Table 11). A common use of the aggregation operators is to query spectral composition. For example, `sum(Area{23}) > 1000` finds spectra with a large aggregate response in the range $m/z = 22.5$–23.5. This is a more reliable way of querying spectral composition than the previous example because mass spectral signals are occasionally split among multiple peaks with similar MZ values. For spectra without any peaks of the selected m/z, the aggregated data are zeros.

6.1.3 Relative Operators

Relative operators connect the column and value in a search condition (see Table 12). All the relative operators can be used for columns with time, single, or double type data. All the relative operators also can be used for identifier object columns. Columns with boolean and word data can only be queried using the == operator. Note that double equals sign requires an exact match; there will likely be few if any matches to an exact query for floating point numbers like `Da == 1.00001`.

Some relative operators are *range* operators like =[). Range operators expect two-element vector values so that *X =[) [A B]* means “find particles with *X* between *A* and *B*”. This is equivalent

Table 12: Relative Operators

Operator	Description
==	Exact match
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
= [ ]	*X =[] [A B]* is equivalent to *X >= A and X <= B*
= [ )	*X =[) [A B]* is equivalent to *X >= A and X < B*
=	*X = [A B]* is equivalent to *X >= A and X < B*
= ( ]	*X =(] [A B]* is equivalent to *X > A and X <= B*
= ( )	*X =() [A B]* is equivalent to *X > A and X < B*

to *X >= A AND X < B*. MZ ranges may have one value so that MZ = A finds particles with MZ >= A - 0.5 and MZ < A + 0.5. For MZ range criteria, the range is given by YAADA.DeltaMZ which has a default value of 0.5.

6.1.4 Values

Values are generally numbers. Time values can be entered as numbers in Matlab time format or as text. Time text can have the form DD-MMM-YYYY HH:MM:SS or DD-MMM-YY HH:MM:SS. For example to search for particles detected in the morning of September 24, 1996, use the query Time = [24-Sep-96 06:00:00 24-Sep-96 12:00:00]. This is equivalent to Time = [729292.25 729292.5]. False boolean values are entered as 0 or a word starting with “F” or “N”; all other values are interpreted as true.

6.1.5 Set Operators

Elementary queries find sets of particles, these sets can be combined with set operators (see Table 13). Set Operators are evaluated from left to right unless parentheses alter the operator precedence.

6.2 Returned Identifiers

A table name can be given for the run_query function to specify the type of identifier returned. The returned identifier objects are the primary key for the table, e.g. peakids are returned if the table name is PEAK. If no table name is given, run_query returns partids.

Table 13: Set Operators

Operator	Definition	Description
and	Intersection	Objects in both sets
or	Union	Objects in either set
xor	Exclusive Or	Objects in either set, but not both sets
andnot	Set Difference	Objects in the first set that are not in the second set

YAADA converts the primary key of the searched table to the returned identifier objects. For example, a search for partids based on data in the PEAK table would select peakids then promote them to partids. For a search for peakids based on data in the PART table, run_query selects partids then finds the child peakids. Note that promotion is a fast operation while get_children is slow.

6.3 Query Optimization

There are often many ways to specify a query, some of which are significantly faster than others. YAADA executes queries by loading each chunk into memory then finding the matches for that chunk; this process is repeated for all the applicable chunks in a study. Loading chunks consumes most of the time it takes to execute a query, so fast queries minimize the number of chunks read into memory. For example, the following searches can be expressed equivalently as

```
pid = run_query('A > 1 and B > 2');
```

or

```
pid1 = run_query('A > 1');
pid2 = run_query('B > 2');
pid = intersect(pid1,pid2);
```

These yield the same result. Execution of the more complex search is generally faster since for the elementary queries, YAADA loads many of the same chunks for both queries. This results in longer search times. Therefore, unless intermediate results of less complex queries are useful, queries should be complex to reduce execution times. The main exception to this recommendation is discussed next.

Recall that chunks contain data for one instid and that the data are sorted by time. Thus YAADA will minimize the number of chunks loaded if the query includes an *overall* InstCode and/or Time condition. The convention is to place the InstCode and Time conditions first in a query like

```
InstCode == BST and Time = [23-Sep-96 12:00:00 23-Sep-96 16:00:00]
and Area{23} > 1000
```

For this query, YAADA will likely load only one PART chunk and one PEAK chunk to execute the query. This substantially improves query execution times.

Note that the InstCode and Time conditions must apply to the entire query. If we replace the second and in the previous query with or, then YAADA will have to load *all* of the PART and PEAK chunks to execute

```
InstCode == BST and Time = [23-Sep-96 12:00:00 23-Sep-96 16:00:00]
or Area{23} > 1000
```

Similarly YAADA will load all the PART and PEAK chunks to execute

```
(InstCode == BST and Time = [23-Sep-96 12:00:00 23-Sep-96
  16:00:00] and Area{23} > 1000) or \\
(InstCode == BST and Time = [23-Sep-96 16:00:00 23-Sep-96
  20:00:00] and Area{23} > 1000)
```

This is because the InstCode and Time conditions only apply to part of the query. This query will execute much faster if split it is into two queries and the results are combined with a union operation.

References

[1] E. E. Gard, J. E. Mayer, B. D. Morrical, T. Dienes, D. P. Fergenson, and K. A. Prather. Real-time analysis of individual atmospheric aerosol-particles - design and performance of a portable ATOFMS. *Anal. Chem.*, 69:4083–4091, 1997.

[2] C. A. Noble and K. A. Prather. Real-time measurement of correlated size and composition profiles of individual atmospheric aerosol particles. *Environ. Sci. Technol.*, 30(9):2667–2680, 1996.

[3] K. Salt, C. A. Noble, and K. A. Prather. Aerodynamic particle sizing versus light-scattering intensity measurement as methods for real-time particle sizing coupled with time-of-flight mass spectrometry. *Anal. Chem.*, 68:230–234, 1996.

[4] E. E. Gard, M. J. Kleeman, D. S. Gross, L. S. Hughes, J. O. Allen, B. D. Morrical, D. P. Fergenson, T. Dienes, M. E. Gälli, R. J. Johnson, G. R. Cass, and K. A. Prather. Direct observation of heterogeneous chemistry in the atmosphere. *Science*, 279:1184–1187, 1998.

[5] L. S. Hughes, J. O. Allen, M. J. Kleeman, R. J. Johnson, G. R. Cass, E. E. Gard, D. S. Gross, M. E. Gälli, B. D. Morrical, D. P. Fergenson, T. Dienes, C. A. Noble, D.-Y. Liu, P. J. Silva, and K. A. Prather. The size and composition distribution of atmospheric particles in Southern California. *Environ. Sci. Technol.*, 33:3506–3515, 1999.

[6] L. S. Hughes, J. O. Allen, P. V. Bhave, M. J. Kleeman, G. R. Cass, D. Y. Liu, D. P. Fergenson, B. D. Morrical, and K. A. Prather. Evolution of atmospheric particles along trajectories crossing the Los Angeles basin. *Environ. Sci. Technol.*, 34:3058–3068, 2000.

[7] J. O. Allen, D. P. Fergenson, E. E. Gard, B. D. Morrical, L. S. Hughes, M. J. Kleeman, D. S. Gross, M. E. Gälli, K. A. Prather, and G. R. Cass. Particle detection efficiencies of aerosol time of flight mass spectrometers under ambient sampling conditions. *Environ. Sci. Technol.*, 34:211–217, 2000.

[8] J. O. Allen, D. P. Fergenson, S. H. Pastor, L. S. Hughes, and K. A. Prather. Analysis and visualization of single-particle mass spectral data using YAADA, an object-oriented software toolkit. In preparation.

[9] N. J. Higham and D. J. Higham. *Matlab Guide*. Society for Industrial & Applied Mathematics, 2000.

[10] S. J. Chapman. *MATLAB Programming for Engineers*. Brooks/Cole, second edition, 2001.

[11] D.-Y. Liu, R. J. Wenzel, and K. A. Prather. Aerosol time-of-flight mass spectrometry measurements during the atlanta supersite experiment: Part 1. Submitted for publication in *J. Geophys. Res.*

A Installation

In order to run YAADA, you need Matlab (version 5.3 or later) and Perl (version 5.6 or later) programs installed on your computer. Matlab is available from the Mathworks (www.mathworks.com). You can determine your version of Matlab with the command

```
>> ver
```

Perl is Open Source software available at (www.perl.com). You should also test the Perl installation by running Perl at the Matlab prompt as

```
>> ! c:/perl/bin/perl -v
```

where *c:/perl/bin* is the Perl installation directory. If Perl is correctly installed, a message like

```
This is perl, v5.6.1 built for cygwin
(with 1 registered patch, see perl -V for more detail)

Copyright 1987-2000, Larry Wall
```

will be displayed.

1. Create a YAADA home directory (*X*:/yaada).
2. Unzip YAADA.zip in the home YAADA directory. These subdirectories will be created
   X:/yaada/aerosol
   X:/yaada/class
   X:/yaada/contrib
   X:/yaada/demo
   X:/yaada/general
   X:/yaada/import
   X:/yaada/plot
   X:/yaada/quant
   X:/yaada/query
   X:/yaada/user
   Program files will be placed in the directory structure as documented in manifest.txt.
3. Initialize, make, and start the demo or other dataset as described in Section 2.1.

Patches are revised programs to fix bugs and extend the features of YAADA. To install patches, replace obsolete program files with their new versions, The patch files, if any, are located in the download site as separate files with the extension m. Be sure to copy the patch files to the correct subdirectory (see manifest.txt).

B Function Reference

This Appendix describes the functions in the standard YAADA packages. Each function is listed in its calling syntax followed by a description.

In this section, place holders are used for variables of different types; these are

bool	boolean variable
col	column object
id	any identifier object
idx	index to vector elements, column rows, or table rows
iid	instrument identifier object
kid	peak identifier object
num	numeric variable
ph	patch handle
pid	particle identifier object
S	structure for subscripted reference
sid	spectrum identifier object
str	string variable
tbl	table object
x	any type variable

B.1 Identifier Object Methods

The identifier objects, instid, partid, specid, and peakid, uniquely identify data elements. These objects also have an inherent hierarchy which establish relations between data elements. The identifier objects are composed of fields as:

Object	Parent	Field
instid	—	instrument serial number (0–65535)
partid	instid	particle serial number (0–4,294,967,295)
specid	partid	polarity (0–1)
peakid	specid	peak serial number (0–65535)

Most of the methods in this section are available for all the identifier objects. The main exceptions are the methods which create identifier objects; these are named instid, partid, specid, peakid.

B.1.1 Object Creation

[*iid*] = **instid** (*num*)
[*pid*] = **partid** (*iid,num*)
[*sid*] = **specid** (*pid,num*)
[*kid*] = **peakid** (*sid,num*)

The preferred method to create an object or set of objects is to input the parent object and a numeric vector. The numeric vector is the serial number or polarity of the new child object. N children are created from the same parent object if one parent object and a vector of length N are input.

The creation functions return a null object if called without parameters, e.g. partid. They return an empty object if called with empty input, e.g. partid([]).

[*iid*] = **instid** (*str*)
[*pid*] = **partid** (*str*)
[*sid*] = **specid** (*str*)
[*kid*] = **peakid** (*str*)

Objects also can be created from character matrices which are 1, 3, 4, or 5 columns wide for instid, partid, specid, and peakid, respectively. These formats are the same as those returned by the char methods described below.

[*iid*] = **instid** (*num*)
[*pid*] = **partid** (*num*)
[*sid*] = **specid** (*num*)
[*kid*] = **peakid** (*num*)

Objects also can be created from numeric matrices which have 1, 2, 3, or 4 columns for instid, partid, specid, and peakid, respectively. These formats are the same as those returned by the double methods described below.

[*iid*] = **instid** (*pid|sid|kid*)
[*pid*] = **partid** (*sid|kid*)
[*sid*] = **specid** (*kid*)

Another way to create an object is to promote a child object to its parent object. This is the preferred method to create an object from its children.

B.1.2 Type Conversion

[*str*] = **char** (*id*)

The char methods return a matrix of characters which are 1, 3, 4, or 5 columns wide for instid, partid, specid, and peakid, respectively. The character representation of instid is one column of instrument serial numbers. In the character representation of partid, the first column lists the instrument serial numbers, the second and third columns list the particle serial numbers. The particle serial numbers are the second column times 2^{16} plus the third column. In the character representation of specid, the first three columns are like those for partid and the fourth column lists the polarities. In the character representation of peakid, the first four columns are like those for specid and the fifth lists the peak serial numbers.

The character representations of identifier objects often include unprintable characters which produce odd behavior, for example bell ringing, when displayed on a terminal. The character representations can be thought of as 16 bit unsigned integer (uint16) representations. Characters are used instead of uint16 data because Matlab offers more built-in functions for character data than uint16 data.

[*num*] = **double** (*id*)

The double methods return a matrix of numbers 1, 2, 3, or 4 columns wide for instid, partid, specid, and peakid, respectively. The double representation of instid is a column vector of instrument serial numbers. The double representation of partid is a matrix in which the first column lists instrument serial numbers, and the second column lists the particle serial numbers. In the double representation of specid, the first two columns are like those for partid, and the third column lists the polarities. In the double representation of peakid, the first three columns are like those for specid, and the fourth lists the peak serial numbers.

B.1.3 Find Related Objects

[*iid*] = **pid2iid** (*pid*)
[*iid*] = **sid2iid** (*sid*)
[*pid*] = **sid2pid** (*sid*)
[*iid*] = **kid2iid** (*kid*)
[*pid*] = **kid2pid** (*kid*)
[*sid*] = **kid2sid** (*kid*)

These promotion methods convert an object to its parent object. Use of the creation methods, instid, partid, and specid, is preferred over these methods since creation methods work with any the child object type.

[*id*] = **get_children** (*id,ChildType*)

Finds all the objects of type *ChildType* that are related to a set of parent identifier objects. Get_children searches the data tables for related objects since the identity of children is *not* stored in objects. Therefore, this can be a much slower operation than the promotion methods, especially for a parent object with a large number of children.

B.1.4 Set Operations

[*id,idx,idx*] = **intersect** (*id,id,...*)
[*id,idx*] = **setdiff** (*id,id*)
[*id,idx,idx*] = **setxor** (*id,id*)
[*id,idx,idx*] = **union** (*id,id,...*)
[*idx*] = **ismember** (*id,id*)

The set operations combine sets of identifier objects like the built-in Matlab functions. See the Matlab functions of the same name for explanations of the returned indices. The intersect and union methods can operate on more than 2 sets of identifier objects. In this case, the indices are not meaningful and are not returned.

Different identifier object types can be combined with set operations. In this case, the child object is promoted and the set operation returns a parent object.

[*OutID,OutName*] = **combine** (*InID,InName,bool*)

Combine creates combinations of identifier objects The input InID and InName are cell vectors of set ids and names. Names must be unique. If InName is omitted, the sets are named 'A' through 'Z'. The boolean variable is true for an exclusive combination; this defaults to true.

OutID is a nested cell vector of combined identifier objects. The first nested vector, OutID{1}, contain uncombined data sets. OutID{1}{1} contains the uncombined first data set ('A'). The second nested vector, OutID{2} contains combinations of 2 data sets so that OutID{2}{1} contains the combination of the first two data sets ('A-B'). Note that the vector lengths vary. OutName contains names for the output sets in the same order as OutID. For exclusive combinations, names are concatenated with '-'; single data sets are given a suffix '-'; examples are 'A-B-C', 'A-'. For inclusive combinations, names are concatenated with '+'; single data sets are given a suffix '+'; examples are 'A+B+C', 'A+'.

B.1.5 Relational Operations

Function Definition	**Symbolic Equivalent**
[*bool*] = **eq** (*id,id*)	*id == id*
[*bool*] = **ge** (*id,id*)	*id >= id*
[*bool*] = **gt** (*id,id*)	*id > id*
[*bool*] = **le** (*id,id*)	*id <= id*
[*bool*] = **lt** (*id,id*)	*id < id*
[*bool*] = **ne** (*id,id*)	*id = id*

Relational operator methods compare two identifier objects and return a boolean vector. These operators are commonly called symbolically, e.g. pid1 >= pid2 is equivalent to ge(pid1,pid2).

Comparisons are made first between parent objects. Thus, a comparison of two peakids tests the instrument serial numbers, particle serial numbers, polarities, then peak serial numbers in sequence.

Sets of objects can be compared; in this case the compared objects must either have the same length, or one of the objects must have unit length. The returned boolean vector has the same length as the the longest object.

Different identifier object types can be compared. In this case, the child object is promoted before comparison.

B.1.6 Arithmetic Operations

Function Definition	**Symbolic Equivalent**
[*id*] = **plus** (*id,num*)	*id* = *id* + num
[*id*] = **minus** (*id,num*)	*id* = *id* − num

Arithmetic operator methods add (subtract) integers to (from) the serial numbers in identifier objects. Overflow (underflow) operations generate warnings and the serial number is set to the maximum value (zero). These operators are commonly called symbolically, e.g. pid2 = pid1 + N.

Arithmetic operations are not defined for specid since addition to (subtraction from) polarity is not meaningful.

B.1.7 Display

display (*id*)

Display shows identifier objects as columns of 16 bit unsigned integers.

[*num*] = **length** (*id*)

Length returns the number of objects in a set.

[*num*] = **end** (*id*)

End returns the number of objects in a set as part of an indexed reference. For example pid1 = pid1(2:end) removes the first object from pid1.

B.1.8 Sorting and Combining

[*id,idx*] = **sort** (*id*)
[*id,idx,idx*] = **unique** (*id*)

Sort returns identifier objects in ascending order; objects are sorted in parent–child order. Unique returns identifier objects in ascending order with no repetitions. The original indices of the sorted sets are returned. See the Matlab functions of the same name for explanations of the returned indices.

[*id*] = **clone** (*id,num*)

Clone creates *num* duplicate objects from one identifier.

[*id*] = **merge** (*id,id*)

Merge combines objects. The merged object is not sorted and may contain duplicates.

B.1.9 Subscript Referencing

[*x*] = **subsref** (*id,S*)

Subscript reference methods return object data using a format similar to that used to retrieve structure elements in Matlab. For example pid1(1:5) returns the first five particle identifiers in the set. The parameter S specifies how the object is referenced and is created by Matlab (see Matlab documentation on object methods for details on S).

Identifier objects can be referenced by subscripts in three ways:

- *id2 = id1(idx)*
- *id2 = id1(idx).ParentName*
- *num = id1(idx).SerialName*

In the first case, objects of the original type are returned for the matching indices. In the second case parent objects are returned for the matching indices. In the third case serial numbers or polarity values are returned for the matching indices.

[*id*] = **subsasgn** (*id,S,x*)

Identifier objects also can be assigned by subscripts in three ways:

- *id2(idx) = id1*
- *id2(idx).ParentName = id1*
- *id2(idx).SerialName = num*

In the first case, assigned objects are of the original type. In the second case parent objects are assigned. In the third case serial numbers or polarity values are assigned.

B.2 Column Methods

Column objects contain one type of data, for example the aerodynamic size of particles. The column object contains its own description and optional sorting index. Column objects have the structure shown in Table 2. Data can be of one of the types shown in Table 3. User-written objects, including the identifier objects, are also valid data types.

B.2.1 Column Creation

[*col*] = **column** (*ColumnName,Description,Units,Type,MakeIndex,Data*)

Creates a column object with the given name, description, units, and type. *ColumnName* and *Type* are words. *Description* and *Units* are text. *MakeIndex* is a boolean; it is true if the column will be indexed. *Data* is a vector of data.

B.2.2 Display

display (*col*)

Displays a summary of column contents in the form

Column: *ColumnName*, *Description* (*Units*)
Type: *Type*
Rows: *num*

[*num*] = **numrow** (*col*)

Returns the number of rows of data in the column.

B.2.3 Sorting and Combining

[*col*] = **sort** (*col*)

Sort returns the column with its index updated if MakeIndex is true.

[*col*] = **merge** (*col,col*)

Combines two columns into one. The merged data are concatenated from the first and second input columns, with data from the first input column coming first. The names, descriptions, units, types of the input columns must match.

[*col*] = **split** (*col,idx*)

Returns a column with data from the the referenced rows of the original column. Note that split returns a *column object* while subsref (see B.2.4) returns *data* from the column.

B.2.4 Subscripted Referencing

[*x*] = **subsref** (*col,S*)

Subscript reference methods return column data using a format similar to that used to retrieve structure elements in Matlab. The parameter S specifies how the object is referenced and is created by Matlab (see Matlab documentation on object methods for details on S).

Column objects can be referenced by subscripts in two ways:

- *x = col.FieldName*
- *x = col(idx)*

Use the first form to retrieve any of the non-data fields like units. Use the second form to retrieve a range of data given by the row indices. Note that the second form returns data from the column, not a column object.

[*x*] = **subsasgn** (*col,S,x*)

Column objects can be assigned by subscripts in two ways:

- *col.FieldName = x*

Table 14: Table Object Structure

table.name	name of table
.rowsize	size of a row in bytes
.collist	list of column names in cell vector of words
.primarykey	name of primary key column
.column	cell vector of column objects

- *col(idx) = x*

Use the first form to assign any of the non-data fields. Use the second form to assign a range of data given by the row indices. The assigned data must have the same number of elements as there are row indices; or have one element, in which case that element is assigned to all the rows.

[*idx*] = **search** (*ColumnName,RelOp,Limit*)

Search finds data in column which match a condition. RelOp is a relative or range operator (see Table 12). Limit is the limit (range) to search for. Limit should be a scalar for relative operator searches; a 2 element vector for range searches. Search uses the sorted column index if available.

B.3 Table Methods

Table objects contain complex data in rows and columns. Table objects are composed of a number of column objects which have the same number of rows. Table objects have the structure shown in Table 14.

B.3.1 Table Creation

[*tbl*] = **table** (*TableName*)
[*tbl*] = **table** (*TableName,Column1,Column2,...*)

A new table is created with *TableName* optionally followed by column objects.

[*tbl*] = **table** (*tbl,Column1,Column2,...*)

Call table with a table object followed by one or more column objects to add columns to an existing table.

B.3.2 Display

display (*tbl*)

Displays a summary of table contents in the form

```
Table TableName, num rows
ColumnName1
ColumnName2
...
ColumnNameN
```

[*num*] = **numrow** (*tbl*)

Returns the number of rows of data in the table.

[*num*] = **fieldcount** (*tbl*)

Fieldcount returns the number of fields including column objects in the table object.

B.3.3 Sorting and Combining

[*tbl*] = **sort** (*tbl*)

Data in all columns are sorted by the primary key. Sort indices are updated for other columns if the column's MakeIndex is true.

[*tbl*] = **merge** (*tbl,tbl*)

Combines two tables into one. The merged data are concatenated from the first and second input tables, with data from the first input table coming first. The row size, primary key, and column objects of the input tables must match. The output table inherits the name from the first input table.

[*tbl*] = **split** (*tbl,idx*)

Returns a table with data from the the referenced rows of the original table. Note that split returns a table object with the same structure as the input table.

B.3.4 Subscripted Referencing

[*x*] = **subsref** (*tbl,S*)

Subscript reference methods return table data using a format similar to that used to retrieve structure elements in Matlab. The parameter S specifies how the object is referenced and is created by Matlab (see Matlab documentation on object methods for details on S).

Table objects can be referenced by subscripts in these ways:

- *x = Table.FieldName*
- *tbl = Table({'ColumnName1','ColumnName2' ...})*
- *tbl = Table(idx,{'ColumnName1','ColumnName2' ...})*
- *col = Table.ColumnName*

- *col = Table('ColumnName')*
- *x = Table(idx).ColumnName*
- *x = Table(idx,'ColumnName')*

Use the first form to retrieve any of the non-data fields like collist. For the other forms, the type of object returned depends on what was specified. The returned object types are:

Table object is returned when multiple columns are specified. The returned table contains only the rows and columns specified. If rows were not specified, all rows of the original table are included. The new table's name is temp.

Column object is returned when a single column is specified and no rows are specified.

Data vector is returned when a single column and a row range are specified.

[*x*] = **subsasgn** (*tbl,S,x*)

Table objects also can be assigned by subscripts in these ways:

- *Table.FieldName = x*
- *Table(idx).ColumnName = x*
- *Table(idx,'ColumnName') = x*

Use the first form to assign any of the non-data fields like collist. Use the other forms to assign a data vector to rows in a column.

B.4 Aerosol Calculations

B.4.1 Particle Size Calibration

[*Da*] = **da_log** (*Coef,Velocity*)

Calculates aerodynamic diameter from measured velocity using a logarithmic calibration function. *Coef* is a vector of logarithmic coefficients and calibration limits in the form

Coef = *[a b Min Max]*.

Velocity is a vector of particle velocities in m/s. *Da* is a vector of particle aerodynamic diameters in μm. *Da* is calculated as

$$D_a = a * \exp(b * \text{Velocity})$$

for *Min* $<=$ *Velocity* $<$ *Max*. *Da* is set to NaN if *Velocity* is outside of the calibration range.

[*Da*] = **da_noz** (*Coef,Velocity*)

Calculates aerodynamic diameter from measured velocity using a mixed logarithmic and polynomial calibration function. *Coef* is a vector of coefficients and calibration limits in the form

Coef = *[C(1) C(2) …C(N+1) C(N+2) C(N+3) Min Max]*

where N is the order of the polynomial. Velocity is a vector of particle velocities in m/s. Da is a vector of particle aerodynamic diameters in μm. Da is calculated as

$$D_a = C(1) + C(2) * v + C(3) * v^2 ... + C(N+1) * v^N + C(N+2) * \exp(C(N+3) * v)$$

for *Min* <= *Velocity* < *Max*. *Da* is set to NaN if *Velocity* is outside of the calibration range.

[*Da*] = **da_poly** (*Coef,Velocity*)

Calculates aerodynamic diameter from measured velocity using a polynomial calibration function. *Coef* is a vector of coefficients and calibration limits in the form

Coef = *[C(1) C(2) …C(N+1) Min Max]*

where N is the order of the polynomial. *Velocity* is a vector of particle velocities in m/s. *Da* is a vector of particle aerodynamic diameters in μm. *Da* is calculated as

$$D_a = C(1) * v^N + C(2) * v^{(N-1)} + ...C(N+1)$$

for *Min* <= *Velocity* < *Max*. *Da* is set to NaN if *Velocity* is outside of the calibration range. Note that the order of coefficients in da_noz and da_poly are reversed.

B.4.2 Particle Size Conversion

[*Dp*] = **da2dp** (*Da,SpecGrav,Lambda*)

Converts aerodynamic to physical diameter. *Da* is particle aerodynamic diameter in μm. *SpecGrav* is the specific gravity. *Lambda* is the mean free path in μm. *Dp* is the physical diameter in μm. This function assumes spherically shaped particles. Lambda is optional and defaults to the value for air at standard temperature and pressure, 0.0651 μm.

[*Dp*] = **da2dp_lookup** (*Da,SpecGrav,Lambda*)

Converts aerodynamic to physical diameter using a lookup table. Variables are the same as those in da2dp. This function uses a lookup table with interpolation in *Da* and *SpecGrav*. *Dp* values are accurate to better than 10^{-5} relative to the precisely calculated values. The function is more efficient than da2dp and should be used for a large number of inputs. The lookup table is created when this function is first called and stored in YAADA.YAADADir as da2dptable.mat.

[*Da*] = **dp2da** (*Dp,SpecGrav,Lambda*)

Converts physical to aerodynamic diameter. *Dp* is particle physical diameter in μm. *SpecGrav* is the specific gravity. *Lambda* is the mean free path in μm. *Da* is the aerodynamic diameter in μm. This function assumes spherically shaped particles. Lambda is optional and defaults to

Table 15: Preprocessed data files

File Extension	Content
.inst	instrument operating conditions
.pkl	hit particle size and mass spectral data
.sem	missed particle size data
.sef	particle size data acquired during fast scatter mode

the value for air at STP, 0.0651 μm.

[*Da*] = **dp2da_lookup** (*Dp,SpecGrav,Lambda*)

Converts physical to aerodynamic diameter using a lookup table. Variables are the same as those in dp2da. This function uses a lookup table with interpolation in *Dp* and *SpecGrav*. *Da* values are accurate to better than 10^{-5} relative to the precisely calculated values. The function is more efficient than dp2da and should be used for a large number of inputs. The lookup table is is created when this function is first called and stored in YAADA.YAADADir as dp2datable.mat.

B.5 Database Structure and Import

B.5.1 Database Definition

data_def

Creates an empty database from a data definition. Each study must have exactly one data definition. The data definition can only be altered before the study data files are built.

The data definition is stored in the table (DATADEF) which describes itself and the other tables (see Table 5). Empty tables in the database are generated from the descriptions in DATADEF by data_def. DATADEF and the empty data tables are stored in datadef.mat in the current study directory (YAADA.StudyDir).

B.5.2 Conversion of Raw Data to PK2 Format

digest_tw97 (*PklDir,Pk2Dir*)
digest_tw00 (*PklDir,Pk2Dir*)
digest_tsi00 (*PklDir,Pk2Dir*)

Converts preprocessed data files created by the data acquisition software. Digest_tw97 and digest_tw00 processes data files generated by UCR data acquisition software by Tas Dienes (TasWare) to the PK2 format. digest_tsi00 processes data files generated by the the 2000 version of TSI, Inc., data acquisition software (see Table 15).

All files in *PklDir* directory and its subdirectories are processed. Raw data directories can contain one instrument file for the entire directory, or one instrument file for each .pkl file. If there is one instrument file in a directory, these data are copied to every .pk2 file and the base file name is arbitrary. If there is one instrument file for each .pkl, the base file name (prefix) must match the .pkl base file name.

PK2 format files are written in the *Pk2Dir* directory. These files are named in the form IIIYYYYMMDDHHMMSS.pk2 where the first three letters are the instrument code and the remainder is the time of the first particle in the PK2 file.

The bulk of the data manipulation for these programs are done by the Perl scripts tw97.pl, tw00.pl, and tsi00.pl.

B.5.3 Data Importation and Verification

digest_pk2 (*Pk2Dir*)

Creates data table chunks from PK2 data files where *Pk2Dir* is the directory containing PK2 files. All files in *Pk2Dir* and its subdirectories will be processed.

It is common for a few PK2 files to have errors which prevent incorporation of the data. Digest_pk2 is designed so that these errors do not require that the entire database be redigested. Digest_pk2 saves its important data to a file after each PK2 file is successfully digested. If an error, typically an error during file loading, occurs, the user can fix the problematic PK2 file and restart the digestion process where it left off by calling digest_pk2 again. To discard the intermediate results and restart the digestion process, reinitialize the database with init, then run digest_pk2.

Digest_pk2 calls the Perl script pk2split.pl to split PK2 format data files to smaller files which are conveniently read by Matlab. The PK2 file is split into these files in the temporary directory

File Name	Content
table.tmp	data format
inst.tmp	instrument data
part.tmp	particle data
spec.tmp	spectral data
peak.tmp	peak data

The table.tmp and inst.tmp files have lines with the same format as the PK2 file. The part.tmp, spec.tmp, and peak.tmp files are matrices of numbers which Matlab reads quickly. The temporary files are deleted when digest_pk2 exits.

[*DataDef*] = **parse_table** (*TableFile*)

Reads text file of table data (*TableFile*) which has 2 types of lines

- comments that start with *%*
- table definition lines with the form *TableName: ColumnName1 ... ColumnNameN*

DataDef is a cell matrix with rows in the form *{TableName {Column1Name, Column2Name ...}}*. Note that DataDef is a cell matrix and DATADEF is a table object; these contain similar information in different forms.

[*[iid,bool]*] = **parse_inst** (*DataDef,InstFile*)

Reads text file of instrument data and updates the instrument table (INST). *DataDef* is a cell matrix output by parse_table; this is currently ignored. *InstFile* is a text file with instrument data; it has 2 types of lines

- comments that start with *%*
- data lines with the form *Column = Value*

Every PK2 file has instrument data; since many PK2 files have the same instrument data, a new instrument is added only when data in the *InstFile* differs from the last instrument. *Parse_inst* returns the current instid and true if a new instrument was added.

parse_part (*DataDef,PartFile,SpecFile,PeakFile,iid*)

Reads data in the format of the *DataDef* cell matrix from files with particle (*PartFile*), spectral (*SpecFile*), and peak (*PeakFile*) data. The data files are uncommented tables of numbers which can be read into Matlab quickly. The *PartFile* columns are in the order given by *DataDef* with these additions:

- the lines start with the particle serial number
- times are expanded into separate columns for year, month, day, hour, minute, and second

The *SpecFile* columns are in the order given by *DataDef* with the addition that lines start with the particle serial number. The *PeakFile* columns are in the order given by *DataDef* with the addition that lines start with the particle serial number.

Parse_part adds new data to the PART, SPEC, PEAK tables in memory. It then calls split_chunk to save parts of these tables to chunk files.

split_chunk (*CloseChunk*)

Splits tables into chunks and saves them to files. New chunks are created when the table in memory exceeds the recommended chunk size (YAADA.ChunkSize). New chunks are also created for new instruments. SPEC and PEAK chunks are split so that data for a single particle are not split between two chunks. If *CloseChunk* is true then all data in the tables in memory are written as chunks, including remainders of tables that are smaller than YAADA.ChunkSize.

resort (*tbl*)

Sorts all tables in a chunk list table (*tbl*).

update_da

Calculates Da from Velocity data for all PART chunks.

update_hit

Determines if particles were hit from spectral data and updates the Hit column. This is done for all PART chunks.

update_area

Calculates AreaIntegral in all SPEC chunks. Also calculates RelArea in all PEAK chunks.

B.5.4 Data Integrity Checks

check_all

Runs check_chunk, check_id, and check_part to check the integrity of the entire database.

check_chunk (*tbl*)

Checks the contents of chunk list table (*tbl*) pointers and reports these errors:

- Overlapping primary keys in a chunk list entry
- Overlapping primary keys between chunk list entries
- Mismatched primary keys between a chunk list entry and related chunk
- Mismatched times between a chunk list entry and related chunk

check_id (*tbl*)

Checks that identifier objects are unique within each chunk. Running the combination of check_chunk and then check_id tests if identifier objects are unique in the entire database.

check_part

Checks that physical particles are unique in the entire database. “Physical particles” are those with a unique combination of InstCode—Time—Velocity. A small number of duplicate InstCode—Time—Velocity combinations are expected since time and velocity data are discretized. Duplicate combinations of InstCode—Time—Velocity can be ignored if they are not continuous and are less than 1% of the particles in the database.

B.6 Chunk Handling

[*ChunkName*] = **find_chunk** (*TableName,id*)
[*ChunkName*] = **find_chunk** (*TableName,InstCode,Start,Stop*)

Finds chunks for a virtual table that contain specific data. Chunk names are returned as a cell vector. The first form returns chunk names for the virtual table TableName and are related to the identifier objects in id. The second form returns chunk names related to an instrument code (InstCode) and time range (Start and Stop). Time limits can be omitted or given as NaN to ignore a limit.

load_chunk (*ChunkName,TableName*)

Reads a chunk of data from a file (*ChunkName*) and stores it in global variables. If a *TableName* is given, the chunk is loaded into the table global variable (PART, SPEC, or PEAK). The chunk is not loaded from disk if it is already in the table variable. If the chunk is loaded from disk, the chunk is also stored in CHUNK.

If a *TableName* is not given, the chunk is loaded into the CHUNK global variable. The chunk is not loaded from disk if it is already in CHUNK. CHUNK is a generic “register” to hold the latest loaded chunk. Functions which do not know or care about the table which owns a chunk should call load_chunk without a table name.

The currently opened chunks are stored in YAADA.OpenChunk. Load_chunk reads and updates this information. For this reason chunks should *not* be loaded with the Matlab load function.

B.7 Query Processing

[*id*] = **run_query** (*QueryText,TableName,Verbose*)

Finds identifiers which match a query. *QueryText* is the search criteria which is parsed by parse_query; see parse_query for a description of query syntax. *TableName* is the table whose primary key id objects are returned. *TableName* is optional and defaults to PART. If *Verbose* is set, the elapsed time and number of matches are shown after the query has been executed. *Verbose* defaults to YAADA.Verbose if omitted.

[*QueryCell*] = **parse_query** (*Query,TableName*)

Parses a query and returns a nested cell. Queries are text strings with elementary queries joined by set operators. The elementary queries are made of

- Column Name
- Relational or Range Operator
- Value

Column Names specify the type of data to search, valid column names are listed in DATADEF. Valid relational operators are:

==	Equal to
<, >	Less than, greater than
<=, >=	Less than or equal to, greater than or equal to

Valid Range Operators are:

=[]	*Min <= X <= Max*
=	*Min <= X < Max*
=[)	*Min <= X < Max*
=(]	*Min < X <= Max*
=()	*Min < X < Max*

Note that = is shorthand for =[). Values are scalars for relational operators and two element vectors for range operators.

Set operators are used to combine elementary searches. Valid set operators are

and	intersection
andnot	set difference
or	union
xor	exclusive or

Set operators are evaluated from left to right unless parentheses alter the operator precedence.

Additional search criteria are available to search for peak data within a spectrum. These are described in parse_column.

[*AggOp,ColumnName,RowCond*] = **parse_column** (*ColumnQuery*)

Parses a complex column in a query. Columns can include optional aggregation operations and conditions in one of these forms

- *ColumnName*
- *ColumnName {RowCondition}*
- *AggOp(ColumnName)*
- *AggOp(ColumnName {RowCondition})*

Row conditions and aggregation operations are applicable only for peak data within a spectrum. These are

- Mass-to charge (m/z) conditions
- Aggregation operators
- Relative comparisons

Peak m/z conditions are given in curly braces directly after the column name, e.g. Area{23} > 100 searches for peaks at $m/z = 23$ with areas greater than 100. The m/z criterion may be a range of two numbers or a single number. In the case of a single number, the range is the number +/- YAADA.DeltaMZ. The default value of YAADA.DeltaMZ is 0.5.

Aggregation operations condense data from multiple rows into one value (see Table 11).

[*QueryText*] = **disp_query** (*QueryCell*)

Displays a parsed query in the order the query will be run.

B.8 Data Retrieval

[*ColData1,ColData2...*] = **get_column** (*id,ColName1,ColName2...*)

Returns data from a virtual table for a set of identifier objects, *id*, which are the primary keys for a table. Data from columns in the table, *ColName1,ColName2...*, are returned in vectors *ColData1,ColData2...*. The columns must all be in a table which has *id* as its primary key. Get_column returns NaN for identifier objects not found in the data table.

[*Spectrum*] = **get_spectrum** (*pid,Polarity,ColList*)

Returns spectra for particles in the *pid* set. *Spectrum* is a cell vector with one element for each particle, *i*, in this format:

Spectrum{i,1}	PeakID
Spectrum{i,2}	MZ

Additional columns contain data from columns in ColList. ColList is optional and defaults to {'Area','RelArea','Height','Width', 'BlowScale'}. So the default cell matrix has the form

Spectrum{i,1}	PeakID
Spectrum{i,2}	MZ
Spectrum{i,3}	Area
Spectrum{i,4}	RelArea
Spectrum{i,5}	Height
Spectrum{i,6}	Width
Spectrum{i,7}	BlowScale

The contents of each element are vectors with one element for each peak in the spectrum. The vectors are empty for particles which do not have a spectrum of *Polarity*. Note that the lengths of these vectors differ among the particles.

[*NegResponse,PosResponse*] = **get_int_spectrum** (*pid,MaxMZ,ResponseType,Polarity,AggOp*)

Returns table of responses at integral m/z for particle identifiers. *NegArea* and *PosArea* are matrices of peak areas aggregated using *AggOp* for integral negative and positive m/z values. *MaxMZ* is the upper (lower) limit of m/z range; the *PosResponse* columns span m/z = 1 to *MaxMZ*, the *NegResponse* columns span m/z = -1 to -*MaxMZ*. *MaxMZ* is optional and defaults to 350. *ResponseType* can be any column in PEAK; this is optional and defaults to Area. *Polarity* specifies the spectrum polarity as

0	negative spectra
1	positive spectra
2	negative and positive spectra (default)

AggOp specifies how to combine multiple peaks with the same integral m/z value. Valid *AggOp*s are COUNT, MEAN, MEDIAN, SUM , MIN, MAX. *AggOp* is optional and defaults to SUM.

The rows of *NegResponse* and *PosResponse* match the particles in *pid*. For example *ResponseType* Area and *AggOp* SUM, *PosResponse*(1,23) is the sum of areas of peaks with m/z = 22.5-23.5 for the first particle in *pid*. All the rows of the area tables are zeros for missed particles.

[*OutID,OutIDIdx,IDCount,BinCut,BinMid*] = **bin_on_column** (*InID, ColumnName,Start,Stop,NumBin,Scale*)

Bins set of identifiers based on column data. InID is a set of identifier objects. ColumnName is the name of the column on which to bin the InID set. The column must be from a table whose primary key is the same type as InID. Data are collected into NumBin bins which begin at Start and end at Stop. The bins have uniform widths on a linear or logarithmic scale, specified with Scale as lin or log. Scale is optional and defaults to linear. Bin_on_column can also be called with a vector of column bin divisions given specifically in Start; in this case Stop, NumBin, and Scale are ignored.

OutID is a cell array, each cell contains the IDs for a bin. OutIDIdx is a cell array, each cell contains pointers from the binned identifier objects to the original InID set. IDCount is a vector of identifier object counts in each bin. BinCut is a vector of bin divisions. Note, there are N+1 bin divisions. BinMid is a vector of bin midpoints.

B.9 Plot Formats

Abbreviated documentation of the plotting programs are presented here. Use help *Function-Name* to view more complete documentation online.

B.9.1 Plots

digital_ms (*PartID,Polarity,MinPeakArea,PeakMZRange,ResponseType*)

Digital_ms draws a digital mass spectrum.

[*lh,eh*] = **lundgren** (*DaCut,conc,err,linestyle,limits*)

Lundgren plots aerosol distribution versus $\log(D_a)$.

[*lh*] = **lundlog** (*DaCut,conc,linestyle,limits*)

Lundlong plots log aerosol distribution versus $\log(D_a)$.

[*ph*] = **Lundstack** (*DaCut,conc,color,style,numlines,limit*)

Lundstack plots data in stacked Lundgren plot.

msview

Interactively displays particles and mass spectra.

plot_hit_miss

Plots frequency of hit and missed particles versus time.

[*ph*] = **timestack** (*Start,Stop,Conc,Color,Style,NumLines,Limit*)

Plots stacked time series data.

B.9.2 Crosshatching

[*ph*] = **xhatch** (*xx,yy,pattern,numlines*)

Draws a black and white pattern of lines or dots in a rectangle.

[*xh*] = **xhatch_bar** (*xx,style,numlines*)

Plots a stacked bar graph with patches instead of colors.

[*lh*] = **xhatch_legend** (*pattern,numlines,desc,pos*)

Makes legend for xhatch plots.

B.9.3 Plot Formatting

set_font

Sets default font and font size for plots.

[*th*] = **text_rel** (*relx,rely,words*)

Places text on current axes at a relative position.

[*th*] = **xlabel_timedate** (*NewLim, ShowDates, Offset, FontName, FontSize*)

Labels current figure x axis with dates and times.

B.10 Quantitative Comparison

The quantitative comparison package is under development. Only the instrument busy time function is included in this release.

B.10.1 Instrument Busy Time

[*BusyTime*] = **busy_time** (*InstID,Start,Stop,NumBin*)

Busy_time calculates instrument busy time in days from NumSized, the number of particles sized in a period, NumHit, the number of particles hit in a period, and AvgPosInFolder, the average PositionInFolder for the hit particles. Note that hit particles are counted in both NumSized and NumHit.

$\text{BusyTime} = a * \text{NumSized} + b * \text{NumHit} + c * \text{NumHit} * \text{AvgPosInFolder}$

B.11 General Functions

B.11.1 Search

[*Idx*] = **binary_search** (*X,RelOp,Limit*)

Finds rows in sorted array which match a condition. *X* can be a matrix of numbers, a vector of id objects, or a column vector sorted in ascending order. If *X* is a matrix, comparisons are made on rows. *RelOp* is a relational operator <, <=, >, >=, ==. *Limit* is the minimum (maximum) value to search for.

[*Idx*] = **search** (*X,RelOp,Limit*)

Finds elements in a vector which match a condition. *X* can be a vector of numbers or id objects. *RelOp* can be a valid relative or range operator (see Table 12). *Limit* is the limit (range) to search for. *Limit* should be a scalar for relative operator searches; a 2 element vector for range searches.

[*Idx*] = **range_search** (*X,RangeOp,Limit*)

Finds rows in sorted array which match a range condition. *X* can be a matrix of numbers, a vector of id objects, or a column vector sorted in ascending order. If *X* is a matrix, comparisons are made on rows. *RangeOp* is a valid range operator (see Table 12). *Limit* is a 2 element vector with the minimum and maximum values to search for.

B.11.2 Row-wise Matrix Comparison

[*Truth*] = **eqrow** (*A,B*)
[*Truth*] = **nerow** (*A,B*)
[*Truth*] = **gerow** (*A,B*)
[*Truth*] = **gtrow** (*A,B*)
[*Truth*] = **lerow** (*A,B*)
[*Truth*] = **ltrow** (*A,B*)

These functions compare the rows of matrices *A* and *B*. *A* and *B* must have the same number of columns. *A* and *B* must have an equal number of rows, or one must have only one row. Leftmost columns are the most significant in the comparison, so for

```
A = [1 2; 3 4; 5 6; 7 8]
B = [1 2; 4 3; 5 7; 7 7]
Truth = ltrow(A,B);
```

returns Truth = [0; 1; 1; 0]. *Truth* is a vector with the same number of rows as *A* and *B*.

B.11.3 String Operations

[*Pos*] = **findword** (*String,Word*)

Finds a whole word within a string. Words are bounded by white space or the beginning and ending of the string. *Pos* is the position of the start of the word in the string.

[*PartCell*] = **get_part_str** (*pid*)

Create a cell vector describing a set of partids. *PartCell* is a cell vector of strings, one for each particle in the form

```
III DD-MMM-YYYY HH:MM:SS DD.DD
```

where *III* is the InstCode, *DD-MMM-YYYY* is the date, *HH:MM:SS* is the time, and *DD.DD* is the aerodynamic diameter in μm.

[*String*] = **trim** (*String*)

Removes leading and trailing blanks from a string.

B.11.4 NaN Operations

[*x*] = **maxnan** (*x*)
[*x*] = **meannan** (*x*)
[*x*] = **mediannan** (*x*)
[*x*] = **minnan** (*x*)
[*x*] = **sortnan** (*x*)
[*x*] = **sumnan** (*x*)

These functions perform operations on vectors ignoring NaN values. Note that similar functions with names like nanmax are available in the Matlab Statistics Toolkit.

B.11.5 Type Identification

[*x*] = **isdigit** (*bool*)

True if *x* is a digit (0-9).

[*bool*] = **isid** (*x*)

True if *x* is an identifier object.

[*bool*] = **isinteger** (*x*)

True if *x* is an array of integers.

[*bool*] = **ispunct** (*x*)

True if *x* is a punctuation mark.

[*bool*] = **isscalar** (*x*)

True if *x* is a scalar.

[*bool*] = **istablename** (*x,TableType*)

True if *x* is a table in the current database. *TableType* is an optional table type like

all	any table (default)
data	only data tables (DATADEF, ChunkLists excluded)
chunklist	only chunklist tables

[*bool*] = **isvector** (*x*)

True if *x* is a scalar or vector.

[*bool*] = **isword** (*x*)

True if *x* is a character string without spaces.

B.11.6 Type and Object Operations

[*bool*] = **bool2num** (*x*)

Converts logical values to binary values. Bool2num makes these conversions:

x	bool
non-zero number	1
words starting with “f”,”F’,”n”, or “N”	0
all other words	1

Groups of words should be input as a cell vector; groups of numbers as a vector.

[*Compare*] = **cmp_id_class** (*id1,id2*)

Compares classes of ID objects. *Compare* is 0 if the *id1* and *id2* are objects of the same class; -1 if the class of *id1* is inferior to *id2*; +1 if the class of *id1* is superior to *id2*. *Id1* and *id2* can be identifier objects or names of identifier object classes.

[*x*] = **empty_type** (*Type*)

Returns an empty variable of *Type*.

[*TableName*] = **list_table** (*TableType*)

Returns a list of tables in current database as a string. Tables in the string are selected with *TableType* as

TableType	Returned Tables
all	all tables (default)
data	only data tables (DATADEF and ChunkLists excluded)
chunklist	only chunklist tables

Table names are returned as capitalized words separated with spaces. A common use of list_table is to make tables accessible as global variables. To do this:

```
eval(['global ' list_table(TableType))]);
```

[*TableName*] = **list_table2** (*TableType*)

Returns a list of tables in current database as capitalized words in a cell vector. Tables in the cell are selected with *TableType* (see list_table).

[*x*] = **null_type** (*Type*)

Returns a null variable of *Type*.

[*B*] = **promote_id** (*A,IDType*)

Promotes an identifier object *A* to a parent object type, *IDType*. Output objects are unique and sorted.

B.11.7 Miscellany

[*num*] = **datenum2** (*str*)

Calls Matlab function datenum, but in case of bad date string, displays a warning and returns 0.

[*C,IA,IB*] = **intersect** (*A,B,Flag*)

Finds values *C* common to sets *A* and *B*. Index vectors *IA* and *IB* are returned such that *C = A(IA)* and *C = B(IB)*, or for matrices *C = A(IA,:)* and *C = B(IB,:)*.

Calling as intersect(*A*,*B*,'rows') where *A* are *B* are matrices with the same number of columns returns the rows common to *A* and *B*.

Calling as intersect(*A*,*B*,'sorted') returns the elements common to both the sorted arrays *A* and *B*. Intersection of sorted arrays is much faster than the normal intersect function. *A* and *B* must be sorted. Unlike the Matlab intersect row vectors are *not* recast to column vectors.

[*old*] = **obsolete** (*DependentFile, File1, File2, ...FileN*)

Determines whether a file is older than any files in a list. Returns 1 if *DependentFile* is older than any file in file list or if *DependentFile* does not exist. Returns 0 if *DependentFile* is newer than all files in file list. Returns -1 in case of error. File names can be full path names or relative path names in UNIX style.

[*x,KeyIdx*] = **sortstruct** (*SortedX,SortIdx*)

Sortstruct sorts structure using a key field. KeyIdx is the key field index, the default is 1.

[*BinCut,BinMid*] = **split_bin** (*Start,Stop,NumBin,Scale*)

Calculates bin divisions for *NumBin* bins which begin at *Start* and end at *Stop*. The bins have uniform widths on a linear or logarithmic scale, specified by setting *Scale* to lin or log. *Scale* is optional and defaults to linear. *BinCut* is a vector of bin divisions. Note, there are *NumBin*+1 bin divisions. *BinMid* is a vector of bin midpoints.

Can also be called with specific bin divisions as a vector in *Start*. In this case *Stop*, *NumBin*, and *Scale* are ignored.

[*Truth*] = **type_match** (*Type*,*x*)

True if *x* has type *Type*.

C Data File Formats

C.1 Instrument Data File Format

The .inst files are plain text files that contain the instrument conditions. ATFOMS data can be archived in these self-contained, self-documenting, human readable, and platform independent files. The file is made up of two types of lines, comment lines which start with "%" and data lines in the form *Field = Value*. The first lines of a PK2 file should be comments which identify the instrument and author. Each column of instrument data is given on a separate line. The names and descriptions of these fields are given in the data definition table. Vectors should be entered on the same line separated by white space.

```
% This is an example .inst data file

% Instrument Table Data

AvgLaserPower = 1.0
BusyTimeFunction = busy_scale
BusyTimeParam = [0.13 0.504 0.000167]
DaCalibFunction = da_noz
DaCalibParam = [0 0 0 0 0 0 0 7.658206E+01 -1.140446E-02 200 600]
InstCode = TST
InstName = Testmeister
InstDesc = Testmeister in Schonau
SampleFlow = 3.334e-7
ExpName = Synthetic
ExpDesc = Synthetic data creation exercise
PreProcDesc = by Sylvia W. Pastor, David P. Fergenson, Jonathan O. Allen
PreProcDate = 01-Jan-1999
MinHeight = 10
MinArea = 12
OpName = t1
OpDesc = Normal Operation
PosDefaultZero = 0
NegDefaultZero = 0
PosDefaultVoltage = 0
NegDefaultVoltage = 0

% end of example
```

C.2 PK2 Data File Format

YAADA imports ATOFMS data from data files in the PK2 format described in this section. The PK2 files are plain text files that contain all the instrument condition, particle, and mass spectral data for a sampling period. ATFOMS data can be archived in these self-contained, self-documenting, human readable, and platform independent files.

PK2 files contain data on a single instrument-sampling study combination. PK2 files written

Table 16: PK2 Data File Line Types

First Word	Line Type
%	Comment
Table:	List of tables in database
TableName:	List of columns in table
ColumnName	Instrument data
Date	Particle data
>	Spectrum data
>>	Peak data

by YAADA are named in the format IIIYYYYMMDDHHMMSS.PK2, where III is the instrument identifier, YYYYMMDD and HHMMSS give the date and time of the first particle acquired. The file is made up of seven types of lines identified by the first word (see Table 16).

The first lines of a PK2 file should be comments which identify the data source and any preprocessing. The next lines describe the database structure. The first line begins with Table: and lists the table names separated by whitespace. Subsequent lines list the table name, a colon, and column names separated by whitespace for each table.

Instrument data come after the database structure. Each column of instrument data is given on a separate line. The names and descriptions of these fields are given in the data definition table. Vectors should be entered on the same line separated by white space. The instrument conditions must be the same for all the data in a PK2 file.

Data for each particle are given on a separate line with the columns printed in the same order as listed on the Particle: line. For the default database, particle data lines have the format:

```
Time Velocity PositionInFolder FastScatter
```

Time has the format DD-MMM-YYYY_HH:MM:SS. Velocity is a floating point number in m/s. PositionInFolder is an integer. FastScatter is a boolean values, 0 or 1.

Mass spectra data are given on separate lines after the data for the parent hit particle. Lines of spectrum data start with ">". For the default database, spectrum data lines have the format:

```
> Polarity FileNameLength AreaIntegral Noise BaseLine FitVoltage FitZero
```

Peak data are given on separate lines immediately after the data for the parent spectrum. Lines of peak data start with ">>". For the default database, peak data lines have the format:

```
>> MZ Area Height Width BlowScale
```

MZ, Area, Width, and Height are floating point numbers. BlowScale is a boolean value, 0 or 1.

```
% Demonstration data set for YAADA.  These data are _synthetic_
% and do not represent actual aerosol sampling results
%
% Created 05 Jan 00

% Data structure updated
% JOA  24 Dec 01

Table: Inst Part Spec Peak
Inst: AvgLaserPower BusyTimeFunction BusyTimeParam DaCalibFunction DaCalibParam ExpDesc ExpName
Part: Time Velocity PositionInFolder FastScatter
Spec: Polarity FileNameLength
Peak: MZ Area Height BlowScale

% Instrument Table Data

AvgLaserPower = 1.0
BusyTimeFunction = busy_scale
BusyTimeParam = [0.13 0.504 0.000167]
DaCalibFunction = da_noz
DaCalibParam = [0 0 0 0 0 0 0 7.658206E+01 -1.140446E-02 200 600]
InstCode = TST
InstName = Testmeister
InstDesc = Testmeister in Schonau
SampleFlow = 3.334e-7
ExpName = Synthetic
ExpDesc = Synthetic data creation exercise
PreProcDesc = by Sylvia W. Pastor, David P. Fergenson, Jonathan O. Allen
PreProcDate = 01-Jan-1999
MinHeight = 10
MinArea = 12
OpName = t1
OpDesc = Normal Operation
PosDefaultZero = 0
NegDefaultZero = 0
PosDefaultVoltage = 0
NegDefaultVoltage = 0

% Particle Data

01-Apr-1992_10:00:00   290.28     1 0
01-Apr-1992_10:00:01   327.69     2 0
01-Apr-1992_10:00:01   289.58     3 0
01-Apr-1992_10:00:02   295.06     4 0
01-Apr-1992_10:00:03   352.22     1 0
> 1 10
>> +0001.08   113   15 0
>> +0001.18   113   15 0
>> +0022.28   373   62 0
>> +0022.48   150   32 0
>> +0022.62   288  101 0
>> +0023.11    14   14 0
>> +0023.70    26   15 0
>> +0037.95    63   33 0
>> +0053.61    21   21 0
01-Apr-1992_10:00:04   293.61     5 0
01-Apr-1992_10:00:04   277.46     6 0

% end of example
```

D YAADA Programming Guidelines

To improve the readability and portability of YAADA programs, we have adopted these conventions.

D.1 File Locations

Physical locations of Matlab programs and data are set in the `startup.m` (or similar) file which sets the path to programs data.

D.2 Variable Names

Long lived variable names have the first letter of each word capitalized. Multiword names are concatenated, e.g. `SamplerCode`. Do not use plural form.

Short lived variables are those used for a few lines or within a single loop. These should be all lower case with concatenation, e.g. `sampleidx`.

D.3 Program Names

Scripts and functions are named by an action verb followed by the subject with words separated by "_", e.g. `run_query.m`. Script and function names are all in lower case.

D.4 Abbreviations

Some common abbreviations are

Col	column
Da	aerodynamic diameter
desc	description
Inst	instrument
kid	Peak identifier (also PeakID)
MZ	mass to charge ratio
num	number
Part	Particle
pid	Particle identifier (also PartID)
ptr	Pointer
Spec	Spectrum

D.5 Program Help

See the Matlab HELP.M file for instructions on how to add help to program files.