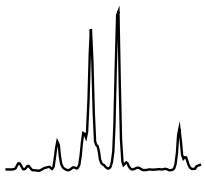

PyMS User Guide

with contributions from:

PyMS version 1.0

A Python toolkit for processing of chromatography–mass spectrometry data

Contents

Introduction

1.1 About PyMS

PyMS is a Python toolkit for processing of chromatography–mass spectrometry data. The main idea behind PyMS is to provide a framework and a set of components for rapid development and testing of methods for processing of chromatography–mass spectrometry data. An important objective of PyMS is to decouple processing methods form visualization and the concept of interactive processing. This is useful for high-throughput processing tasks and when there is a need to run calculations in the batch mode.

PyMS is modular and consists of several sub-packages written in Python programming language [1]. PyMS is released as open source, under the GNU Public License version 2.

There are four parts of the pyms project:

- pyms – The PyMS code
- pyms-docs – The PyMS documentation
- pyms-test – Examples of PyMS use

Each part is a separate project on Google Code that can be downloaded separately. The data used in PyMS documentation and examples is available from the Bio21 Institute server:
`http://bioinformatics.bio21.unimelb.edu.au/pyms-data/`
In addition, the current PyMS API documentation is available from here:
`http://bioinformatics.bio21.unimelb.edu.au/pyms.api/index.html`

1.2 PyMS installation

There are several ways to install PyMS depending your computer configuration and preferences. The recommended way install PyMS is to compile Python from sources and install PyMS within the local Python installation. This procedure is described below.

PyMS has been developed on Linux, and a detailed installation instructions for Linux are given below. Installation on any Unix-like system should be similar. We have not tested PyMS under Microsoft Windows.

1.2.1 Downloading PyMS source code

PyMS source code resides on Google Code servers, and can be accessed from the following URL: http://code.google.com/p/pyms/. Under the section "Source" one can find the instructions for downloading the source code. The same page provides the link under "This project's Subversion repository can be viewed in your web browser" which allows one to browse the source code on the server without actually downloading it.

Google Code maintains the source code with the program 'subversion' (an open-source version control system). To download the source code one needs to use the subversion client program called 'svn'. The 'svn' client exists for all mainstream operating systems[1], for more information see http://subversion.tigris.org/. The book about subversion is freely available on-line at http://svnbook.red-bean.com/. Subversion has extensive functionality. However only the very basic functionality is needed to download PyMS source code.

If the computer is connected to the internet and the subversion client is installed, the following command will download the latest PyMS source code:

```
$ svn checkout http://pyms.googlecode.com/svn/trunk/ pyms
A    pyms/Peak
A    pyms/Peak/__init__.py
A    pyms/Peak/List
A    pyms/Peak/List/__init__.py
.....
Checked out revision 71.
```

1.2.2 PyMS installation

PyMS installation consists of placing the PyMS code directory (pyms/) in place visible to Python interpreter. This can be in the standard place for 3rd party software (the directory site-packages/). If PyMS code is placed in a non-standard place the Python interpreter needs to be made aware of it before before it is possible to import PyMS modules (see the Python sys.path.append() command).

We recommend compiling your own Python installation for PyMS.

In addition to the PyMS core source code, a number of external packages is used to provide additional functionality. These are explained below.

[1] For example, on Linux CentOS 4 we have installed the RPM package 'subversion-1.3.2-1.rhel4.i386.rpm' to provide us with the subversion client 'svn'.

1.2.3 Package 'NumPy'

The package NumPy is provides numerical capabilities to Python. This package is used throughout PyMS (and also required for some external packages used in PyMS), to its installation is mandatory.

The NumPy web site `http://numpy.scipy.org/` provides the installation instructions and the link to the source code.

1.2.4 Package 'pycdf' (required for reading ANDI-MS files)

The pycdf (a python interface to Unidata netCDF library) source and installation instructions can be downloaded from `http://pysclint.sourceforge.net/pycdf/`. Follow the installation instructions to install pycdf.

1.2.5 Package 'Pycluster' (required for peak alignment by dynamic programming)

The peak alignment by dynamic programming is located in the subpackage pyms.Peak.List.DPA. This subpackage used the Python package 'Pycluster' as the clustering engine. Pycluster with its installation instructions can be found here: `http://bonsai.ims.u-tokyo.ac.jp/ mdehoon/software/cluster/index.html`.

1.2.6 Package 'scipy.ndimage' (required for TopHat baseline corrector)

If the full SciPy package is installed the 'ndimage' will be available. However the SciPy contains large amount of functionality, and its installation is somewhat involved. In some situations in may be preferable to install only the subpackage 'ndimage'. The UrbanSim web site [2] provides instructions how to install a local copy of 'ndimage'. These instructions and the link to the file 'ndimage.zip' are here:
`http://www.urbansim.org/opus/releases/opus-4-1-1/docs/installation/scipy.html`

1.3 Current PyMS development environment

PyMS is currently being developed with the following packages:

```
Python-2.5.2
numpy-1.1.1
netcdf-4.0
pycdf-0.6-3b
Pycluster-1.41
```

A quick installation guide for packages required by PyMS is given below.

1. Python installation:

```
$ tar xvfz Python-2.5.2.tgz
$ cd Python-2.5.2
$ ./configure
$ make
$ make install
```

   This installs python in /usr/local/lib/python2.5. Make sure that python called from the command line is the one just compiled and installed.

2. NumPy installation:

```
$ tar xvfz numpy-1.1.1.tar.gz
$ cd numpy-1.1.1
$ python setup.py install
```

3. pycdf installation

   Pycdf has two dependencies: the Unidata netcdf library and NumPy. The NumPy installation is described above. To install pycdf, the netcdf library must be downloaded (`http://www.unidata.ucar.edu/software/`
   compiled and installed first:

```
$ tar xvfz netcdf.tar.gz
$ cd netcdf-4.0
$ ./configure
$ make
$ make install
```

   The last step will create several binary 'libnetcdf*' files in /usr/local/lib. pycdf can be installed as follows:

```
$ tar xvfz pycdf-0.6-3b
$ cd pycdf-0.6-3b
$ python setup.py install
```

4. Pycluster installation

```
$ tar xvfz Pycluster-1.42.tar.gz
$ cd Pycluster-1.42
$ python setup.py install
```

5. ndimage installation:

```
$ unzip ndimage.zip
$ cd ndimage
$ python setup.py install --prefix=/usr/local
```

Since ndimage was installed outside the scipy package, this requires some manual correction:

```
$ cd /usr/local/lib/python2.5/site-packages
$ mkdir scipy
$ touch scipy/__init__.py
$ mv ndimage scipy
```

1.4 Troubleshooting

The PyMS is essentially a python library (a 'package' in python parlance, which consists of several 'sub-packages'), which for some functionality depends on other python libraries, such as NumPy, pycdf, and Pycluster. The most likely problem with PyMS installation is a problem with installing one of the PyMS dependencies.

1.4.1 Pycdf import error

On Red Hat Linux 5 the SELinux is enabled by default, and this causes the following error while trying to import properly installed pycdf:

```
$ python
Python 2.5.2 (r252:60911, Nov  5 2008, 16:25:39)
[GCC 4.1.1 20070105 (Red Hat 4.1.1-52)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pycdf
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.5/site-packages/pycdf/__init__.py", line 22, in <module>
    from pycdf import *
  File "/usr/local/lib/python2.5/site-packages/pycdf/pycdf.py", line 1096, in <module>
    import pycdfext as _C
  File "/usr/local/lib/python2.5/site-packages/pycdf/pycdfext.py", line 5, in <module>
    import _pycdfext
ImportError: /usr/local/lib/python2.5/site-packages/pycdf/_pycdfext.so:
    cannot restore segment prot after reloc: Permission denied
```

This problem is removed simply by disabling SELinux (login as 'root', open the menu Administration → Security Level and Firewall, tab SELinux, change settings from 'Enforcing' to 'Disabled').

This problem is likely to occur on Red Hat Linux derivative distributions such as CentOS.

1.5 PyMS tutorial and examples

A tutorial illustrating various PyMS features in provided in subsequent chapter of this User Guide. The commands executed interactively are grouped together by example, and provided as Python scripts in the project 'pyms-test' (this is a Google code project, similar to the project 'pyms' which contains the PyMS source code).

The setup used in the examples below is as follows. The projects 'pyms', 'pyms-test', 'pyms-docs', and 'data' are all in the same directory, '/x/PyMS'. In the project 'pyms-test' there is a directory corresponding to each example coded with the example number (ie. `pyms-test/21a/` corresponds to Example 1a in Chapter 2).

In each example directory, there is a script named 'proc.py' which contains the commands given in the example. Provided that the paths to 'pyms' and 'pyms-data' are set properly, these scripts could be run with the following command:

```
$ python proc.py
```

Before running each example the Python interpreter was made aware of the PyMS location with the following commands:

```
import sys
sys.path.append("/x/PyMS")
```

For brevity these commands will not be shown in the examples below, but they are included in 'pyms-test' example scripts. The above path may need to be adjusted to match your own directory structure.

All data files (raw data files, peak lists etc.) used in the example below can be found at
`http://bioinformatics.bio21.unimelb.edu.au/pyms/data/`
and are assumed to be located in the 'data' directory.

GC-MS Raw Data Model

2.1 Introduction

PyMS can read gas chromatography-mass spectrometry (GC-MS) data stored in Analytical Data Interchange for Mass Spectrometry (ANDI-MS),[2] and Joint Committee on Atomic and Molecular Physical Data (JCAMP-DX)[3] formats. These formats are essentially recommendations, and it is up to individual vendors of mass spectrometry processing software to implement "export to ANDI-MS" or "export to JCAMP-DX" features in their software. It is also possible to get third party converters. The information contained in the exported data files can vary significantly, depending on the instrument, vendor's software, or conversion utility.

For PyMS, the minimum set of assumptions about the information contained in the data file are:

- The data contain the m/z and intensity value pairs across a scan.
- Each scan has a retention time.

Internally, PyMS stores the raw data from ANDI files or JCAMP files as a GCMS_data object.

2.2 Reading the raw GC-MS data

2.2.1 Reading JCAMP GC-MS data

[*This example is in pyms-test/20a*]

The PyMS package pyms.GCMS.IO.JCAMP provides capabilities to read the raw GC-MS data stored in the JCAMP-DX format.

The file 'gc01_0812_066.jdx' (located in 'data') is a GC-MS experiment converted from Agilent ChemStation format to JCAMP format using File Translator Pro.[4] This file can be loaded in Python as follows:

[2] ANDI-MS was developed by the Analytical Instrument Association.
[3] JCAMP-DX is maintained by the International Union of Pure and Applied Chemistry.
[4] ChemSW, Inc.

```
>>> from pyms.GCMS.IO.JCAMP.Function import JCAMP_reader
>>> jcamp_file = "/x/PyMS/data/gc01_0812_066.jdx"
>>> data = JCAMP_reader(jcamp_file)
 -> Reading JCAMP file '/x/PyMS/pyms-data/gc01_0812_066.jdx'
>>>
```

The above command creates the object 'data' which is an *instance* of the class GCMS_data.

2.2.2 Reading ANDI GC-MS data

[*This example is in pyms-test/20b*]

The PyMS package pyms.GCMS.IO.ANDI provides capabilities to read the raw GC-MS data stored in the ANDI-MS format.

The file 'gc01_0812_066.cdf' (located in 'data') is a GC-MS experiment converted to ANDI-MS format from Agilent ChemStation (from the same data as in example 20a above). This file can be loaded as follows:

```
>>> from pyms.GCMS.IO.ANDI.Function import ANDI_reader
>>> ANDI_file = "/x/PyMS/data/gc01_0812_066.cdf"
>>> data = ANDI_reader(ANDI_file)
 -> Reading netCDF file '/x/PyMS/pyms-data/gc01_0812_066.cdf'
>>>
```

The above command creates the object 'data' which is an *instance* of the class GCMS_data.

2.3 A GCMS_data object

2.3.1 Methods

[*The following examples are the same in pyms-test/20a and pyms-test/20b*]

The object 'data' (from the two previous examples) stores the raw data as a *GCMS_data* object. Within the GCMS_data object, raw data are stored as a list of *Scan* objects and a list of retention times. There are several methods available to access data and attributes of the GCMS_data and Scan objects.

The GCMS_data object's methods relate to the raw data. The main properties relate to the masses, retention times and scans. For example, the minimum and maximum mass from all of the raw data can be returned by the following:

```
>>> data.get_min_mass()
>>> data.get_max_mass()
```

A list of all retention times can be returned by:

```
>>> time = data.get_time_list()
```

The index of a specific retention time (in seconds) can be returned by:

```
>>> data.get_index_at_time(400.0)
```

Note that this returns the index of the retention time in the data closest to the given retention time of 400.0 seconds.

The method `get_tic()` returns a total ion chromatogram (TIC) of the data as an IonChromatogram object:

```
>>> tic = data.get_tic()
```

The IonChromatogram object is explained in a later chapter.

2.3.2 A Scan data object

A Scan object contains a list of masses and a corresponding list of intensity values from a single mass-spectrum scan in the raw data. Typically only non-zero (or non-threshold) intensities and corresponding masses are stored in the raw data.

[*The following examples are the same in pyms-test/20a and pyms-test/20b*]

A list of all the raw Scan objects can be returned by:

```
>>> scans = data.get_scan_list()
```

A list of all masses in a scan (e.g. the 1st scan) is returned by:

```
>>> scans[0].get_mass_list()
```

A list of all corresponding intensities in a scan is returned by:

```
>>> scans[0].get_intensity_list()
```

The minimum and maximum mass in an individual scan (e.g. the 1st scan) are returned by:

```
>>> scans[0].get_min_mass()
>>> scans[0].get_max_mass()
```

2.3.3 Exporting data and printing information about a data set

[*This example is in pyms-test/20c*]

Often it is of interest to find out some basic information about the data set, e.g. the number of scans, the retention time range, and m/z range and so on. The GCMS_data class provides a method info() that can be used for this purpose.

```
>>> from pyms.GCMS.IO.ANDI.Function import ANDI_reader
>>> andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
>>> data = ANDI_reader(andi_file)
 -> Reading netCDF file '/x/PyMS/data/gc01_0812_066.cdf'
>>> data.info()
 Data retention time range: 5.093 min -- 66.795 min
 Time step: 0.375 s (std=0.000 s)
 Number of scans: 9865
 Minimum m/z measured: 50.000
 Maximum m/z measured: 599.900
 Mean number of m/z values per scan: 56
 Median number of m/z values per scan: 40
>>>
```

To export the entire raw data to a file, use the method write():

```
>>> data.write("output/data")
 -> Writing intensities to 'output/data.I.csv'
 -> Writing m/z values to 'output/data.mz.csv'
```

This method takes the string ("output/data", in this example) and writes two CSV files. One has extention ".I.csv" and contains the intensities ("output/data.I.csv" in this example), and the other has the extension ".mz" and contains the corresponding table of m/z value ("output/data.mz.csv" in this example). In general these are not two-dimensional matrices, because different scans may have different number of m/z values recorded.

2.3.4 Comparing two GC-MS data sets

[*This example is in pyms-test/20d*]

Occasionally it is useful to compare two data sets. For example, one may want to check the consistency between the data set exported in netCDF format from the manufacturer's software, and the JCAMP format exported from a third party software.

For example:

```
>>> from pyms.GCMS.IO.JCAMP.Function import JCAMP_reader
>>> from pyms.GCMS.IO.ANDI.Function import ANDI_reader
>>> andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
>>> jcamp_file = "/x/PyMS/data/gc01_0812_066.jdx"
>>> data1 = ANDI_reader(andi_file)
 -> Reading netCDF file '/x/PyMS/data/gc01_0812_066.cdf'
>>> data2 = JCAMP_reader(jcamp_file)
 -> Reading JCAMP file '/x/PyMS/data/gc01_0812_066.jdx'
```

To compare the two data sets:

```
>>> from pyms.GCMS.Function import diff
>>> diff(data1,data2)
 Data sets have the same number of time points.
   Time RMSD: 1.80e-13
 Checking for consistency in scan lengths ... OK
 Calculating maximum RMSD for m/z values and intensities ...
   Max m/z RMSD: 1.03e-05
   Max intensity RMSD: 0.00e+00
```

If the data is not possible to compare, for example because of different number of scans, or inconsistent number of m/z values in between two scans, diff() will report the difference. For example:

```
>>> data2.trim(begin=1000,end=2000)
Trimming data to between 1000 and 2000 scans
>>> diff(data1,data2)
 -> The number of retention time points different.
 First data set: 9865 time points
 Second data set: 1001 time points
 Data sets are different.
```

GC-MS data derived objects

In this chapter the methods for converting the raw GC-MS data to an Intensity Matrix object are illustrated.

In the raw GC-MS data, consecutive scans do not necessarily contain the same mass per charge (mass) values. For data processing, it is often necessary to convert the data to a matrix with a set number of masses and scans. In PyMS there are functions to explicitly convert the raw mass values to consistent values across all scans.

3.1 IntensityMatrix Object

The general scheme for converting raw mass values is to bin intensity values based on the interval the corresponding mass belongs to. The general procedure is as follows:

- Set the interval between bins, lower and upper bin boundaries
- Calculate the number of bins to cover the range of all masses.
- Centre the first bin at the minimum mass found for all the raw data.
- Sum intensities whose masses are in a given bin.

A mass, m, is considered to belong to a bin when $c - l \leq m < c + u$, where c is the centre of the bin, l is the lower boundary and u is the upper boundary of the bin. The default bin interval is one with a lower and upper boundary of ± 0.5.

A function to bin masses to the nearest integer is also available. The default bin interval is one with a lower boundary of -0.3 and upper boundary of +0.7 (as per the NIST library).

3.1.1 Build intensity matrix

[*This example is in pyms-test/30a*]

An intensity matrix on the raw GC-MS data can be built using the following function. First the raw data is imported as before.

```
>>> from pyms.GCMS.IO.JCAMP.Function import JCAMP_reader
>>> jcamp_file = "/x/PyMS/data/gc01_0812_066.jdx"
>>> data = JCAMP_reader(jcamp_file)
 -> Reading JCAMP file '/x/PyMS/pyms-data/gc01_0812_066.jdx'
>>>
```

Then the data can be converted to an intensity matrix using the functions available in "pyms.GCMS.Function", namely `build_intensity_matrix()` and `build_intensity_matrix_i()`.

The default operation of `build_intensity_matrix()` is to use a bin interval of one and treat the masses as floating point numbers. The default intensity matrix can be built as follows:

```
>>> from pyms.GCMS.Function import build_intensity_matrix
>>> im = build_intensity_matrix(data)
```

The size as the number of scans and the number of bins is returned by:

```
>>> im.get_size()
```

There are 9865 scans and 551 bins in this example.

The raw masses have been binned into new mass units based on the minimum mass in the raw data and the bin size. A list of the new masses are returned by:

```
>>> masses = im.get_mass_list()
```

It is also possible to search for a particular mass, by finding the index of the binned mass closest to the desired mass. For example, the index of the closest binned mass to a mass of 73.3 m/z is returned by:

```
>>> index = im.get_index_of_mass(73.3)
```

The value of the closest mass can be returned by:

```
>>> print im.get_mass_at_index(index)
```

A mass of 73.0 is returned in this example.

3.1.2 Build intensity matrix parameters

[*This example is in pyms-test/30b*]

The bin interval can be set to values other than one. For example, the bin interval can be set to 0.5. The boundaries can also be adjusted. In this example, to fit the 0.5 bin interval, the upper and lower boundaries are set to ± 0.25.

```
im = build_intensity_matrix(data, 0.5, 0.25, 0.25)
```

The size of the intensity matrix will reflect the change in the number of bins.

```
>>> im.get_size()
```

There are 9865 scans (as before) and 1101 bins in this example.

The index and binned mass of the mass closest to 73.3 should also reflect the different binning.

```
>>> index = im.get_index_of_mass(73.3)
>>> print im.get_mass_at_index(index)
```

A mass of 73.5 is returned in this example.

3.1.3 Build integer mass intensity matrix

[*This example is in pyms-test/30c*]

It is also possible to build an intensity matrix with integer masses and a bin interval of one. The default range for the binning is -0.3 and +0.7 mass units. The function is imported from "pyms.GCMS.Function".

```
>>> from pyms.GCMS.Function import build_intensity_matrix_i
>>> im = build_intensity_matrix_i(data)
```

The masses are now all integers.

```
>>> index = im.get_index_of_mass(73.3)
>>> print im.get_mass_at_index(index)
```

A mass of 73 is returned in this example.

The lower and upper bounds can be adjusted by `build_intensity_matrix_i(data, lower, upper)`.

3.2 MassSpectrum Object

[*This example is in pyms-test/31*]

A MassSpectrum object contains two attributes, `mass_list` and `mass_spec`, a list of the mass values and corresponding intensities, respectively. MassSpectrum is returned by the IntensityMatrix method of `get_ms_at_index(index)`.

For example, the properties of the first MassSpectrum object of an IntensityMatrix, *im*, can be investigated by;

```
>>> ms = im.get_ms_at_index(0)
>>> print len(ms)
>>> print len(ms.mass_list)
>>> print len(ms.mass_spec)
```

The length of all attributes should be the same.

3.3 IonChromatogram Object

[*This example is in pyms-test/31*]

An IonChromatogram object is a one dimensional vector containing mass intensities as a function of retention time. This can can be either m/z channel intensities (for example, the ion chromatogram at m/z = 73), or cumulative intensities over all measured m/z (TIC).

An IonChromatogram for the TIC and a given mass or index can be obtained by;

```
>>> tic = data.get_tic()
>>> ic = im.get_ic_at_index(0)
>>> ic = im.get_ic_at_mass(73)
```

This will return, respectively: the TIC; the ion chromatogram of the first mass; and the ion chromatogram of the mass closest to 73.

An ion chromatogram object has a method `is_tic()` which returns True if the ion chromatogram is a TIC, False otherwise:

```
>>> print "'tic' is a TIC:", tic.is_tic()
'tic' is a TIC: True
>>> print "'ic' is a TIC:",ic.is_tic()
'ic' is a TIC: False
```

3.3.1 Writing IonChromatogram data to a file

[*This example is in pyms-test/31*]

The method `write()` of IonChromatogram object allows one to save the ion chromatogram object to a file:

```
>>> tic.write("output/tic.dat", minutes=True)
>>> ic.write("output/ic.dat", minutes=True)
```

The flag minutes=True indicates that retention time will be saved in minutes. The ion chromatogram object saved with with the `write` method is a plain ASCII file which contains a pair of (retention time, intensity) per line.

```
$ head tic.dat
  5.0930 2.222021e+07
  5.0993 2.212489e+07
  5.1056 2.208650e+07
  5.1118 2.208815e+07
  5.1181 2.200635e+07
  5.1243 2.200326e+07
  5.1306 2.202363e+07
  5.1368 2.198357e+07
  5.1431 2.197408e+07
  5.1493 2.193351e+07
```

3.4 Saving data

[*This example is in pyms-test/32*]

A matrix of intensity values can be saved to a file with the function `save_data()` from `pyms.Utils.IO`. A matrix of intensity values can be returned from an IntensityMatrix with the method `get_matrix_list()`. For example,

```
>>> from pyms.Utils.IO import save_data
>>> mat = im.get_matrix_list()
>>> save_data("output/im.dat", mat)
```

It is also possible to save the list of masses (from `im.get_mass_list()`) and the list of retention times (from `im.get_time_list()`) using the `save_data()` function. For convenience, the intensity values, mass list and time list, can be saved with the method `export_ascii()`. For example,

```
>>> im.export_ascii("output/data")
```

will create "data.im.dat", "data.mz.dat", and "data.rt.dat" where these are the intensity matrix, retention time vector, and m/z vector. By default the data is saved as space separated data with a ".dat" extension. It is also possible to save the data as comma separated data with a ".csv" extension by the command "`im.export_ascii("output/data", "csv")`".

Additionally, the entire IntensityMatrix can be exported as LECO CSV. This is useful for import into other analytical software packages. The format is a header line specifying the column heading information as: "scan, retention time, mass1, mass2, ⋮", and then each row as the intensity data.

```
>>> im.export_leco_csv("output/data_leco.csv")
```

Data filtering

4.1 Introduction

In this chapter filtering techniques that allow pre-processing of GC-MS data for analysis and comparison to other pre-processed GC-MS data are covered.

4.2 Time strings

Before considering the filtering techniques, the mechanism for representing retention times is outlined here.

A time string is the specification of a time interval, that takes the format 'NUMBERs' or 'NUMBERm' for time interval in seconds or minutes. For example, these are valid time strings: '10s' (10 seconds) and '0.2m' (0.2 minutes).

4.3 Intensity Matrix resizing

Once an IntensityMatrix has been constructed from the raw GC-MS data, the entries of the matrix can be modified. These modifications can operate on the entire matrix, or individual masses or scans.

4.3.1 Retention time range

[*This example is in pyms-test/40a*]

A basic operation on the GC-MS data is to select a specific time range for processing. In PyMS, any data outside the chosen time range is discarded. The `trim()` method operates on the raw data, so any subsequent processing only refers to the trimmed data.

Given a previously loaded raw GC-MS data file, *data*, the data can be trimmed to specific scans;

```
>>> data.trim(1000, 2000)
>>> print data.info()
```

or specific retention times (in "seconds" or "minutes");

```
>>> data.trim("6.5m", "21m")
>>> print data.info()
```

4.3.2 Mass spectrum range and entries

[*This example is in pyms-test/40b*]

An IntensityMatrix object has a set mass range and interval that is derived from the data at the time of building the intensity matrix. The range of mass values can be modified. This is done, primarily, to ensure that the range of masses used are consistent when comparing samples.

Given a previously loaded raw GC-MS data file that has been converted into an IntensityMatrix, *im*, the mass range can be "cropped" to a new (smaller) range;

```
>>> im.crop_mass(60, 400)
>>> print im.get_min_mass(), im.get_max_mass()
```

It is also possible to set all intensities for a given mass to zero. This is useful for ignoring masses associated with sample preparation. The mass can be "nulled" via;

```
>>> data.null_mass(73)
>>> print sum(im.get_ic_at_mass(73).get_intensity_array())
```

4.4 Noise smoothing

The purpose of noise smoothing is to remove high-frequency noise from data, and thereby increase the contribution of the signal relative to the contribution of the noise.

4.4.1 Window averaging

[*This example is in pyms-test/41a*]

A simple approach to noise smoothing is moving average window smoothing. In this approach the window of a fixed size ($2N + 1$ points) is moved across the ion chromatogram, and the intensity value at each point is replaced with the mean intensity calculated over the window size. The example below illustrates smoothing of TIC by window averaging.

Load the data and get the TIC:

```
>>> andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
>>> data = ANDI_reader(andi_file)
 -> Reading netCDF file '/x/PyMS/data/gc01_0812_066.cdf'
>>> tic = data.get_tic()
```

Apply the mean window smoothing with the 5-point window:

```
from pyms.Noise.SavitzkyGolay import window_smooth
tic1 = window_smooth(tic, window=5)
 -> Window smoothing (mean): the wing is 2 point(s)
```

Apply the median window smoothing with the 5-point window:

```
>>> tic2 = window_smooth(tic, window=5, median=True)
 -> Window smoothing (median): the wing is 2 point(s)
```

Apply the mean windows smoothing, but specify the window as a time string (in this example, 7 seconds):

```
>>> tic3 = window_smooth(tic, window='7s')
 -> Window smoothing (mean): the wing is 9 point(s)
```

Time strings are explained in the Section 4.2.

4.4.2 Savitzky–Golay noise filter

[*This example is in pyms-test/41b*]

A more sophisticated noise filter is the Savitzky-Golay filter. Given the data loaded as above, this filter can be applied as follows:

```
>>> from pyms.Noise.SavitzkyGolay import savitzky_golay
>>> tic1 = savitzky_golay(tic)
 -> Applying Savitzky-Golay filter
      Window width (points): 7
      Polynomial degree: 2
```

In this example the default parameters were used.

4.5 Baseline correction

[*This example is in pyms-test/42*]

Baseline distortion originating from instrument imperfections and experimental setup is often observed in mass spectrometry data, and off-line baseline correction is often an important step in data pre-processing. There are many approaches for baseline correction. One advanced approach is based top-hat transform developed in mathematical morphology [3], and used extensively in digital image processing for tasks such as image enhancement. Top-hat baseline correction was previously applied in proteomics based mass spectrometry [4].

PyMS currently implements only top-hat baseline corrector, using the SciPy package 'ndimage'. For this feature to be available either SciPy (Scientific Tools for Python [5]) must be installed, or the local versions of scipy's ndimage must be installed. For the SciPy/ndimage installation instructions please see the section 1.2.6.

Application of the top-hat baseline corrector requires the size of the structural element to be specified. The structural element needs to be larger than the features one wants to retain in the spectrum after the top-hat transform. In the example below, the top-hat baseline corrector is applied to the TIC of the data set 'gc01_0812_066.cdf', with the structural element of 1.5 minutes:

```
>>> from pyms.GCMS.IO.ANDI.Function import ANDI_reader
>>> andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
>>> data = ANDI_reader(andi_file)
 -> Reading netCDF file '/x/PyMS/data/gc01_0812_066.cdf'
>>> tic = data.get_tic()
>>> from pyms.Noise.SavitzkyGolay import savitzky_golay
>>> tic1 = savitzky_golay(tic)
 -> Applying Savitzky-Golay filter
      Window width (points): 7
      Polynomial degree: 2
>>> from pyms.Baseline.TopHat import tophat
>>> tic2 = tophat(tic1, struct="1.5m")
 -> Top-hat: structural element is 239 point(s)
>>> tic.write("output/tic.dat",minutes=True)
>>> tic1.write("output/tic_smooth.dat",minutes=True)
>>> tic2.write("output/tic_smooth_bc.dat",minutes=True)
```

In the interactive session shown above, the data set if first loaded, Savitzky-Golay smoothing was applied, followed by baseline correction. Finally the original, smoothed, and smoothed and baseline corrected TIC were saved in the directory 'output/'.

4.6 Pre-processing the IntensityMatrix

[*This example is in pyms-test/43*]

The entire noise smoothing and baseline correction can be applied to each ion chromatogram in the intensity matrix;

```
>>> jcamp_file = "/x/PyMS/data/gc01_0812_066.jdx"
>>> data = JCAMP_reader(jcamp_file)
>>> im = build_intensity_matrix(data)
>>> n_scan, n_mz = im.get_size()
>>> for ii in range(n_mz):
...     print "Working on IC#", ii+1
...     ic = im.get_ic_at_index(ii)
...     ic_smooth = savitzky_golay(ic)
...     ic_bc = tophat(ic_smooth, struct="1.5m")
...     im.set_ic_at_index(ii, ic_bc)
...
```

The resulting IntensityMatrix object can be "dumped" to a file for later retrieval. There are general perpose object file handling methods in `pyms.Utils.IO`. For example;

```
>>> from pyms.Utils.IO import dump_object
>>> dump_object(im, "output/im-proc.dump")
```

Peak detection and representation

5.1 Peak Object

Fundamental to GC-MS analysis is the identification of individual components of the sample mix. The basic component unit is represented as a signal Peak. In PyMS a signal peak is represented as 'Peak' object defined in `pyms.Peak.Class`, and functions to detect the peaks are also provided (discussed at the end of the chapter).

A peak object stores a minimal set of information about a signal peak, namely, the retention time at which the peak apex occurs and the mass spectra at the apex. Additional information, such as, peak width, TIC and individual ion areas can be filtered from the GC-MS data and added to the Peak object information.

5.1.1 Creating a Peak Object

[*This example is in pyms-test/50*]

A peak object can be created for a scan at a given retention time by providing the retention time (in minutes or seconds) and the MassSpectrum object of the scan. For example, first a file is loaded and an IntensityMatrix, *im*, built, then a MassSpectrum, *ms*, can be selected at a given time (31.17 minutes in this example).

```
>>> from pyms.GCMS.Function import build_intensity_matrix_i
>>> from pyms.GCMS.IO.ANDI.Function import ANDI_reader
>>> andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
>>> data = ANDI_reader(andi_file)
>>> im = build_intensity_matrix_i(data)
>>> index = im.get_index_at_time(31.17*60.0)
>>> ms = im.get_ms_at_index(index)
```

Now a Peak object can be created for the given retention time and MassSpectrum.

```
>>> from pyms.Peak.Class import Peak
```

```
>>> peak = Peak(31.17, ms, minutes=True)
```

By default the retention time is assumed to be in seconds. The parameter `minutes` can be set to True if the retention time is given in minutes. As a matter of convention, PyMS internally stores retention times in seconds, so the `minutes` parameter ensures the input and output of the retention time are in the same units.

5.1.2 Peak Object properties

[*This example is in pyms-test/50*]

The retention time of the peak can be returned with `get_rt()`. The retention time is returned in seconds with this method. The mass spectrum can be returned with `get_mass_spectrum()`.

The Peak object constructs a unique identification (UID) based on the spectrum and retention time. This helps in managing lists of peaks (covered in the next chapter). The UID can be returned with `get_UID()`. The format of the UID is the masses of the two most abundant ions in the spectrum, the ratio of the abundances of the two ions, and the retention time (in the same units as given when the Peak object was created). The format is `Mass1-Mass2-Ratio-RT`. For example,

```
>>> print peak.get_rt()
1870.2
>>> print peak.get_UID()
319-73-74-31.17
```

5.1.3 Modifying a Peak Object

[*This example is in pyms-test/51*]

The Peak object has methods for modifying the mass spectrum. The mass range can be cropped to a smaller range with `crop_mass()`, and the intensity values for a single ion can be set to zero with `null_mass()`. For example, the mass range can be set from 60 to 450 m/z, and the ions related to sample preparation can be ignored by setting their intensities to zero;

```
>>> peak.crop_mass(60, 450)
>>> peak.null_mass(73)
>>> peak.null_mass(147)
```

The UID is automatically updated to reflect the changes;

```
>>> print peak.get_UID()
319-205-54-31.17
```

It is also possible to change the peak mass spectrum by calling the method `set_mass_spectrum()`.

5.2 Peak detection

The general use of a Peak object is to extract them from the GC-MS data and build a list of peaks. In PyMS, the function for peak detection is based on the method of Biller and Biemann (1974)[6]. The basic process is to find all maximising ions in a pre-set window of scans, for a given scan. The ions that maximise at a given scan are taken to belong to the same peak.

The function is `BillerBiemann()` in `pyms.Deconvolution.BillerBiemann.Function`. The function has parameters for the window width for detecting the local maxima (`points`), and the number of `scans` across which neighbouring, apexing, ions are combined and considered as belonging to the same peak. The number of neighbouring scans to combine is related to the likelyhood of detecting a peak apex at a single scan or several neighbouring scans. This is more likely when there are many scans across the peak. It is also possible, however, when there are very few scans across the peak. The scans are combined by taking all apexing ions to have occurred at the scan that had to greatest TIC prior to combining scans.

5.2.1 Sample processing and Peak detection

[*This example is in pyms-test/52*]

The process for detecting peaks is to pre-process the data by performing noise smoothing and baseline correction on each ion (as in *pyms-test/52*). The first steps then are;

```
>>> from pyms.GCMS.IO.ANDI.Function import ANDI_reader
>>> from pyms.GCMS.Function import build_intensity_matrix
>>> from pyms.Noise.SavitzkyGolay import savitzky_golay
>>> from pyms.Baseline.TopHat import tophat
>>>
>>> andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
>>> data = ANDI_reader(andi_file)
>>>
>>> im = build_intensity_matrix(data)
>>> n_scan, n_mz = im.get_size()
>>>
>>> for ii in range(n_mz):
...     ic = im.get_ic_at_index(ii)
...     ic_smooth = savitzky_golay(ic)
...     ic_bc = tophat(ic_smooth, struct="1.5m")
...     im.set_ic_at_index(ii, ic_bc)
...
```

Now the Biller and Biemann based technique can be applied to detect peaks.

```
>>> from pyms.Deconvolution.BillerBiemann.Function import BillerBiemann
```

```
>>> peak_list = BillerBiemann(im)
>>> print len(peak_list)
9845
```

Note that this is nearly as many peaks as there are scans in the data (9865 scans). This is due to noise and the simplicity of the technique.

The number of detected peaks can be constrained by the selection of better parameters. Parameters can be determined by counting the number of points across a peak, and examining where peaks are found. For example, the peak list can be found with the parameters of a window of 9 points and by combining 2 neighbouring scans if they apex next to each other;

```
>>> peak_list = BillerBiemann(im, points=9, scans=2)
>>> print len(peak_list)
3698
```

The number of detected peaks has been reduced, but there are still many more than would be expected from the sample. Functions to filter the peak list are covered in the next section.

5.3 Filtering Peak Lists

[*This example is in pyms-test/53*]

There are two functions to filter the list of Peak objects. The first, `rel_threshold()`, modifies the mass spectrum stored in each peak so any intensity that is less than a given percentage of the maximum intensity for the peak is removed. The second, `num_ions_threshold()` removes any peak that has less than a given number of ions above a given threshold. Once the peak list has been constructed, the filters can be applied by;

```
>>> from pyms.Deconvolution.BillerBiemann.Function import \
... rel_threshold, num_ions_threshold
>>> pl = rel_threshold(peak_list, percent=2)
>>> new_peak_list = num_ions_threshold(pl, n=3, cutoff=10000)
>>> print len(new_peak_list)
146
```

The number of detected peaks is now more realistic of what would be expected in the test sample.

5.4 Peak area estimation

[*This example is in pyms-test/54*]

The Peak object does not contain any information about the width or area of the peak when it is created. This information can be added after the instantiation of a Peak object. The area of the peak can be set by the `set_area()` method of the peak object.

The peak area can by obtained by the `peak_sum_area()` function in `pyms.Peak.Function`. The function determines the total area as the sum of the ion intensities for all masses that apex at the given peak. To calculate the peak area of a single mass, the intensities are added from the apex of the mass peak outwards. Edge values are added until the following conditions are met: the added intensity adds less than 0.5% to the accumulated area; or the added intensity starts increasing (i.e. when the ion is common to co-eluting compounds). To avoid noise effects, the edge value is taken at the midpoint of three consecutive edge values.

Given a list of peaks, areas can be determined and added as follows:

```
>>> from pyms.Peak.Function import peak_sum_area
>>> for peak in peak_list:
...     area = peak_sum_area(intensity_matrix, peak)
...     peak.set_area(area)
...
```

Peak alignment by dynamic programming

PyMS provides functions to align GC-MS peaks by dynamic programming [7]. The peak alignment by dynamic programming uses both peak apex retention time and mass spectra. This information is determined from the raw GC-MS data by applying a series of processing steps to produce data that can then be aligned and used for statistical analysis. The details are described in this chapter.

6.1 Preparation of multiple experiments for peak alignment by dynamic programming

6.1.1 Creating an Experiment

[*This example is in pyms-test/60*]

Before aligning peaks from multiple experiments, the peak objects need to be created and encapsulated into PyMS experiment objects. During this process it is often useful to pre-process the peaks in some way, for example to null certain m/z channels and/or to select a certain retention time range.

To capture the data and related information prior to peak alignment, an `Experiment` object is used. The `Experiment` object is defined in `pyms.Experiment.Class`.

The procedure is to proceed as described in the previous chapter. Namely: read a file; bin the data into fixed mass values; smooth the data; remove the baseline; deconvolute peaks; filter the peaks; set the mass range; remove uninformative ions; and estimate peak areas. The process is given in the following program listing.

```
01  import sys, os
02  sys.path.append("/x/PyMS")
03
04  from pyms.GCMS.IO.ANDI.Function import ANDI_reader
05  from pyms.GCMS.Function import build_intensity_matrix_i
```

```
06 from pyms.Noise.SavitzkyGolay import savitzky_golay
07 from pyms.Baseline.TopHat import tophat
08 from pyms.Peak.Class import Peak
09 from pyms.Peak.Function import peak_sum_area
10
11 from pyms.Deconvolution.BillerBiemann.Function import BillerBiemann, \
12     rel_threshold, num_ions_threshold
13
14 # deconvolution and peak list filtering parameters
15 points = 9; scans = 2; n = 3; t = 3000; r = 2;
16
17 andi_file = "/x/PyMS/data/a0806_077.cdf"
18
19 data = ANDI_reader(andi_file)
20
21 # integer mass
22 im = build_intensity_matrix_i(data)
23
24 # get the size of the intensity matrix
25 n_scan, n_mz = im.get_size()
26
27 # smooth data
28 for ii in range(n_mz):
29     ic = im.get_ic_at_index(ii)
30     ic1 = savitzky_golay(ic)
31     ic_smooth = savitzky_golay(ic1)
32     ic_base = tophat(ic_smooth, struct="1.5m")
33     im.set_ic_at_index(ii, ic_base)
34
35 # do peak detection on pre-trimmed data
36
37 # get the list of Peak objects
38 pl = BillerBiemann(im, points, scans)
39
40 # trim by relative intensity
41 apl = rel_threshold(pl, r)
42
43 # trim by threshold
44 peak_list = num_ions_threshold(apl, n, t)
45
46 print "Number of Peaks found:", len(peak_list)
47
48 # ignore TMS ions and set mass range
49 for peak in peak_list:
```

```
50      peak.crop_mass(50,540)
51      peak.null_mass(73)
52      peak.null_mass(147)
53      # find area
54      area = peak_sum_area(im, peak)
55      peak.set_area(area)
56
```

The resulting list of peaks can now be stored as an `Experiment` object.

```
from pyms.Experiment.Class import Experiment
from pyms.Experiment.IO import store_expr

# create an experiment
expr = Experiment("a0806_077", peak_list)

# set time range for all experiments
expr.sele_rt_range(["6.5m", "21m"])

store_expr("output/a0806_077.expr", expr)
```

Once an experiment has been defined, it is possible to limit the peak list to a desired range using `sele_rt_range()`. The resulting experiment object can then be stored for later alignment.

6.1.2 Multiple Experiments

[*This example is in pyms-test/61a*]

This example considers the preparation of three GC-MS experiments for peak alignment. The experiments are named 'a0806_077', 'a0806_078', 'a0806_079', and represent separate GC-MS sample runs from the same biological sample.

The procedure is the same as above, and repeated for each experiment. For example:

```
...
# define path to data files
base_path = "/x/PyMS/data/"

# define experiments to process
expr_codes = [ "a0806_077", "a0806_078", "a0806_079" ]

# loop over all experiments
for expr_code in expr_codes:
```

```
    print "Processing", expr_code

    # define the names of the peak file and the corresponding ANDI-MS file
    andi_file = os.path.join(base_path, expr_code + ".cdf")
...

...
    # create an experiment
    expr = Experiment(expr_code, peak_list)

    # use same time range for all experiments
    expr.sele_rt_range(["6.5m", "21m"])

    store_expr("output/"+expr_code+".expr", expr)
```

[*This example is in pyms-test/61b*]

The previous set of data all belong to the same experimental condition. That is, they represent one group and any comparison between the data is a within group comparison. For the original experiment, another set of GC-MS data was collected for a different experimental condition. This group must also be stored as a set of experiments, and can be used for between group comparison.

The experiments are named 'a0806_140', 'a0806_141', 'a0806_142', and are processed and stored as above (see pyms-test/61b).

6.2 Dynamic programming alignment of peak lists from multiple experiments

- *This example is in pyms-test/62*
- *This example uses the subpackage pyms.Peak.List.DPA, which in turn uses the Python package 'Pycluster'. For 'Pycluster' installation instructions see the Section 1.2.5.*

In this example the experiments 'a0806_077', 'a0806_078', and 'a0806_079' prepared in pyms-test/61a will be aligned, and therefore the script pyms-test/61a/proc.py must be run first, to create the files 'a0806_077.expr', 'a0806_078.expr', 'a0806_079.expr' in the directory pyms-test/61a/output/. These files contain the post-processed peak lists from the three experiments.

The input script required for running the dynamic programming alignment is given below.

```
"""proc.py
"""
```

```
import sys, os
sys.path.append("/x/PyMS/")

from pyms.Experiment.IO import load_expr
from pyms.Peak.List.DPA.Class import PairwiseAlignment
from pyms.Peak.List.DPA.Function import align_with_tree, exprl2alignment

# define the input experiments list
exprA_codes = [ "a0806_077", "a0806_078", "a0806_079" ]

# within replicates alignment parameters
Dw = 2.5  # rt modulation [s]
Gw = 0.30 # gap penalty

# do the alignment
print 'Aligning expt A'
expr_list = []
expr_dir = "../61a/output/"
for expr_code in exprA_codes:
    file_name = os.path.join(expr_dir, expr_code + ".expr")
    expr = load_expr(file_name)
    expr_list.append(expr)
F1 = exprl2alignment(expr_list)
T1 = PairwiseAlignment(F1, Dw, Gw)
A1 = align_with_tree(T1, min_peaks=2)

A1.write_csv('output/rt.csv', 'output/area.csv')
```

The explanation of the task performed by the input script is given below:

- Lines 16 and 17: input parameters for the alignment by dynamic programming are defined. Dw is the retention time moduleation in seconds, while Gw is the gap penalty. These parameters are explained in detail in [7]
- line 21: The list of experiments is loaded into the variable named E1. E1 is simply a Python list containing three expaeriment objects as elements.
- Line 22: The list of experiments is converted into the list of alignments. Each experiment object is converted into the "alignment" object. In this case the alignment object contains only a single experiment, and is not really an alignment at all (this special case is called 1-alignment). The variable F1 is simply a Python list containing three alignment objects.
- Line 23: all possible pairwise alignments (2-alignments) are calculated from the list of 1-alignments. PairwiseAlignment() is a class, and T1 is an object which contains the dendrogram tree that maps the similarity relationship between the input 1-alignments, and also 1-alignments themselves.

- Line 24: The function align_with_tree() takes the object T1 and aligns the individual alignment supplied with it according the guide tree. In this case, the individual alignment are three 1-alignments, and the function align_with_tree() first creates a 2-alignment from the two most similar 1-alignments and then adds the third 1-alignment to this to create a 3-alignment. The parameter 'min_peaks=2' specifies that any peak column of the data matrix which has less than two peaks in the final alignment will be dropped. This is useful to clean up the data matrix of accidental peaks that are not truly observed over the set of replicates.
- Line 27: the resulting 3-alignment is stored on disk, converted into the alignment tables containing peak retention times ('rt1.csv') and the corresponding peak areas ('area1.csv'). These two files are plain ASCII files is CSV format, and are saved in the directory 05/output/.

In general one is interested in the file 'area1.csv' which contains the data matrix where the corresponding peaks are aligned in the columns and each row corresponds to an experiment. The file 'rt1.csv' is useful for manually inspecting the alignment in some GUI driven program.

Running the above script with `$ python proc.py` produces the following output:

```
01  Aligning input 1
02   -> Loading experiment from the binary file '../04/output/a0806_140.expr'
03   -> Loading experiment from the binary file '../04/output/a0806_141.expr'
04   -> Loading experiment from the binary file '../04/output/a0806_142.expr'
05   Calculating pairwise alignments for 3 alignments (D=2.50, gap=0.30)
06   -> 2 pairs remaining
07   -> 1 pairs remaining
08   -> 0 pairs remaining
09   -> Clustering 6 pairwise alignments. Done
10   Aligning 3 items with guide tree (D=2.50, gap=0.30)
11   -> 1 item(s) remaining
12   -> 0 item(s) remaining
```

6.3 Between-state alignment of peak lists from multiple experiments

[*This example is in pyms-test/63 and pyms-test/61a*]

The Example 5 demonstrates how the peaks lists are aligned within a single experiment with multiple replicates (called "within-state alignment"). For example, if there are 8 experimental replicates performed on wild-type cells, Example 05 gives a recipe how to align such a set of experiments. In practice one is often interested in comparing two experimental states, ie. wild-type and mutant cells. In a typical experimental setup one would collect multiple replicate experiments on each state (for example, 8 experimental replicates on wild-type cells and 8 on the mutant cells). To analyze the results of such an experiment statistically one needs to align the peak lists within each experimental state (wild-type and

mutant) and also between the two states. The result of such an alignment would be the data matrix of integrated peak areas. In the example above the data matrix would contain 16 rows (corresponding to 8 wild type and 8 mutant experiments), while the number of columns would be determined by the number of unique peaks (metabolites) detected in the two experiments.

In principle, the method shown in the Example 5 could be used to align experiments from the two or more experimental states each containing multiple replicate experiments. However, a more careful analysis of the problem shows that the optimal approach to alignment is first to align experiments within each set of replicates (within-state alignment), and then to align the resulting alignments (between-state alignment) [7]. Within each state the experiments are true replicates, and we expect, at least in theory, that all compounds are observed in all experiments. This is however not true between the states, for example in metabolites observed in wild-type versus mutant cells, and this makes the alignment problem harder.

This example demonstrates how the peak lists from two cell states are aligned, the cell state A consisting of three experiments aligned in the Example 04 ('a0806_140', 'a0806_141', and 'a0806_142'), and the cell state B consisting of three experiments aligned in the Example 04a ('a0806_077', 'a0806_078', 'a0806_079'). The example in pyms-text/04a/ is a simple repetition of the example in pyms-text/04/ as explained in the Example 04 above only with different experiments.

The alignment script used to align the two states A and B is given below:

```
01  """proc.py
02  """
03
04  import sys
05  sys.path.append("/home/current/proj/PyMS/")
06
07  from pyms.Experiment.IO import read_expr_list
08  from pyms.Peak.List.DPA.Function import exprl2alignment
09  from pyms.Peak.List.DPA.Class import PairwiseAlignment
10  from pyms.Peak.List.DPA.Function import align_with_tree
11
13  input1 = "input1"
14  input2 = "input2"
15
17  Dw = 2.5  # rt modulation [s]
18  Gw = 0.30 # gap penalty
19
21  print 'Aligning input 1'
22  E1 = read_expr_list(input1)
23  F1 = exprl2alignment(E1)
24  T1 = PairwiseAlignment(F1, Dw, Gw)
25  A1 = align_with_tree(T1, min_peaks=2)
26
27  print 'Aligning input 1'
```

```
28  E2 = read_expr_list(input2)
29  F2 = exprl2alignment(E2)
30  T2 = PairwiseAlignment(F2, Dw, Gw)
31  A2 = align_with_tree(T2, min_peaks=2)
32
34  Db = 10.0 # rt modulation
35  Gb = 0.30 # gap penalty
36
37  print 'Aligning input {1,2}'
38  T9 = PairwiseAlignment([A1,A2], Db, Gb)
39  A9 = align_with_tree(T9)
40
41  A9.write_csv('output/rt.csv', 'output/area.csv')
```

There are two external input files used in this script ('input1' and 'input2'), listing the experiments from the state A and state B. The ifile 'input1' is identical as given in Example 5, while the listing of input file 'input2' defines where are the experiment dumps for the state B:

```
../04a/output/a0806_077.expr
../04a/output/a0806_078.expr
../04a/output/a0806_079.expr
```

The explanations of the alignment script are given below:

- Lines 21-25 run the within-state experiment of the state A, and are explained in the Example 5. Lines 27-31 are identical, and run the within-state alignment of the state B. The within-state alignment of experiments A is encapsulated in the variable A1, while the within-state alignment of the experiments B is encapsulated in the variable A2.
- Lines 34 and 34 specify the alignment parameters for between-state alignment of A and B. In the example the retention time tolerance for between-state alignment is greater compared to the retention time tolerance for the within-state alignment as the two sets of replicates were recorded on different days and we expect less fidelity in retention times between them.
- Lines 37-39 execute the alignment of two alignments. Exactly the same functions are used as in the within-state alignment (at this point the purpose of converting experiments to 1-alignments becomes apparent: this allows a generalization of functions PairwiseAlignment() and align_with_tree(), which always operate on the alignment objects.
- Line 41: the resulting alignment is saved to a file.

Running the above script with the command `$ python proc.py` produces the following output. Both pyms-text/04/proc.py and pyms-text/04a/proc.py need to be run to create the experiment dumps that are input for the alignment demonstrated here.

```
01  Aligning input 1
02   -> Loading experiment from the binary file '../04/output/a0806_140.expr'
03   -> Loading experiment from the binary file '../04/output/a0806_141.expr'
04   -> Loading experiment from the binary file '../04/output/a0806_142.expr'
05   Calculating pairwise alignments for 3 alignments (D=2.50, gap=0.30)
06   -> 2 pairs remaining
07   -> 1 pairs remaining
08   -> 0 pairs remaining
09   -> Clustering 6 pairwise alignments. Done
10   Aligning 3 items with guide tree (D=2.50, gap=0.30)
11   -> 1 item(s) remaining
12   -> 0 item(s) remaining
13  Aligning input 1
14   -> Loading experiment from the binary file '../04a/output/a0806_077.expr'
15   -> Loading experiment from the binary file '../04a/output/a0806_078.expr'
16   -> Loading experiment from the binary file '../04a/output/a0806_079.expr'
17   Calculating pairwise alignments for 3 alignments (D=2.50, gap=0.30)
18   -> 2 pairs remaining
19   -> 1 pairs remaining
20   -> 0 pairs remaining
21   -> Clustering 6 pairwise alignments. Done
22   Aligning 3 items with guide tree (D=2.50, gap=0.30)
23   -> 1 item(s) remaining
24   -> 0 item(s) remaining
25  Aligning input {1,2}
26   Calculating pairwise alignments for 2 alignments (D=10.00, gap=0.30)
27   -> 0 pairs remaining
28   -> Clustering 2 pairwise alignments. Done
29   Aligning 2 items with guide tree (D=10.00, gap=0.30)
30   -> 0 item(s) remaining
```

6.4 Comparing two peak lists by using dynamic programming alignment

[*This example is in pyms-test/68*]

The PyMS package pyms.Peak.List.Metric provides a function to compare two peak lists. This allows peak detection methods from different programs to be evaulated or a peak detection method to be compared to a 'known' or expert evaluated result. The following example compares Xcalibur peak detection and AMDIS peak detection.

```
>>> from from pyms.Experiment.IO import load_amdis_expr,load_xcalibur_expr
>>> from pyms.Peak.List.Metric import metric
```

```
>>> andi_file = "pyms-data/121107B_10.CDF"
>>> xcalibur_peaks_file = "pyms-data/121107B_10_xcalibur_peaks.txt"
>>> amdis_peaks_file = "pyms-data/121107B_10.ELU"
>>> amdis_expr = load_amdis_expr(amdis_peaks_file)
 -> Processing AMDIS experiment
 -> Reading AMDIS ELU file 'pyms-data/121107B_10.ELU'
>>> xcalibur_expr = load_xcalibur_expr(xcalibur_peaks_file,andi_file)
 -> Processing Xcalibur experiment
 -> Reading Xcalibur peak file 'pyms-data/121107B_10_xcalibur_peaks.txt'
 -> Processing netCDF file 'pyms-data/121107B_10.CDF'
    [ 7038 scans, masses from 70 to 600 ]
>>> metric_result = metric(amdis_expr.peaks, xcalibur_expr.peaks)
>>> print "Metric distance is ",metric_result
Metric distance is  0.89724073048
```

The full list of matching peaks and distances between individual peaks can be displayed by setting the verbose flag:

```
>>> metric_result = metric(amdis_expr.peaks, xcalibur_expr.peaks, verbose=True)
[-- output deleted --]
33.71 -
33.71 -
33.98 33.98 0.43
34.50 34.50 0.03
-35.01
35.08 -
-35.77
35.92 -
36.32 -
36.60 -
36.65 -
37.14 -
37.44 37.44 0.31
-39.60
40.99 40.99 0.43
41.59 -
-42.93
Metric distance is  0.89724073048
```

Bibliography

[1] Python. http://www.python.org.

[2] UrbanSim. http://www.urbansim.org/.

[3] Serra J. *Image Analysis and Mathematical Morphology.* Academic Press, Inc, Orlando, 1983. ISBN 0126372403.

[4] Sauve AC and Speed TP. Normalization, baseline correction and alignment of high-throughput mass spectrometry data. *Procedings Gensips*, 2004.

[5] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. http://www.scipy.org/.

[6] Biller JE and Biemann K. Reconstructed mass spectra, a novel approach for the utilization of gas chromatograph–mass spectrometer data. *Anal. Lett.*, 7:515–528, 1974.

[7] Robinson MD, De Souza DP, Keen WW, Saunders EC, McConville MJ, Speed TP, and Likic VA. A dynamic programming approach for the alignment of signal peaks in multiple gas chromatography-mass spectrometry experiments. *BMC Bioinformatics*, 8:419, 2007.