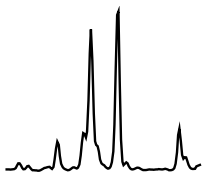

PyMS User Guide

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 provde 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
- pyms-data – Data used in pyms-test

Each part is a separate project on Google Code that can be downloaded separately.

1.2 PyMS installation

There are many 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 in detail 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 by the program called '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.
$ ls -CF
pyms/
$
```

1.2.2 Minimal PyMS installation

The minimal PyMS installation involves dowloading the PyMS code and installing the pycdf package. Download the pycdf package from `http://pysclint.sourceforge.net/pycdf/`, and follow the installation instructions that come with it. To test your pycdf installation:

```
$ python
Python 2.5.1 (r251:54863, Jul 10 2007, 08:23:47)
[GCC 4.1.1 20070105 (Red Hat 4.1.1-52)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pycdf
>>>
```

[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'.

If the pycdf is installed properly the above import statement will produce no error messages.

The PyMS code (ie. directory pyms/) can be placed anywhere as long as the Python interpreter is made aware where it is (see Python documentation for details on how to make packages visible to Python interpreter). In example, we keep the pyms code in `/home/current/proj/PyMS/` (ie. the PyMS code is `/home/current/proj/PyMS/pyms/`). Since this is a non-standard location for Python packages, before running PyMS we set the following in Python:

```
import sys
sys.path.append("/home/current/proj/PyMS/")
```

This makes Python aware that some packages may be in `/home/current/proj/PyMS/`. Therefore to check the PyMS installation one might try the following:

```
$ python
Python 2.5.1 (r251:54863, Jul 10 2007, 08:23:47)
[GCC 4.1.1 20070105 (Red Hat 4.1.1-52)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.path.append("/home/current/proj/PyMS/")
>>> from pyms.IO.ANDI.Class import ChemStation
>>>
```

The last command has loaded the class 'ChemStation' from PyMS. Since this did not produce any error messages the PyMS was accessible (and moreover, that pycdf is properly installed, since the class 'ChemStation' uses pycdf).

For examples on PyMS use see Chapter 2.

1.3 Minmax peak detector

1.3.1 Introduction

Minmax peak detector is the simplest kind of a peak finding algoritam for TIC. It operates by finding peak maxima, and then attempting to determine peak boundaries. Cursory evidence suggests that gives results similar to the ChemStation peak detection, but this was not examined rigorously. The purpose of this algorithm is to provide an example of how 1D peak pickin can be implemented in PyMS. *At present the Minmax algorithm was not properly tested, do not use it for critical publication quality results.*

1.3.2 A brief description of the algorithm

Many peak detection algorithms are used in practice to process GC/LC-MS data, but only a few are fully documented, most notable those of open source projects MZmine [2] and XCMS [3]. MZmine detects peaks by finding local maxima of a certain width [2]. In XCMS peaks are detected by using an empirical signal-to-noise cutoff after matched filtration with a second-derivative Gaussian [3]. PyMS peak detection procedure was developed in-house, and relies on finding local maxima and local minima in the signal, followed by a subsequent refinement of peak left and right boundaries. Peak detection depends on two input parameters: window width over which a peak is expected to be a global maximum, and the scaling factor S used to calculate the intensity threshold $S\sigma$ which must be exceeded at the peak apex. The noise level σ is estimated prior to peak detection by repeatedly calculating median absolute deviation (MAD – a robust estimate of the average deviation) over randomly placed windows and taking the minimum. A detailed description of procedures for peak detection follows.

1. **Extracting local maxima**. Initially, an ordered list of local maxima in the signal with an intensity larger than a threshold is compiled. Two input parameters are specified by the user: the width of a window over which the peak is required to be a global maximum (W); and (2) the scaling factor S used to calculate intensity threshold $S\sigma$, where S is the noise level estimated previously (defaults: $W = 2$ data points, $S = 10$). User specified window is centered on each point of the signal, and the point is deemed to be a local maximum if the following is satisfied:

   (a) It is equal or greater than all of the points within the window W.

   (b) It is greater than at least one point in the half-window interval to the left, and at least one point in the half-window interval to the right.[2]

   (c) Any point closer to the edge of the signal than half-window is rejected.

   Intensity at each local maxima is tested, and those that have the intensity below the threshold N*S are rejected. Accepted local maxima are compiled into a list.

2. **Determination of peak left/right boundaries**. For each local maxima (base maximum) the stretch of the signal between itself and the next local maximum on either side is extracted. These two signal slices are searched for the first local minimum in the direction away from the base maximum point itself. The local maxima are defined in a very similar manner as the local maxima in the previous step. A point is deemed to be a local maximum if:

   (a) It is equal or smaller than all the points within the window W.

   (b) It is smaller than at least one point in the half-window interval to the left, and at least one point in the half-window interval on the right.

   (c) Any point closer to the edge of the slice than half-window is rejected. This has the effect that the boundary point cannot approach next peak's apex closer than half-window.

   (d) If no minimum point is found, set the boundary point to the point furtherest away from the base maximum, but outside to the half-window range of the adjacent peak.

[2] This is to reject points within intervals of uniform intensity

3. **Elimination of peak overlaps**. In spectra dense with peaks peak boundaries as found in the step (2) may overlap due to the effect of user supplied window. The list of pre-peaks is searched for overlapping peaks. In overlapping peaks the right boundary of the lower retention time peak overlaps with the left boundary of the higher retention time peak. The overlapping boundaries are resolved by finding the point of minimum intensity between the two peaks (the split point). The peak boundaries are set to one point to the left from the split point for the right boundary of the lower retention time peak, and to one point to the right from the split point for the left boundary of the higher retention time peak.

4. **Correction for long tails**. In this step peak boundaries are adjusted to remove stretches of near-uniform intensities (i.e. long tails). Each peak is divided at the apex into two halves, and each half is processed individually in the boundary-to-apex direction. A line is fitted through M points from the boundary in the least-squares sense. Prior to calculating the angle between the line and the retention time axis, the rise in intensity is normalized with the intensity at the peak apex. If this angle is below the user specified cutoff (Q) the boundary point is dropped, and the process is repeated. This adjustment is repeated until the best fit through M points from the boundary gives an angle greater than the cutoff. The parameters M and Q are user specified (defaults: $M = 3$, $Q = 1.0°$).

Using PyMS

2.1 Introduction

This chapter demonstrates main functions of PyMS in a tutorial like manner. The data files used in the examples are provided in the project 'pyms-data'. The commands executed interactively are grouped together by example, and provided as Python scripts in the project 'pyms-test'.

The setup used in the examples below is as follows. The projects 'pyms', 'pyms-test', 'pyms-docs', and 'pyms-data' were downloaded in the directory `/home/current/proj/PyMS`. In the project 'pyms-test' there is a directory corresponding to each example coded with the example number (ie. `pyms-test/01/` corresponds to Example 1). 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 by simply:

$ 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("/home/current/proj/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 need to be adjusted to match your own location of pyms.

All data files (raw data files, peak lists etc) used in the example below can be found in 'pyms-data'.

2.2 Example 1: Reading of GC-MS data and basic manipulation of data

2.2.1 Reading ChemStation GC-MS data into PyMS

This example is in pyms-test/01

The PyMS package pyms.IO provides capabilities to read the raw GC-MS data stored in the ANDI-MS format. The function IO.ANDI.ChemStation() provides the interface to ANDI-MS data files saved from Agilent ChemStation software.[3]

The file 'a0806_140.CDF' is a GC-MS experiment exported from Agilent ChemStation (located in 'pyms-data'). This file can be loaded in the memory as follows:

```
>>> from pyms.IO.ANDI.Class import ChemStation
>>> andi_file = "/home/current/proj/PyMS/pyms-data/a0806_140.CDF"
>>> andi_data = ChemStation(andi_file)
 -> Processing netCDF file '/home/current/proj/PyMS/pyms-data/a0806_140.CDF'
    [ 3236 scans, masses from 50 to 550 ]
>>>
```

The above command creates the object 'andi_data' which is an *instance* of the class IO.ANDI.ChemStation.

2.2.2 Exploring an ANDI-MS data object

The object 'andi_data' has several attributes and methods associated with it.

```
>>> print "ANDI-MS data filename:", andi_data.get_filename()
ANDI-MS data filename: /home/current/proj/PyMS/pyms-data/a0806_140.CDF
```

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

```
tic = andi_data.get_tic()
```

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, ion chromatograms at m/z = 65), or cumulative intensities over all measured m/z (TIC).

The method `get_ic_at_index(i)` returns i-th ion chromatogram, as an IonChromatogram object. For example, to get the first ion chromatogram from the data:

[3] ANDI-MS data format stands for Analytical Data Interchange for Mass Spectrometry, and was developed for the description of mass spectrometric data developed in 1994 by Analytical Instrument Association. ANDI-MS is essentially a recommendation, and it is up to individual vendors of mass spectrometry processing software to implement "export to ANDI-MS" feature in their software.

```
ic = andi_data.get_ic_at_index(1)
```

The method `get_ic_at_mass(MZ)` returns the ion chromatogram for m/z = MZ. For example, to get the ion chromatogram that corresponds to m/z = 73:

An ion chromatogram object has a method `is_tic()` which returns True is the ion chromatogram is 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
```

2.2.3 Writing data to a file

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
  5.0944      745997.0000
  5.1002      726566.0000
  5.1059      717704.0000
  5.1116      684214.0000
  5.1173      701866.0000
  5.1230      893306.0000
  5.1287     1278099.0000
  5.1345     1290984.0000
  5.1402      925558.0000
  5.1459      644122.0000
```

Figure 2.1 shows the plot of the file 'tic.dat' produced with the program Gnuplot. The Gnuplot script used to produce this plot is provided as pyms-test/01/output/plot.gnu.

The method `get_intensity_matrix()` of ChemStation object returns the entire matrix of intensities:

```
>>> im = andi_data.get_intensity_matrix()
```

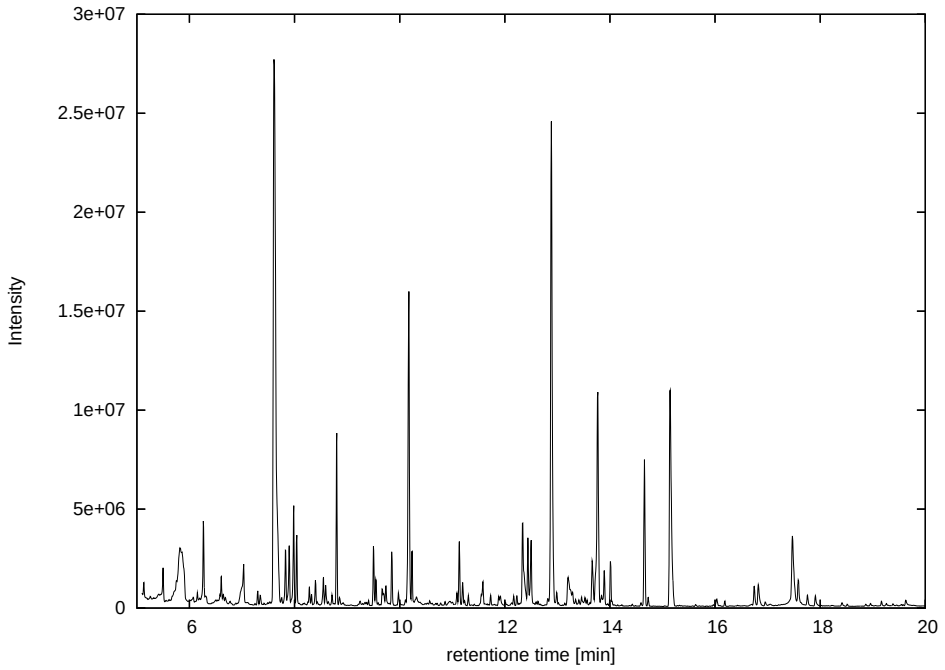

Figure 2.1: The Gnuplot plot of the file 'tic.dat'

```
>>> print "Dimensions of the intensity matrix are:",len(im),"x",len(im[0])
Dimensions of the intensity matrix are: 3236 x 501
```

This data matrix contains 3236 time points (MS scans), and each time point corresponds to a mass spectrum of 501 m/z points.

The intensity matrix can be saved to a file with the function 'save_data()':

```
save_data("output/im.dat", im)
```

The entire data (ie. ChemStation object) can be saved as CSV with the method `export_csv()`. For example,

```
>>> andi_data.export_csv("output/data")
```

will create 'data.im.csv, data.mz.csv, and data.rt.csv where these are the intensity matrix, retention time vector, and m/z vector in the CSV format.

2.3 Example 2: Creating signal peaks

This example is in pyms-test/02 G In PyMS a signal peak is represented as 'Peak' object defined in pyms.Peak.Class.py. A peak object is initialized with two arguments: peak retention time and peak raw area. The following commands create a peak named 'p' with the retetion time of 5.553 min and a peak area of 2759280 (this is the peak no. 3 in the ChemStation peak area report file 'a0806_140.txt'):

```
>>> from pyms.Peak.Class import Peak
>>> p = Peak(5.553*60.0,2759280)
```

As a matter of convention PyMS internally stores retention times in seconds, hence above the retention time is multiplied by 60. Peak raw area is in arbitrary units.

Peak properties can be accessed through its attributes:

```
>>> print "Peak retention time is", p.rt
Peak retention time is 333.18
>>> print "Peak raw area is", p.raw_area
Peak raw area is 2759280.0
```

Other important properties of a peak object are peak normalized area and peak mass spectrum. The peak created in the above example does not have values associated with these two attributes, and they are merely initialized to 'None':

```
>>> print "Peak normalized area is", p.norm_area
Peak normalized area is None
>>> print "Peak mass spectrum is", p.mass_spectrum
Peak mass spectrum is None
```

The peak mass spectrum can be set by calling the method `set_mass_spectrum()`. This method requires the raw data, and fetches mass spectrum at peak retention time:

```
>>> p.set_mass_spectrum(andi_data)
```

This will set the mass spectrum attribute:

```
>>> print p.mass_spectrum
 49976  54520 102752  15570   1872  18392   8765  14966  46136  16141
  1635   1743    686   1019    712   1199   1641   3182   1234  30400
  4261   3746   3348  82392   8354  24824   3797   6086  23312 140480
[--output deleted--]
```

These are m/z channel intensities in arbitrary units. The m/z values themselves are in the mass list attribute:

```
>>> print p.mass_list
[50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81,
[--outout deleted--]
```

The length of the two arrays must match:

```
>>> print len(p.mass_spectrum)
501
>>> print len(p.mass_list)
501
```

The mass spectrum can be written to a file by calling the peak `write_mass_spectum()` method:

```
>>> p.write_mass_spectrum("output/ms.dat")
```

The file 'output/ms.dat' contains the pairs (mz, intensity), one pair per line:

```
$ head output/ms.dat
  50.000        49976.000
  51.000        54520.000
  52.000       102752.000
  53.000        15570.000
  54.000         1872.000
  55.000        18392.000
  56.000         8765.000
  57.000        14966.000
  58.000        46136.000
  59.000        16141.000
```

Figure 2.2 shows the plot of ms.dat created with the program Gnuplot. The gnuplot script used to create this plot is provided as pyms-test/02/output/plot.gnu.

2.4 Example 3: Creating an experiment from ChemStation data

This example is in pyms-test/03

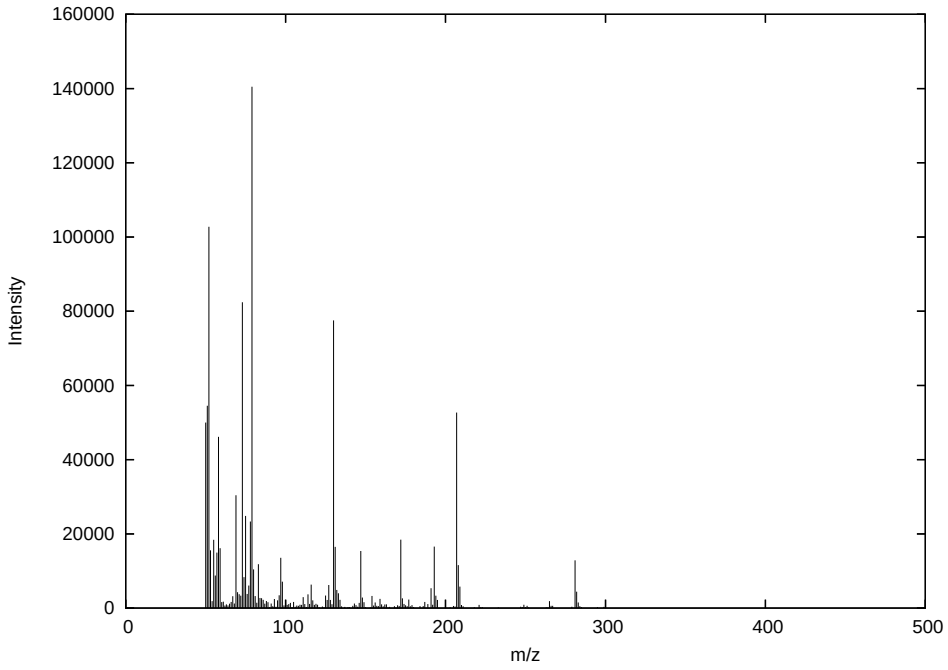

Figure 2.2: The plot of the file 'ms.dat' produced with Gnuplot

The input files used in this example are 'a0806_140.CDF' (ANDI-MS data file exported from Agilent ChemStation) and 'a0806_140.txt.anno' (the corresponding peak area report file generated by ChemStation).

The original peak area report file exported from ChemStation is 'a0806_140.txt'. This file was manually edited to flag non-informative peaks, and also peaks which originate from internal reference compounds added during the sample preparation. For example, below is the snippet of the original file 'a0806_140.txt':

```
52   8.215   470  478  480 PV 3   91906   1414873   0.14%   0.056%
53   8.242   480  483  486 VV 4   99010   1301807   0.13%   0.051%
54   8.285   486  490  495 VV 2  124626   2241940   0.22%   0.088%
55   8.337   495  499  501 VV 2   44820    662145   0.06%   0.026%

56   8.365   501  504  506 VV 2   58181    789953   0.08%   0.031%
57   8.399   506  510  515 VV    713172   9893695   0.97%   0.389%
58   8.443   515  518  521 VV 5  143812   2590885   0.25%   0.102%
59   8.477   521  524  526 VV 4  149127   2438042   0.24%   0.096%
60   8.548   526  536  558 VV 2 55398426 1025005618 100.00%  40.335%
```

and the same file in the file 'a0806_140.txt.anno'

```
52   8.215   470  478  480 PV 3   91906   1414873   0.14%   0.056%
```

```
 53   8.242   480  483  486 VV 4   99010   1301807   0.13%   0.051% BLANK
 54   8.285   486  490  495 VV 2  124626   2241940   0.22%   0.088%
 55   8.337   495  499  501 VV 2   44820    662145   0.06%   0.026%

 56   8.365   501  504  506 VV 2   58181    789953   0.08%   0.031%
 57   8.399   506  510  515 VV    713172   9893695   0.97%   0.389%
 58   8.443   515  518  521 VV 5  143812   2590885   0.25%   0.102%
 59   8.477   521  524  526 VV 4  149127   2438042   0.24%   0.096%
 60   8.548   526  536  558 VV 2 55398426 1025005618 100.00%  40.335% BLANK
```

In the file 'a0806_140.txt.anno' the keywork 'BLANK' was manually added to peaks 53 and 60, which are known to originate from the derivatizing agent use in GC-MS data preparation. These peaks can be excluded from the analysis later.

The peak eluting at 15.590 min originated from scyllo-inositol reference compound added during sample preparation. In the file 'a0806_140.txt.anno' this peak was labelled as follows:

```
178  15.590  1758 1767 1773 VV   5307268  80504143   7.85%   3.168% RF-SI
```

There could be an arbitrary number of reference peaks in the peak list, and each must have a unique reference 'tag' starting with 'RF-' and following with a two letter code denoting a particular reference sompound (in this case SI for scyllo-inositol).

The ChemStation peak list is loaded in PyMS with the function 'read_chem_station_peaks()':

```
>>> from pyms.Peak.List.IO import read_chem_station_peaks
>>> peak_file = "/home/current/proj/PyMS/pyms-data/a0806_140.txt.anno"
>>> peaks = read_chem_station_peaks(peak_file)
 -> Reading ChemStation peak integration report
'/home/current/proj/PyMS/pyms-data/a0806_140.txt.anno'
```

The variable 'peaks' now contains the peaks from the file 'a0806_140.txt.anno' This is merely a Python list:

```
>>> type(peaks)
<type 'list'>
>>> print "The number of peaks is:", len(peaks)
The number of peaks is: 347
```

The next step is to set the mass spectrum is set for each peak. For this we need first to load the raw data:

```
>>> import sys
```

```
>>> sys.path.append("/home/current/proj/PyMS/")
>>> from pyms.IO.ANDI.Class import ChemStation
>>> andi_file = "/home/current/proj/PyMS/pyms-data/a0806_140.CDF"
>>> andi_data = ChemStation(andi_file)
 -> Processing netCDF file '/home/current/proj/PyMS/pyms-data/a0806_140.CDF'
    [ 3236 scans, masses from 50 to 550 ]
```

The following command sets the mass spectrum for each peak,

```
>>> for peak in peaks:
...     peak.set_mass_spectrum(andi_data)
```

The experiment object is initiated with the list of peaks and the experiment label, in this case "a0806_140":

```
>>> from pyms.Experiment.Class import Experiment
>>> expr = Experiment("a0806_140", peaks)
```

In the next steps we call a series of methods associated with the experiment object to set the reference peak, remove blank peaks, create peak normalized area (in this case the same as peak raw area), purge negative peaks (if any), and finally select the retention time range for the experiment to between 6.5 and 21 minutes, discarding all peaks outside this range:

```
>>> expr.set_ref_peak("si")
 [ Reference peak found: 'rf-si' @ 935.400 s ]
  [ Removing reference peak 'rf-si' @ 935.400 s ]
>>> expr.remove_blank_peaks()
        [ Designated blank peak at 438.660 s removed ]
        [ Designated blank peak at 494.520 s removed ]
        [ Designated blank peak at 512.880 s removed ]
        [ Designated blank peak at 751.980 s removed ]
>>> expr.raw2norm_area()
>>> expr.purge_peaks()
 Experiment a0806_140: 0 peaks purged (below threshold=0.00)
>>> expr.sele_rt_range(["6.5m", "21m"])
 -> Selecting peaks by retention time (from 6.5m to 21m): 247 peaks selected
```

Finally, we dump the experiment object to a file allowing it to be used later, for example in the process of peak alignment (see the example pyms-test/04):

```
>>> from pyms.Experiment.IO import dump_expr
>>> dump_expr(expr, "output/a0806_140.pickle")
 -> Experiment 'a0806_140' saved as 'output/a0806_140.pickle'
```

2.5 Example 4: Preparation of multiple experiments for peak alignment by dynamic programming

This example is in pyms-test/04

Before aligning peak 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 retentiontime range.

This example considers the preparation of three GC-MS experiments for the peak alignment, 'a0806_140', 'a0806_141', 'a0806_142'. The input for each experiment consists of two files: the peak list file exported from Agilent ChemStation (peak area report file), and the corresponding ANDI-MS data file. For example, the input files for the experiment 'a0806_140' are:

- a0806_140.txt.a – ChemStation peak area report, manually edited to denote non-informative peaks and the reference peak
- a0806_140.CDF – ANDI-MS file corresponding to 'a0806_140.txt.a'

The ANDI-MS data files are required for the assignment of peak mass spectra, since peak alignment by dynamic programming uses both peak retention times and peak mass spectra [4].

The listing below shows the Python code for the script pyms-test/04/proc.py:

```
01  """proc.py
02  """
03
04  import sys, os
05  sys.path.append("/home/current/proj/PyMS/")
06
07  from pyms.IO.ANDI.Class import ChemStation
08  from pyms.Experiment.Class import Experiment
09  from pyms.Peak.List.IO import read_chem_station_peaks
10  from pyms.Experiment.IO import dump_expr
11
12  base_path = "/home/current/proj/PyMS/pyms-data/"
13
14  expr_codes = [ "a0806_140", "a0806_141", "a0806_142" ]
15
16  for expr_code in expr_codes:
17
18      peak_file = os.path.join(base_path, expr_code + ".txt.a")
19      andi_file = os.path.join(base_path, expr_code + ".CDF")
20
```

```
21      andi_data = ChemStation(andi_file)
22      peaks = read_chem_station_peaks(peak_file)
23
24      andi_data.null_mass(73)
25      andi_data.null_mass(147)
26
27      for peak in peaks:
28          peak.set_mass_spectrum(andi_data)
29          peak.crop_mass_spectrum(50,540)
30
31      expr = Experiment(expr_code, peaks)
32
33      expr.set_ref_peak("si")
34      expr.remove_blank_peaks("blank")
35      expr.raw2norm_area()
36      expr.purge_peaks()
37      expr.sele_rt_range(["6.5m", "21m"])
38
39      dump_expr(expr, "output/" + expr_code + ".expr")
```

The line 14 defines three experiments, by defining only the root name for each experiment. In the line 16 a loop is initiated over all experiments defined in the list 'expr_codes'. The actions in the body of the loop are applied to each experiment in turn:

- Full path names of the peak file and ANDI-MS file are created (lines 18-19)
- ANDI-MS and peak report files are loaded (lines 21-22)
- The m/z channels 73 and 147 are nulled in the raw data files (lines 24-25). These two m/z channels contain strong trailing signals from the derivatizing agent across all retention times, and therefore can potentially lower the sensitivity in mass spectra comparison.
- For each peak in the experiment the mass spectrum is set, and the m/z range is restricted to 50-540 (lines 27-29)
- An experiment object is created from the input data (line 31)
- The reference peak is removed (line 33), non-informative peaks are removed (line 34), peak operation area is created from the raw peak area (line 35), peaks below the threshold are remove (here negative peaks, if any), and the retention time of between 6.5 minutes and 21 minutes is selected.
- The experiment will be dumped onto a file in the sub-directory 'output', under the names 'a0806_140.expr', 'a0806_141.expr', and 'a0806_142.expr'.

The script 04/proc.py can be run in the batch mode from the unix shell propmt:

```
$ python proc.py
```

The output of this command printed on the screen terminal is shown below:

```
01  -> Processing netCDF file '/home/current/proj/PyMS/pyms-data/a0806_140.CDF'
02     [ 3236 scans, masses from 50 to 550 ]
03  -> Reading ChemStation peak integration report
    '/home/current/proj/PyMS/pyms-data/a0806_140.txt.a'
04  -> nulled mass 73
05  -> nulled mass 147
06  [ Reference peak found: 'rf-si' @ 935.400 s ]
07   [ Removing reference peak 'rf-si' @ 935.400 s ]
08  [ Designated blank peak at 438.660 s removed ]
09  [ Designated blank peak at 494.520 s removed ]
10  [ Designated blank peak at 512.880 s removed ]
11  [ Designated blank peak at 751.980 s removed ]
12  Experiment a0806_140: 0 peaks purged (below threshold=0.00)
13  -> Selecting peaks by retention time (from 6.5m to 21m): 247 peaks selected
14  -> Experiment 'a0806_140' saved as 'output/a0806_140.expr'
15  -> Processing netCDF file '/home/current/proj/PyMS/pyms-data/a0806_141.CDF'
16     [ 3236 scans, masses from 50 to 550 ]
17  -> Reading ChemStation peak integration report
    '/home/current/proj/PyMS/pyms-data/a0806_141.txt.a'
18  -> nulled mass 73
19  -> nulled mass 147
20  [ Reference peak found: 'rf-si' @ 935.280 s ]
21   [ Removing reference peak 'rf-si' @ 935.280 s ]
22  [ Designated blank peak at 438.960 s removed ]
23  [ Designated blank peak at 514.380 s removed ]
24  [ Designated blank peak at 751.980 s removed ]
25  Experiment a0806_141: 1 peaks purged (below threshold=0.00)
26  -> Selecting peaks by retention time (from 6.5m to 21m): 245 peaks selected
27  -> Experiment 'a0806_141' saved as 'output/a0806_141.expr'
28  -> Processing netCDF file '/home/current/proj/PyMS/pyms-data/a0806_142.CDF'
29     [ 3236 scans, masses from 50 to 550 ]
30  -> Reading ChemStation peak integration report
    '/home/current/proj/PyMS/pyms-data/a0806_142.txt.a'
31  -> nulled mass 73
32  -> nulled mass 147
33  [ Reference peak found: 'rf-si' @ 935.280 s ]
34   [ Removing reference peak 'rf-si' @ 935.280 s ]
35  [ Designated blank peak at 438.780 s removed ]
36  [ Designated blank peak at 513.000 s removed ]
37  [ Designated blank peak at 752.040 s removed ]
```

```
38  Experiment a0806_142: 0 peaks purged (below threshold=0.00)
39  -> Selecting peaks by retention time (from 6.5m to 21m): 259 peaks selected
40  -> Experiment 'a0806_142' saved as 'output/a0806_142.expr'
```

2.6 Example 5: Dynamic programming alignment of peak lists from multiple experiments

This example is in pyms-test/05

In this example the experiments 'a0806_140', 'a0806_141', and 'a0806_142' prepared in pyms-test/04 will be aligned, and therefore the script pyms-test/04/proc.py must be run first (see Example 4), to create the files 'a0806_140.expr', 'a0806_141.expr', 'a0806_142.expr' in the directory pyms-test/04/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.

```
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
16  Dw = 2.5  # rt modulation [s]
17  Gw = 0.30 # gap penalty
18
20  print 'Aligning input 1'
21  E1 = read_expr_list(input1)
22  F1 = exprl2alignment(E1)
23  T1 = PairwiseAlignment(F1, Dw, Gw)
24  A1 = align_with_tree(T1, min_peaks=2)
25
27  A1.write_csv('output/rt1.csv', 'output/area1.csv')
```

This script uses another file (file named "input1" on line 13) which lists the location of input experiments, one experiment per line:

```
../04/output/a0806_140.expr
../04/output/a0806_141.expr
../04/output/a0806_142.expr
```

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 [4]
- 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 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
```

2.7 Example 6: Between-state alignment of peak lists from multiple experiments

This example is in pyms-test/06 and pyms-test/04a

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) [4]. 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
```

Bibliography

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

[2] Katajamaa M, Miettinen J, and Oresic M. MZmine: toolbox for processing and visualization of mass spectrometry based molecular profile data. *Bioinformatics*, 22(5):634–636, 2006.

[3] Smith CA, Want EJ, O'Maille G, Abagyan R, and Siuzdak G. XCMS: processing mass spectrometry data for metabolite profiling using nonlinear peak alignment, matching, and identification. *Anal Chem*, 78(3):779–87, 2006.

[4] 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.