PyMS User Guide

Copyright © Andrew Isaac, Sean O'Callaghan and Vladimir Likić

PyMS version 1.0

A Python toolkit for processing of chromatography–mass spectrometry data

Contents

Chapter 1

Introduction

PyMS is software for processing of chromatography–mass spectrometry data. PyMS is written in Python programming language [1], and is released as open source, under the GNU Public License version 2. The driving 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. PyMS provides interactive processing capabilities through any of the various interactive Python front ends ("shells"). PyMS is essentially a Python library of chromatography–mass spectrometry processing functions, and individual commands can be collected into scripts which then can be run non-interactively when it is preferable to run data processing in the batch mode.

PyMS functionality consists of modules which are loaded when needed. For example, one such module provides display capabilities, and can be used to display time dependent data (e.g. total ion chromatogram), mass spectra, and signal peaks. This module is loaded only when visualisation is needed. If one is interested only in noise smoothing, only noise filtering functions are loaded into the Python environment and used to smooth the data, while the visualisation module (and other module) need not be loaded at all.

This modularity is supported on all levels of PyMS. For example, if one is interested in noise filtering with the Savitsky-Golay filter, only sub-module for Savitsky-Golay filter need to be loaded from the noise smoothing module, disregarding other modules, as well as other noise smoothing sub-modules. This organisation consisting of hierarchical modules ensures that the processing pipeline is put together from well defined modules each responsible for specific functions; and furthermore different functionalities are completely decoupled from one another, providing that implementing a new functionality (such as test or prototype of a new algorithm) can be implemented efficiently and ensuring that this will not break any existing functionality.

1.1 The PyMS project

The PyMS project consists of three parts, each of which exists as a project in the Google Code repository that can be downloaded separately. The three parts are:

- pyms – The PyMS code (http://code.google.com/p/pyms/)
- pyms-docs – The PyMS documentation (http://code.google.com/p/pyms-docs/)
- pyms-test – Examples of how to use PyMS (http://code.google.com/p/pyms-test/)

The project 'pyms' contains the source code of the Python package PyMS. The project 'pyms-docs' contains PyMS style guide (relevant for those who contribute to the PyMS code) and User Guide (this document). The project 'pyms-test' contains tests and examples showing how to use various PyMS features. These examples are explained in detail in subsequent chapters of this User Guide.

In addition, the current PyMS API documentation (relevant for those who are interested in PyMS internals) is available from here:
`http://bioinformatics.bio21.unimelb.edu.au/pyms.api/index.html`

1.2 PyMS installation

PyMS is written in Python, and extensible and modular object-oriented scripting language [1]. Python is highly portable, cross-platform programming language which works well on all major modern operating systems (Linux, MacOS X, Windows). PyMS is written in pure Python, and therefore works on all platforms on which Python has been ported.

The PyMS is essentially a python library (a 'package' in python parlance, which consists of several 'sub-packages'), and some of its functionality depends on several Python libraries, such as 'numpy' (the Python library for numerical computing), or 'matplotlib' (the Python library for plotting). These also need to work on the operating system of your choice for the functionality they provide to PyMS to be available. In general, the libraries that PyMS uses are available for all operating systems. The exception is 'pycdf' - a python interface to Unidata netCDF library written by Andre Gosselin of the Institut Maurice-Lamontagne, Canada (http://pysclint.sourceforge.net/pycdf/). *This library works only under Linux/Unix and therefore PyMS functionality which depends on it works only under the Unix operating system.*

There are several ways to install PyMS depending your computer configuration and personal preferences. PyMS has been developed on Linux, and a detailed installation instructions for Linux are given below. PyMS should work on all major Linux distributions, and has been tested extensively on Red Hat Linux.

1.2.1 Downloading PyMS source code

PyMS source code resides on publicly accessible 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. Regardless of the target operating system, the first step towards PyMS installation is to download the PyMS source code.

Google Code server maintains the source code with the program 'subversion' (an open-source version control system). To download the source code one needs to use the subversions client. Several subversion clients are available, some are open source, some freeware, and some are commercial (for more information see http://subversion.tigris.org/). The svn client programs are available for all operating systems. For example, on Linux we use the svn client program which ships with most Linux systems called simply 'svn'. The 'svn' exists for all mainstream operating systems[1]. A well known svn client for Windows is TortoiseSVN (http://tortoisesvn.tigris.org/). TortoiseSVN provides graphical user interface, and is tightly integrated with Windows. TortoiseSVN is open source and can be downloaded from the project web page (http://tortoisesvn.tigris.org/). There are also several commercial svn clients for Windows.

Subversion has extensive functionality, however only the very basic functionality is needed to download PyMS source code. For more information about subversion please consult the book freely available at http://svnbook.red-bean.com/.

If the computer is connected to the internet, and the subversion client 'svn' is installed. On Linux, 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
... [ further output deleted ] ...
Checked out revision 71.
```

[1]For example, on Linux CentOS 4 it ships as a part of the RPM package 'subversion-1.3.2-1.rhel4.i386.rpm'

1.3 External Libraries

In addition to the source code, for the full PyMS functionality several external libraries are required.

1.3.1 Package 'NumPy' (required for all aspects of PyMS)

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.

1.3.2 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.

1.3.3 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 uses 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`.

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

If the full SciPy package is installed the 'ndimage' will be available. However the SciPy contains extensive functionality, and its installation is somewhat involved. Sometimes it is preferable to install only the subpackage 'ndimage'. This subpackage is provided as the PyMS-dependencies gzipped file available for download from the PyMS webpage (see below).

1.3.5 Package 'matplotlib' (required for plotting)

The displaying of information such as Ion Chromatograms and detected peaks requires the package matplotlib. The matplotlib package can be downloaded from:
`http://matplotlib.sourceforge.net/`

1.3.6 Package 'mpi4py' (required for parallel processing)

This package is required for parallel processing with PyMS. It can be downloaded from:
`http://code.google.com/p/mpi4py/`

1.4 PyMS installation on Linux

We recommend compiling your own Python installation before installing PyMS. PyMS installation involves placing the PyMS code directory (pyms/) into a location visible to the Python interpreter. This can be in the standard place for 3rd party software (the directory site-packages/), or alternatively if PyMS code is placed in a non-standard location the Python interpreter needs to be made aware of it before before it is possible to import PyMS modules. For more on this please consult the Python command sys.path.append().

PyMS is currently being developed on Linux with the following packages:

```
Python-2.5.2
numpy-1.1.1
netcdf-4.0
pycdf-0.6-3b
ndimage.zip
Pycluster-1.41
matplotlib-0.99.1.2
mpi4py-1.2.1.tar.gz
mpich2-1.2.1p1.tar.gz
```

For easy installation, we provide these packages bundled together into the archive 'PyMS-Linux-deps-1.1.tar.gz' which can be downloaded from the Bio21 Institute web server at the University of Melbourne:
http://bioinformatics.bio21.unimelb.edu.au/pyms.html

In addition to the dependencies bundle, one also needs to dowload the PyMS source code as explained in the section 1.2.1). Below we give a quick installation guide of packages required by PyMS on Linux.

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. It is recommended to 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/netcdf/index.html`),
   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. Then 'pycdf' should 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 tweaking:

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

6. 'matplotlib' installation:

   ```
   $ tar xvfz matplotlib-0.99.1.2
   $ cd matplotlib-0.99.1.1
   $ python setup.py build
   $ python setup.py install
   ```

   The 'pyms.Display' module uses the TKAgg backend for matplotlib. If this is not your default backend, the matplotlibrc file may be edited. To locate the file 'matplotlibrc', in a python interactive session:

   ```
   >>> import matplotlib
   >>> matplotlib.matplotlib_fname()
   ```

   Open the matplotlibrc file in a text editor and adjust the 'backend' parameter to 'TKAgg'.

7. 'mpi4py' installation:

   This package is required for running PyMS processing on multiple processors (CPUs). Instructions how to install this package and run PyMS processing in parallel are given in Section 8.1.

1.5 Troubleshooting

The most likely problem with PyMS installation is a problem with installing one of the PyMS dependencies.

1.5.1 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.

1.6 PyMS tutorial and examples

This document provides extensive tutorial on the use of PyMS, and the corresponding examples can be downloaded from the publicly accessible Google code project ’pyms-test’ (http://code.google.com/p/pyms-test/). The data used in PyMS documentation and examples is available from the Bio21 Institute server at the University of Melbourne. Please follow the link from the PyMS web page:
`http://bioinformatics.bio21.unimelb.edu.au/pyms`
or go directly to
`http://bioinformatics.bio21.unimelb.edu.au/pyms-data/`

A tutorial illustrating various PyMS features in detail is provided in subsequent chapters 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 chapter number (ie. `pyms-test/21a/` corresponds to the Example 21a, from 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.

1.7 Using PyMS on Windows

Python is highly cross-platform compatible, and PyMS works seamlessly on Windows. The only exception is reading of data in ANDI-MS (NetCDF) format, widely used standard format for storing raw chromatography–mass spectrometry data (see 2.1) This capability in PyMS depends on the 'pycdf' library, which is not supported on Windows (see the Subsection 1.3.2). Therefore at present the ability to read ANDI-MS files is limited to Linux. All other PyMS functionality is available under Windows.

We use Linux for the development and deployment PyMS, and this User Guide largely assumes that PyMS is used under Linux. In this section we give some pointers on how to use PyMS under Windows.

1.7.1 PyMS installation on Windows

1. Install Python, NumPy, SciPy, and matplotlib for Windows. The bundle of these software packages tested under Windows XP and Windows 7 can be downloaded from the PyMS project page at the Bio21 Institute, University of Melbourne
   `http://bioinformatics.bio21.unimelb.edu.au/pyms.html`
2. Download the latest PyMS code from the PyMS Google Code project page
   `http://code.google.com/p/pyms/`

3. Unpack the PyMS code and place it in a standard place for Python libraries, or adjust the PYTHONPATH variable to include the path to PyMS. If Python is installed in C:
   Python25, the standard place for Python libraries is C:
   Python25
   Libs
   site-packages

4. Start IDLE, or other Python shell. If PyMS is installed properly, the following command will not return any warning or error messages:

   ```
   >>> import pyms
   ```

1.7.2 Example processing GC-MS data on Windows

The example data can be downloaded from `http://bioinformatics.bio21.unimelb.edu.au/pyms/data/`. We will use the data file in the JCAMP-DX format, named `gc01_0812_066.jdx`. Once downloaded this data file will be placed in the folder `C:Data` for the example below.

The Python environment can be accessed from several Python shells. The default shell that comes with the Python 2.5 installation is IDLE. In this example we first load the raw data,

```
>>> from pyms.GCMS.IO.JCAMP.Function import JCAMP_reader
>>> jcamp_file = "C:\Data\gc01_0812_066.jdx"
>>> data = JCAMP_reader(jcamp_file)
 -> Reading JCAMP file 'C:\Data\gc01_0812_066.jdx'
```

The intensity matrix object is built by binning:

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

In this example, we show how to btain the dimensions of the newly created intensity matrix, then loop over all ion chromatograms, and for each ion chromatogram apply Savitzky-Golay noise filter and tophat baseline correction:

```
>>> n_scan, n_mz = im.get_size()
>>> from pyms.Noise.SavitzkyGolay import savitzky_golay
>>> from pyms.Baseline.TopHat import tophat
>>> for ii in range(n_mz):
    print "working on IC", ii
```

```
    ic = im.get_ic_at_index(ii)
    ic1 = savitzky_golay(ic)
    ic_smooth = savitzky_golay(ic1)
    ic_base = tophat(ic_smooth, struct="1.5m")
    im.set_ic_at_index(ii, ic_base)
```

The resulting noise and baseline corrected ion chromatogram is saved back into the intensity matrix. As this example illustrates, PyMS depends on Python and can be used in the exactly the same way under any operating system that supports Python. For more advanced use refer to subsequent chapters.

Chapter 2

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),[1] and Joint Committee on Atomic and Molecular Physical Data (JCAMP-DX)[2] formats. These formats are essentially recommendations, and it is up to individual vendors of mass spectrometry processing software to implement properly "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. PyMS makes the following minimum set of assumptions about the information contained in the data file:

- 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 Creating a "GCMS_data" object

2.2.1 Reading JCAMP GC-MS data

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

[1] ANDI-MS was developed by the Analytical Instrument Association.

[2] JCAMP-DX is maintained by the International Union of Pure and Applied Chemistry.

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.[3] This file can be loaded in Python as follows:

```
>>> 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" is a GC-MS experiment converted to ANDI-MS format from Agilent ChemStation (the same data as in the 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

[*Examples below are in pyms-test/20a and pyms-test/20b*]

[3]ChemSW, Inc.

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 obtaining 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
>>>
```

The entire raw data can be exported to a file with 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.
```

Chapter 3

GC-MS data derived objects

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, the resulting object is called intensity matrix. In this chapter the methods for converting the raw GC-MS data to an intensity matrix object are illustrated.

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 Discussion of Binning Boundaries

For any chemical element X, let $w(x)$ be the atomic weight of X, and

$$\delta(X) = \frac{w(X) - \{w(X)\}}{w(X)}, \tag{3.1}$$

where $\{a\}$ is the integer value of a (rounded to the nearest integer).

For example, for hydrogen $\delta(^1\text{H}) = \frac{1.007825032-1}{1.007825032} = 0.0076$. Similarly $\delta(^{12}\text{C}) = 0$, $\delta(^{14}\text{N}) = 0.00022$, $\delta(^{16}\text{O}) = -0.00032$, etc.

Let also $\Delta(X) = w(X) - \{w(x)\}$. Then $-0.023 < \Delta(^{31}\text{P}), \Delta(^{28}\text{Si}) < 0$.

Let a compound undergo GC-MS and let Y be one of it's fragments. If Y consists of $k_1$atoms of type X_1, k_2 atoms of type X_2,....., k_r atoms of type X_r, then $\Delta(Y) = k_1 * \Delta(X_1) + k_2 * \Delta(X_2) + ... + k_r * \Delta(X_r)$.

The fragment will usually not contain more than 2 or 3 P or Si atoms and if it's molecular weight is less than 550 it may not contain more than 35 O atoms, so $\Delta(Y) \geq -0.023 * 5 - 0.00051 * 35 = -0.133$.

On the other hand, of Y contains k H atoms and m N atoms, then $\Delta(Y) \leq k * 0.00783 + m * 0.00051$. Since for each two hydrogen atoms at least one carbon (or heavier) atom is needed, giving the limit of no more than 80 hydrogen atoms. Therefore in this case (i.e. H and C atoms only)$\Delta(Y) \leq 80 * 0.00783 = 0.63$. If carbon is replaced by any heavier atom, at least 2 hydrogen atoms will be eliminated and $\Delta(Y)$ will become even smaller.

If the molecular weight of Y does not exceed 550 (typically the largest mass scanned for in a GC-MS setup) then $\mathbf{-0.133 \leq \Delta(Y) \leq 0.63}$. This means that if we set our binning boundaries to $(-0.3, 0.7)$ or $(-0.2, 0.8)$ the opportunity for having a fragment whose molecular weight is very close to the boundary is minimised.

Since the resolution of MS is at least 0.1 dalton, we may assume that it's error does not exceed 0.05, and MS accuracy will not cause additional problems.

3.1.2 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 `build_intensity_matrix()` and `build_intensity_matrix_i()`, available in "pyms.GCMS.Function".

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()
(9865, 551)
```

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 can be obtained as follows:

```
>>> masses = im.get_mass_list()
>>> print masses[:10]
[50.0, 51.0, 52.0, 53.0, 54.0, 55.0, 56.0, 57.0, 58.0, 59.0]
```

The last command prints the first ten masses. The methods `im.get_min_mass()` and `get_max_mass()` return the minimum and maximum mass:

```
>>> print im.get_min_mass()
50.0
>>> print im.get_max_mass()
600.0
```

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 can be found by using the methods `get_index_of_mass()`:

```
>>> index = im.get_index_of_mass(73.3)
>>> print index
23
```

The value of the closest mass can be returned by the method `get_mass_at_index()`:

```
>>> print im.get_mass_at_index(index)
73.0
```

A mass of 73.0 is returned in this example.

3.1.3 Build intensity matrix parameters

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

The bin interval can be set to values other than one, and binning boundaries can also be adjusted. In the example below, 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()
```

In this example there are 9865 scans (as before), but 1101 bins.

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.

3.1.4 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 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 mass values and corresponding intensities, respectively. MassSpectrum is returned by the IntensityMatrix method `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 as follows:

```
>>> 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 object 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 to LECO CSV format. This facility is useful for import into other analytical software packages. The format has 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")
```

3.5 Importing ASCII data

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

The LECO CSV data format can be used to import ASCII data directly into an IntensityMatrix object. The data must follow the format outlined above. For example, the file saved above can be read and compared to the original:

```
>>> from pyms.GCMS.Class import IntensityMatrix
>>>
>>> iim = IntensityMatrix([0],[0],[[0]])
>>>
>>> iim.import_leco_csv("output/data_leco.csv")
>>>
```

```
>>> print im.get_size()
>>> print iim.get_size()
```

The line “IntensityMatrix([0],[0],[[0]])” is required to create an empty IntensityMatrix object.

Chapter 4

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 cropped. 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.Window 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 Window Averaging on Intensity Matrix

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

In the previous section, window averaging was applied to an Ion Chromatogram object (in that case a TIC). Where filtering is to be performed on all Ion Chromatograms, the `window_smooth_im()` function may be used instead.

The use of this function is identical to the Ion Chromatogram `window_smooth()` function, except that an Intensity Matrix is passed to it.

For example, to perform window smoothing on an Intensity Matrix object with a 5 point window and mean window smoothing:

```
>>> from pyms.Noise.Window import window_smooth_im()
... im is a PyMS IntensityMatrix object
>>> im_smooth = window_smooth_im(im, window = 5, median = False)
```

4.4.3 Savitzky–Golay noise filter

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

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.

4.4.4 Savitzky-Golay Noise filtering of Intensity Matrix Object

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

The `savitzky_golay()` function described in the previous section acts on a single Ion Chromatogram. Where it is desired to perform Savitzky Golay filtering on the whole Intensity matrix the function `savitzky_golay_im()` may be used as follows:

```
>>> from pyms.Noise.SavitzkyGolay import savitzky_golay_im
... im is a PyMS IntensityMatrix object
>>> im_smooth = savitzky_golay(im)
```

4.5 Baseline correction

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

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 [2], 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 [3].

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 [4]) must be installed, or the local versions of scipy's ndimage must be installed. For the SciPy/ndimage installation instructions please see the section 1.3.4.

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/'.

4.5.1 Tophat Baseline correction on an Intensity Matrix object

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

The function outlined in the instructions above `tophat()`, acts on a single Ion Chromatogram. To perform baseline correction on an Intenstity Matrix object (i.e. on all Ion Chromatograms) the `tophat_im()` function may be used.

Using the same definition for ”`struct`” as above, use of the `tophat_im()` function is as follows:

```
>>> from pyms.Baseline.TopHat import tophat_im()
... im is an Intensity Matrix object
>>> im_base_corr = tophat(im, struct="1.5m")
```

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)
...
```

Alternatively, the filtering may be performed on the Intensity Matrix without using a `for` loop, as outlined in the sections above. However filtering by Ion Chromatogram in a `for` loop as described here is much faster.

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")
```

Chapter 5

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 (the class defined in `pyms.Peak.Class`). PyMS provides functions to detect peaks and create (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. In the example below, 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)[5]. 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 Noise analysis for peak filtering

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

In the previous section the cutoff parameter for peak filtering was set by the user. This can work well for individual data files, but can cause problems when applied to large experiments with many individual data files. Where experimental conditions have changed slightly between experimental runs, the ion intensity over the GC-MS run may also change. This means that an inflexible cutoff value can work for some data files, while excluding too many, or including too many peaks in other files.

An alternative to manually setting the value for cutoff, is to use the pyms function `window_analyzer()`. This function examines a Total Ion Chromatogram (TIC) and computes a value for the median absolute deviation in troughs between peaks. This gives an approximate threshold value above which false peaks from noise should be filtered out.

To compute this noise value:

```
>>> from pyms.Noise.Analysis import window_analyzer
... data is a GCMS data object
>>> tic = data.get_tic()
>>> noise_level = window_analyzer(tic)
```

Now the usual peak deconvolution steps are performed, and the peak list is filtered using this noise value as the cutoff:

```
... pl is a peak list, n is number of ions above threshold
>>> peak_list = num_ions_threshold(pl, n, noise_level)
```

5.5 Peak area estimation

[*This example is in pyms-test/55*]

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()`, or `set_ion_areas()` method of the peak object.

The total 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)
...
```

5.5.1 Individual Ion Areas

[*This example is in pyms-test/56*]

While the previous approach uses the sum of all areas in the peak to estimate the peak area, the user may also choose to record the area of each individual ion in each peak.

This can be useful when the intention is to later perform quantitation based on the area of a single characteristic ion for a particular compound. It is also essential if using the Common Ion Algorithm for quantitation, outlined in section 6.4.

To set the area of each ion for each peak, the following code is used:

```
>>> from pyms.Peak.Function import peak_top_ion_areas
>>> for peak in peak_list:
>>>     area_dict = peak_top_ions_areas(intensity_matrix, peak)
>>>     peak.set_ion_areas(area_dict)
...
```

This will set the areas of the 5 most abundant ions in each peak. If it is desired to record more than the top five ions, the argument `num_ions=x` should be supplied, where `x` is the number of most abundant ions to be recorded. e.g.

```
>>>      area_dict = peak_top_ions_areas(intensity_matrix, peak, num_ions=10)
```

will record the 10 most abundant ions for each peak.

The individual ion areas can be set instead of, or in addition to the total area for each peak.

5.5.2 Reading the area of a single ion in a peak

If the individual ion areas have been set for a peak, it is possible to read the area of an individual ion for the peak. e.g.

```
>>> peak.get_ion_area(101)
```

will return the area of the m/z value 101 for the peak. If the area of that ion has not been set (i.e. it was not one of the most abundant ions), the function will return `None`.

Chapter 6

Peak alignment by dynamic programming

PyMS provides functions to align GC-MS peaks by dynamic programming [6]. 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.

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

from pyms.GCMS.IO.ANDI.Function import ANDI_reader
from pyms.GCMS.Function import build_intensity_matrix_i
from pyms.Noise.SavitzkyGolay import savitzky_golay
from pyms.Baseline.TopHat import tophat
from pyms.Peak.Class import Peak
from pyms.Peak.Function import peak_sum_area

from pyms.Deconvolution.BillerBiemann.Function import BillerBiemann, \
    rel_threshold, num_ions_threshold

# deconvolution and peak list filtering parameters
points = 9; scans = 2; n = 3; t = 3000; r = 2;

andi_file = "/x/PyMS/data/a0806_077.cdf"

data = ANDI_reader(andi_file)

# integer mass
im = build_intensity_matrix_i(data)

# get the size of the intensity matrix
n_scan, n_mz = im.get_size()

# smooth data
for ii in range(n_mz):
    ic = im.get_ic_at_index(ii)
    ic1 = savitzky_golay(ic)
    ic_smooth = savitzky_golay(ic1)
    ic_base = tophat(ic_smooth, struct="1.5m")
    im.set_ic_at_index(ii, ic_base)

# do peak detection on pre-trimmed data

# get the list of Peak objects
pl = BillerBiemann(im, points, scans)

# trim by relative intensity
apl = rel_threshold(pl, r)

# trim by threshold
peak_list = num_ions_threshold(apl, n, t)
```

```

print "Number of Peaks found:", len(peak_list)

# ignore TMS ions and set mass range
for peak in peak_list:
    peak.crop_mass(50,540)
    peak.null_mass(73)
    peak.null_mass(147)
    # find area
    area = peak_sum_area(im, peak)
    peak.set_area(area)

```

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 1.3.3.*

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.

A script for running the dynamic programming alignment on these experiments 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 script reads the experiment files from the directory where they were stored (`61a/output`), and creates a list of the loaded `Experiment` objects. Each experiment object is converted into an `Alignment` object with `exprl2alignment()`. In this example, there is only one experimental condition so the alignment object is only for within group alignment (this special case is called 1-alignment). The variable F1 is a Python list containing three alignment objects.

The pairwise alignment is then performed. The parameters for the alignment by dynamic programming are: Dw, the retention time modulation in seconds; and Gw, the gap penalty. These parameters are explained in detail in [6]. `PairwiseAlignment()`, defined in `pyms.Peak.List.DPA.Class` is a class that calculates the similarity between a ll peaks in one sample with those of another sample. This is done for all possible pairwise alignments (2-alignments). The output of `PairwiseAlignment()` (T1) is an object which contains the dendrogram tree that maps the similarity relationship between the input 1-alignments, and also 1-alignments themselves.

The function `align_with_tree()` takes the object T1 and aligns the individual alignment objects according to the guide tree. In this example, the individual alignments 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 that 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.

Finally, the resulting 3-alignment is saved by writing 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 `62/output/`.

The file 'area1.csv' 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.

6.3 Between-state alignment of peak lists from multiple experiments

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

In the previous example the list of peaks were aligned within a single experiment with multiple replicates ("within-state alignment"). In practice, it is of more interest to compare the two experimental states. In a typical experimental setup there can be multiple replicate experiments on each experimental state or condition. To analyze the results of such an experiment statistically, the list of peaks need to be aligned within each experimental state and also between the states. The result of such an alignment would be the data matrix of integrated peak areas. The data matrix contains a row for each sample and the number of columns is determined by the number of unique peaks (metabolites) detected in all the experiments.

In principle, all experiments could be aligned across conditions and replicates in the one process. However, a more robust approach is to first align experiments within each set of replicates (within-state alignment), and then to align the resulting alignments (between-state alignment) [6].

This example demonstrates how the peak lists from two cell states are aligned. The cell state, A, consisting of three experiments aligned in *pyms-test/61a* ('a0806_077', 'a0806_078', 'a0806_079') and cell state, B, consisting of three experiments aligned in *pyms-test/61b* ('a0806_140', 'a0806_141', and 'a0806_142').

The between group alignment can be performed by the following alignment commands.

```
# between replicates alignment parameters
Db = 10.0 # rt modulation
Gb = 0.30 # gap penalty

print 'Aligning input {1,2}'
T9 = PairwiseAlignment([A1,A2], Db, Gb)
A9 = align_with_tree(T9)

A9.write_csv('output/rt.csv', 'output/area.csv')
```

where A1 and A2 are the results of the within group alignments (as above) for group A and B, respectively.

In this example the retention time tolerance for between-state alignment is greater compared to the retention time tolerance for the within-state alignment as we expect less fidelity in retention times between them. The same functions are used for the within-state and between-state alignment. The result of the alignment is saved to a file as the area and retention time matrices (described above).

6.4 Common Ion Area Quantitation

[*This example is in pyms-test/64*]

The `area.csv` file produced in the preceding section lists the total area of each peak in the alignment. The total area is the sum of the areas of each of the individual ions in the peak. While this approach produces broadly accurate results, it can result in errors where neighbouring peaks or unfiltered noise add to the peak in some way.

One alternative to this approach is to pick a single ion which is common to a particular peak (compound), and to report only the area of this ion for each occurance of that peak in the alignment. Using the method `common_ion()` of the PyMS class

Alignment, PyMS can select an ion for each aligned peak which is both abundant and occurs most often for that peak. We call this the 'Common Ion Algorithm' (CIA).

To implement this in PyMS, it is essential that the individual ion areas have been set (see section 5.5.1).

6.4.1 Using the Common Ion Algorithm

When using the CIA for area quantitation, a different method of the class Alignment is used to write the area matrix; `write_common_ion_csv()`. This requires a list of the common ions for each peak in the alignment. This list is generated using the Alignment class method `common_ion()`.

Continuing from the previous example, the following invokes common ion filtering on previously created alignment object 'A9':

```
>>> common_ion_list = A9.common_ion()
```

The variable 'common_ion_list' is a list of the common ion for each peak in the alignment. This list is the same length as the alignment. To write peak areas using common ion quantitation:

```
>>> A9.write_common_ion_csv('output/area_common_ion.csv',common_ion_list)
```

Chapter 7

The Display module

PyMS has graphical capabilities to display information such as ion chromatogram objects (ICs), total ion chromatogram (TIC), and detected lists of peaks. This functionality is provided by the package matplotlib, which must be installed in order to use the Display module and to run the scripts referred to in this chapter. The instructions for this are provided in 1.3.5.

7.1 Displaying the TIC

[*This example is in pyms-test/70a*]

TIC must first be extracted from the data (see section 3.3). Once the TIC object is created, a simple call to the function `plot_ics()` displays the TIC in a graphical window. In addition to the TIC, two strings may be passed to plot_ics() which label the data (in this case 'TIC'), and the overall plot ('TIC for gc01_0812_066').

```
>>> plot_ics(tic, 'TIC','TIC for gc01_0812_066')
```

The window in figure 7.1 should now be displayed:

To zoom in on a portion of the plot, select the button, hold down the left mouse button while dragging a rectangle over the area of interest. To return to the original view, click on the button.

The button allows panning across the zoomed plot.

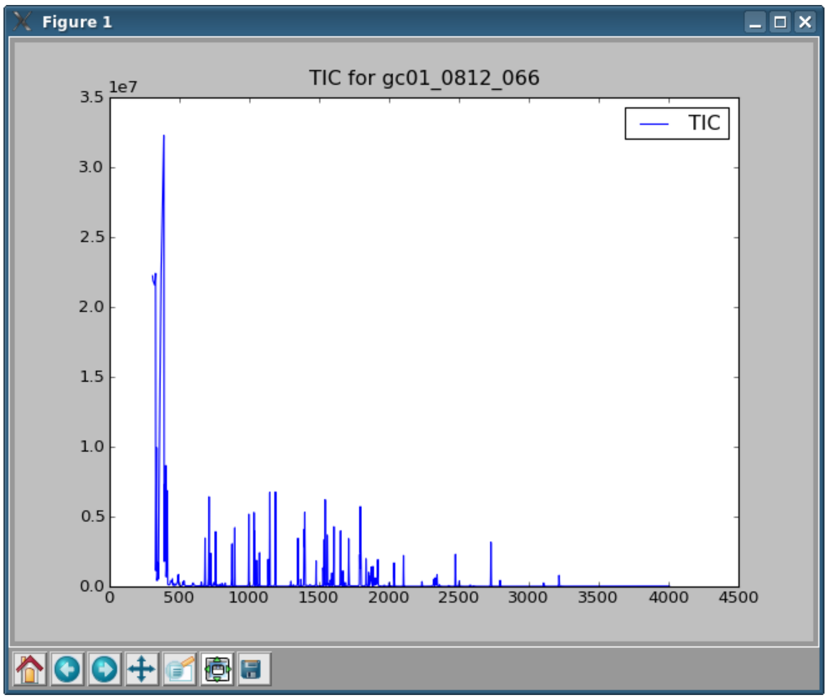


Figure 7.1: Graphics window displayed by the script 70a/proc.py

7.2 Displaying multiple ion chromatogram objects

[*This example is in pyms-test/70b*]

The Display module can plot multiple ICs and the TIC on the same figure, as shown in the following example. First, a list of ion chromatogram objects is created:

```
>>> tic = data.get_tic()
>>> ic = im.get_ic_at_mass(73)
>>> ic1 = im.get_ic_at_mass(147)
>>> ics = [tic, ic, ic1]
```

This list is passed to the function `plot_ics()`, along with a list of strings to label each ion chromatogram:

```
>>> plot_ics(ics, ['TIC', '73','147'], 'TIC and ICs for m/z = 73 & 147')
```

This results in figure 7.2 being displayed:

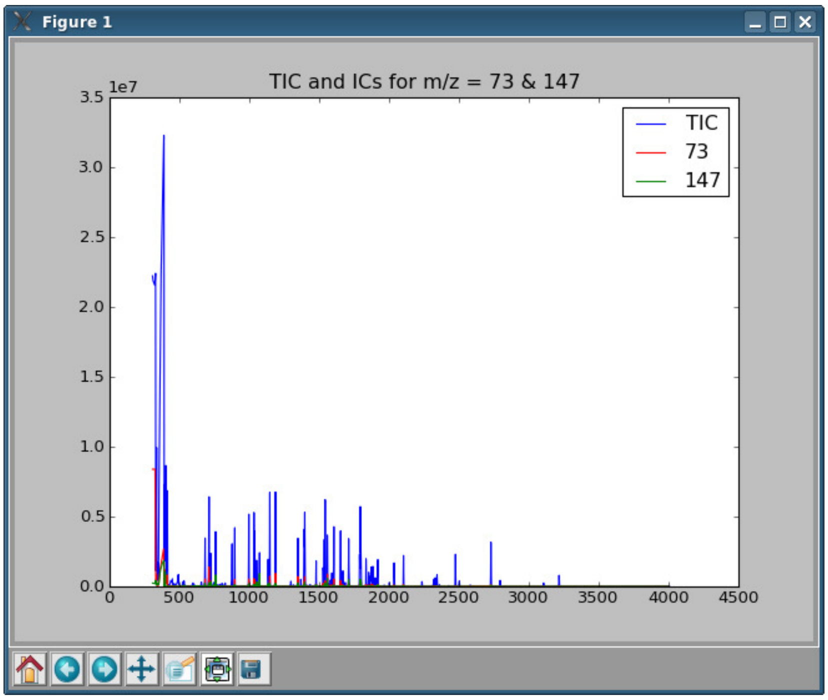


Figure 7.2: Graphics window displayed by the script 70b/proc.py

7.3 Displaying the mass spectrum

[*This example is in pyms-test/70c*]

The pyms Display module can also be used to display individual mass spectra. The following selects the mass spectrum of interest:

```
... im is an intensity matrix
>>> ms = im.get_ms_at_index(1024)
```

The pyms function `plot_ms()` can then be called to display the mass spectrum to the screen.

```
>>> plot_ms(ms, 'Mass Spectrum at index 1024')
```

The resulting window is shown below in figure 7.3

7.4 Displaying detected peaks on the TIC plot

[*This example is in pyms-test/71*]

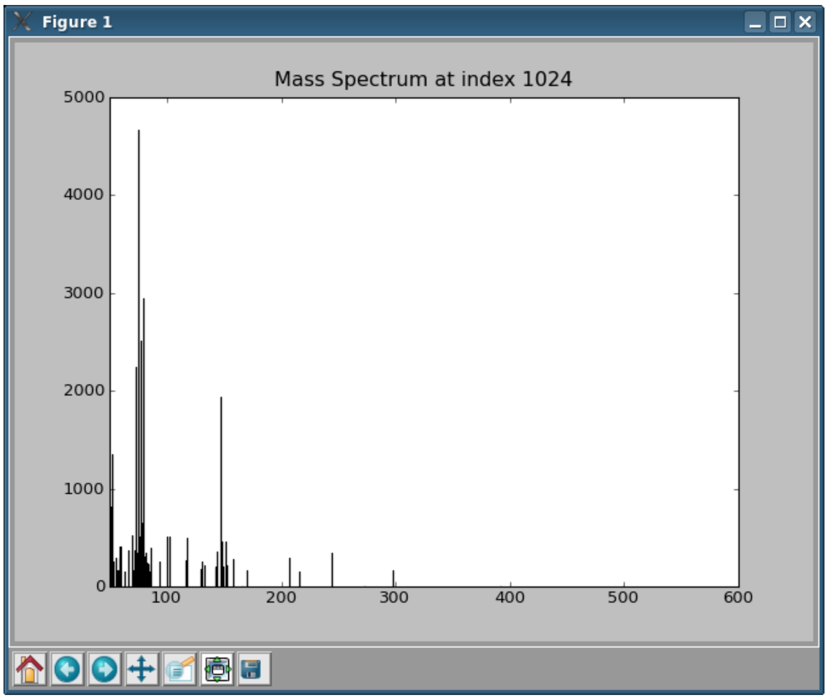


Figure 7.3: Graphics window displayed by the script 70c/proc.py

The Display class is a more powerful implementation of plotting which allows plotting of detected peaks and user interaction with the plot figure.

In order to plot a list of peaks, the peaks must be created first. The example pyms-test/71 contains the script `proc_save_peaks.py` which produces such a peak list. For more information on detecting peaks see section 5.2. The function `store_peaks()` in `proc_save_peaks.py` stores the peaks, while `load_peaks()` in `proc.py` loads them for the Display class to use.

When using the Display Class, a series of plots is added to the figure one at a time before the final plot is displayed. The first step is always creating an instance of the Display class:

```
>>>display = Display()
```

Next some ics and the TIC are plotted. Unlike in simple plotting, the TIC is plotted using a separate function. This function ensures that the TIC is always plotted in blue for easy reference. The legend for each IC is supplied to the functions, but the overall figure label is not supplied at this time.

```
>>> display.plot_ics(ics, ['73','147'])
>>> displat.plot_tic(tic, 'TIC')
```

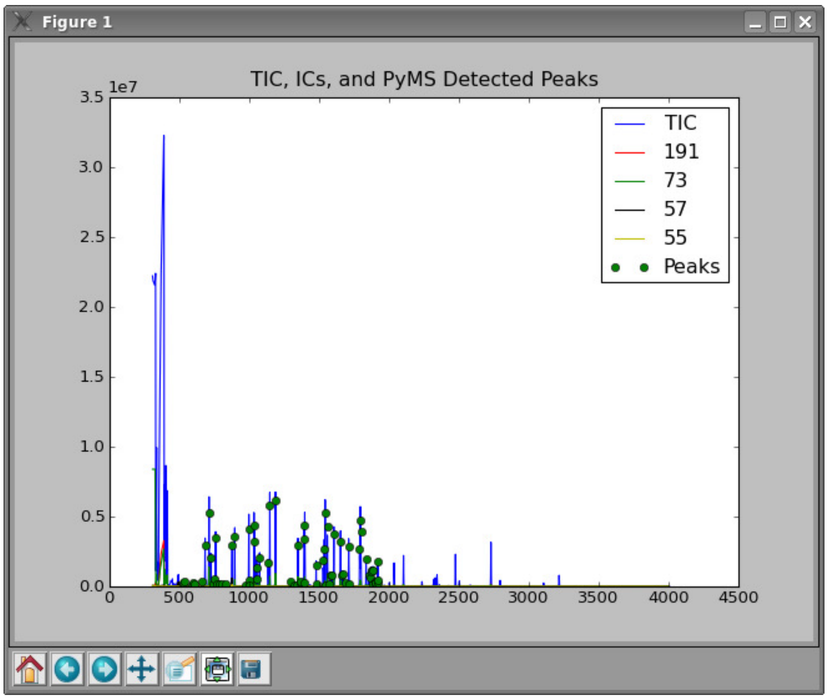


Figure 7.4: Graphics window displayed by the script 71/proc.py

The function `plot_peaks()` adds the PyMS detected peaks to the figure.

```
>>>display.plot_peaks(peak_list, 'Peaks')
\end{verbatgoogle.com.au/im}

Finally, the function {\tt do\_plotting()} is called to draw the figure with
labels and display the plot.

\begin{verbatim}
>>> display.do_plotting('TIC, ICs, and PyMS Detected Peaks')
```

This should result in figure 7.4 being displayed:

7.4.1 User interaction with the plot window

When using the Display class, the resulting figure window can be used to access data about the displayed peaks.

Clicking on a peak causes a list of the 5 highest intensity eluting ions at that peak to be written to the screen in order. Clicking a mouse button over one of the peaks should result in output similar to the following:

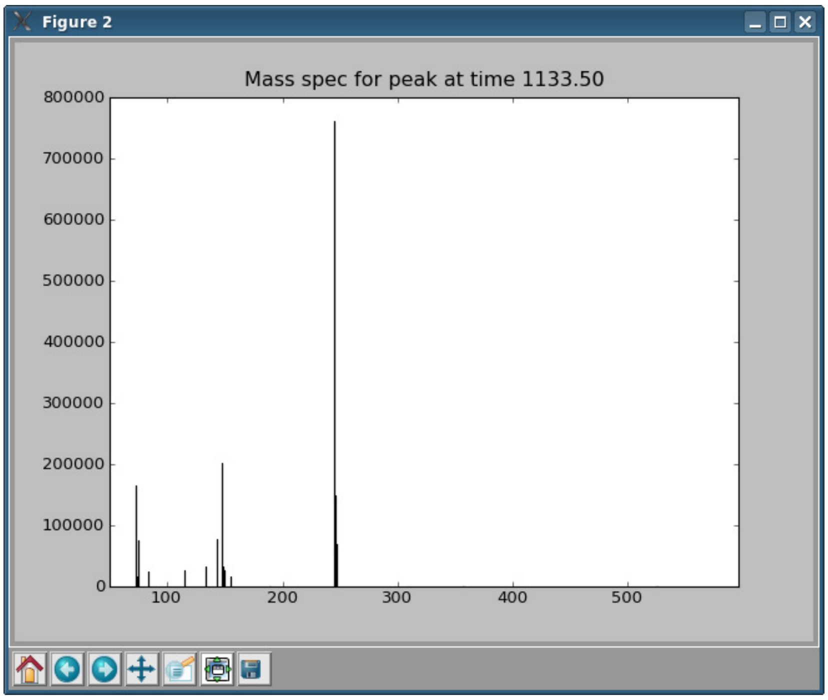


Figure 7.5: The mass spectrum displayed by PyMS when a peak in the graphics window of Figure 7.4 is clicked on

```
>>> mass      intensity
>>> 273      1003678.47619
>>> 73       625396.428571
>>> 147      526953.333333
>>> 363      255903.714286
>>> 347      241031.333333
```

In addition, clicking a mouse button other than the left button on a peak displays the mass spectrum at the peak in a new window (figure 7.5):

Clicking on other peaks will display further mass spectrums for those peaks in new windows.

Chapter 8

Parallel processing with PyMS

8.1 Requirements

Using PyMS parallel capabilities requires installation of the package 'mpi4py', which provides bindings of the Message Passing Interface (MPI) for the Python programming language. This package can be downloaded from `http://code.google.com/p/mpi4py/`. Since 'mpi4py' provides only Python bindings, it requires an MPI implementation. We recommend using mpich2:
`http://www.mcs.anl.gov/research/projects/mpich2/`
We show the installation of 'mpich2' and 'mpi2py' on Linux system from software distributions downloaded from the projects' web site.

8.1.1 Installation of 'mpich2'

1. From the mpich2 project web site download the current distribution of mpich2 (in our case the file 'mpich2-1.2.1p1.tar.gz').

2. Prepare the directory for mpich2 installation. In this example we have chosen to use /usr/local/mpich2/. Our version of mpitch2 is 1.2.1, and to allow for the installation of different version later, we create a subdirectory "1.2.1",

   ```
   $ mkdir -vp /usr/local/mpich2/1.2.1
   ```

   The above command will make the directory /usr/local/mpich2/ and also /usr/local/mpich2/1.2.1. Note that /usr/local is usually owned by root, and the above commands may require root privileges.

3. Unpack this file and change to the source code directory:

```
$ tar xvfz mpich2-1.2.1p1.tar.gz
$ cd  mpich2-1.2.1p1
```

4. Configure, compile, and install mpich2:

```
$ ./configure --prefix=/usr/local/mpich2/1.2.1 --enable-sharedlibs=gcc
$ make
$ make install
```

   If /usr/local/mpich2/1.2.1 is owned by rood, the above command may require root privileges.

8.1.2 Installation of 'mpi4py'

1. From the mpi4py project web site download the current distribution of mpi4py (in our case the file 'mpi4py-1.2.1.tar.gz').
2. Unpack this file and change to the source code directory:

```
$ tar xvfz mpi4py-1.2.1.tar.gz
$ cd mpi4py-1.2.1
```

3. Edit the file 'mpi.cfg' to reflect the location of mpich2. In our case this file after editing contained the following:

```
# MPICH2
[mpi]
mpi_dir              = /usr/local/mpich2/1.2.1
mpicc                = %(mpi_dir)s/bin/mpicc
mpicxx               = %(mpi_dir)s/bin/mpicxx
```

4. Install mpi4py:

```
$ python setup.py install
```

5. Check that mpi4py works:

```
$ python
Python 2.5.2 (r252:60911, Sep 10 2008, 14:39:22)
[GCC 4.1.1 20070105 (Red Hat 4.1.1-52)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import mpi4py
>>>
```

   If the above command import produced no output, mpi4py is installed properly and ready to use.

8.2 Background to using PyMS in parallel

Any processing that loops through ion chromatograms or mass spectra can be performed in parallel, by distributing the processing of individual ion chromatograms or mass spectra to different CPUs by using the efficient MPI mechanism.

Before the parallel processing can be deployed, data needs to be binned to produce an IntensityMatrix object, as described in the Section 3.1. This is essentially a two dimensional matrix, with ion chromatograms along one dimension and mass spectra along the other dimension.

Consider the processing which applies a noise smoothing function to each ion chromatogram. We first read the raw data:

```
andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
data = ANDI_reader(andi_file)
```

Then build the intensity matrix, and get its dimensions:

```
im = build_intensity_matrix_i(data)
n_scan, n_mz = im.get_size()
```

The last command sets the variables n_scan and n_mz to the number scans and number of m/z values present in data, respectively. Processing of ion chromatograms with the noise smoothing function requires fetching of each ion chromatogram from the data, and application of the noise smoothing function. This can be achieved with a simple loop:

```
for ii in n_mz:
    print ii+1,
    ic = im.get_ic_at_index(ii)
    ic_smooth = window_smooth(ic, window=7)
```

This example epitomizes the typical processing required on the GC-MS data. Another, equally important processing, is that of individual mass spectra. In this case the same logic can be applied, except that one would loop over the other dimension of the IntensityMatrix object 'im'. That is, one would loop over all the scan indices, and use the method get_ms_at_index() to fetch individual mass spectra:

```
for ii in n_scan:
    print ii+1,
    ms = im.get_ms_at_index(ii)
    # here do something the the mass spectra 'ms'
```

Processing of data in this fashion is computationally intensive. A typical data set may consist of 3,000-10,000 scans and 500 m/z values. If complex processing algorithms are applied to each ion chromatogram (or mass spectra), the processing will quickly become computationally prohibitive.

The type of calculation illustrated above is an ideal candidate for parallelization because each ion chromatogram (or mass spectrum) are processed independently. PyMS takes advantage of this and allows one to harvest the power of multiple CPUs to speed-up the processing. To achieve this PyMS can distributes the loop from the above (either type, ie. over ion chromatograms or mass spectra) over the available CPUs, achieving a linear speed-up with the number of CPUs.

8.3 Using PyMS in parallel

Using PyMS in parallel requires a minimal intervention, only that special method of the IntensityMatrix object is invoked in the for loop described above. For looping over all ion chromatograms in parallel,

```
for ii in im.iter_ic_indices():
    print ii+1,
    ic = im.get_ic_at_index(ii)
    ic_smooth = window_smooth(ic, window=7)
```

The only change is that

```
for ii in n_mz:
```

is replaced with

```
for ii in im.iter_ic_indices()
```

The corresponding method for looping over all mass spectra would involve replacing:

```
for ii in n_scan:
```

with

```
for ii in im.iter_ms_indices()
```

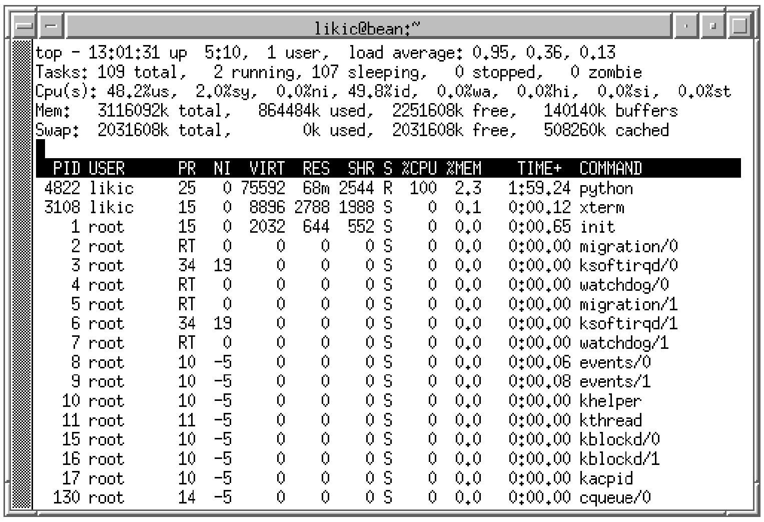
The special constructs `for ii in im.iter_ic_indices():` and `for ii in im.iter_ms_indices()` will distribute the calculation in parallel if MPI capability is available (ie. mpi4py is installed on the system, and multiple CPUs are available). If MPI capability is not available, the processing will be performed in a serial mode. Running in parallel also requires some prior preparations, as explained below.

Consider how the following script that performs noise smoothing example described above (named 'proc.py'). This script is can be run in serial or parallel mode.

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

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

from pyms.GCMS.IO.ANDI.Function import ANDI_reader
from pyms.GCMS.Function import build_intensity_matrix_i
from pyms.Noise.Window import window_smooth

# read the raw data as a GCMS_data object
andi_file = "/x/PyMS/data/gc01_0812_066.cdf"
data = ANDI_reader(andi_file)

# build the intensity matrix
im = build_intensity_matrix_i(data)

# get the size of the intensity matrix
n_scan, n_mz = im.get_size()
print "Size of the intensity matrix is (n_scans, n_mz):", n_scan, n_mz

# loop over all m/z values, fetch the corresponding IC, and perform
# noise smoothing
for ii in im.iter_ic_indices():
    print ii+1,
    ic = im.get_ic_at_index(ii)
    ic_smooth = window_smooth(ic, window=7)
```

A simple running of this script will produce a serial run without any warning messages:

```
$ python proc.py
 -> Reading netCDF file '/x/PyMS/data/gc01_0812_066.cdf'
Size of the intensity matrix is (n_scans, n_mz): 9865 551
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
... [ further output deleted ] ...
```



```
likic@bean:~
top - 13:01:31 up  5:10,  1 user,  load average: 0.95, 0.36, 0.13
Tasks: 109 total,   2 running, 107 sleeping,   0 stopped,   0 zombie
Cpu(s): 48.2%us,  2.0%sy,  0.0%ni, 49.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3116092k total,   864484k used,  2251608k free,   140140k buffers
Swap:  2031608k total,        0k used,  2031608k free,   508260k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4822 likic     25   0 75592  68m 2544 R  100  2.3   1:59.24 python
 3108 likic     15   0  8896 2788 1988 S    0  0.1   0:00.12 xterm
    1 root      15   0  2032  644  552 S    0  0.0   0:00.65 init
    2 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0
    4 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    5 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    6 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
    8 root      10  -5     0    0    0 S    0  0.0   0:00.06 events/0
    9 root      10  -5     0    0    0 S    0  0.0   0:00.08 events/1
   10 root      10  -5     0    0    0 S    0  0.0   0:00.00 khelper
   11 root      11  -5     0    0    0 S    0  0.0   0:00.00 kthread
   15 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/0
   16 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/1
   17 root      10  -5     0    0    0 S    0  0.0   0:00.00 kacpid
  130 root      14  -5     0    0    0 S    0  0.0   0:00.00 cqueue/0
```

Figure 8.1: The xterm output of the program 'top' with PyMS running in serial mode on the computer with multiple CPUs

Inspection of the CPU usage during the execution of the program shows that only one CPU is utilised 100% (although multiple CPUs are available) as shown in Figure 8.2.

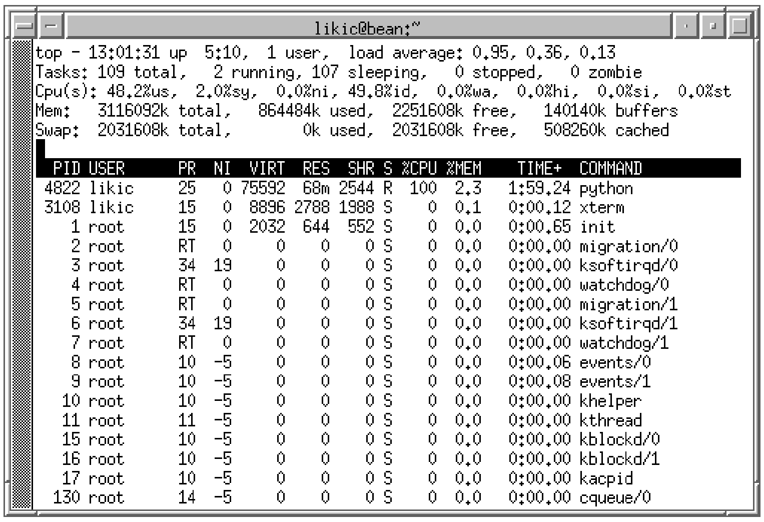
To run the above script in parallel, one needs first to start the mpitch2 process launcher, called 'mpd' (this is a program, in this example located in `/usr/local/mpich2/1.2.1/bin/mpd`,
see the section). This can be achieved as follows:

```
$ /usr/local/mpich2/1.2.1/bin/mpd --daemon
```

The above command start 'mpd' as a daemon (the program runs in the background, without a controlling terminal). A common problem that causes the above command is fail is the absence of the `.mpd.conf` file which 'mpd' requires to be present in the home directory of the user who is starting the process. Here is an excerpt from the 'mpd' help page:

```
A file named .mpd.conf file must be present in the user's home directory
with read and write access only for the user, and must contain at least
a line with MPD_SECRETWORD=<secretword>
To run mpd as root, install it while root and instead of a .mpd.conf file
use mpd.conf (no leading dot) in the /etc directory.'
```

Fixing this problem is simple, and requires creating the file `/.mpd.conf`, which in our case contains only one line:



```
likic@bean:~
top - 13:01:31 up  5:10,  1 user,  load average: 0.95, 0.36, 0.13
Tasks: 109 total,   2 running, 107 sleeping,   0 stopped,   0 zombie
Cpu(s): 48.2%us,  2.0%sy,  0.0%ni, 49.8%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3116092k total,   864484k used,  2251608k free,   140140k buffers
Swap:  2031608k total,        0k used,  2031608k free,   508260k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4822 likic     25   0 75592  68m 2544 R  100  2.3   1:59.24 python
 3108 likic     15   0  8896 2788 1988 S    0  0.1   0:00.12 xterm
    1 root      15   0  2032  644  552 S    0  0.0   0:00.65 init
    2 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0
    4 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    5 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    6 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
    8 root      10  -5     0    0    0 S    0  0.0   0:00.06 events/0
    9 root      10  -5     0    0    0 S    0  0.0   0:00.08 events/1
   10 root      10  -5     0    0    0 S    0  0.0   0:00.00 khelper
   11 root      11  -5     0    0    0 S    0  0.0   0:00.00 kthread
   15 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/0
   16 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/1
   17 root      10  -5     0    0    0 S    0  0.0   0:00.00 kacpid
  130 root      14  -5     0    0    0 S    0  0.0   0:00.00 cqueue/0
```

Figure 8.2: The xterm output of the program 'top' with PyMS running in serial mode on the computer with multiple CPUs

```
MPD_SECRETWORD=euSe0veo
```

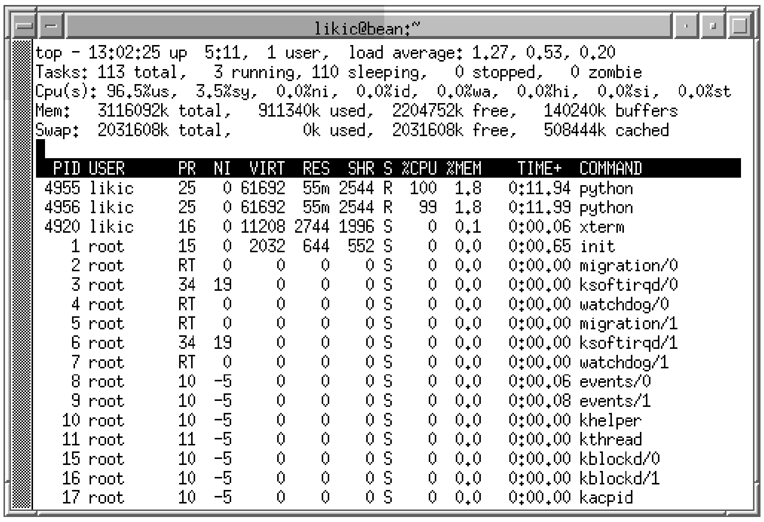
After this the 'mpd' can be launched. Running the PyMS script in the parallel mode requires the use of 'mpirun' command,

```
$ /usr/local/mpich2/1.2.1/bin/mpirun -np 2 python proc.py
```

The above command prepare 'python proc.py' to run in parallel, in this case by using two CPUS (`-np 2`). The execution produces the following output:

```
$ /usr/local/mpich2/1.2.1/bin/mpirun -np 2 python proc.py
 -> Reading netCDF file '/x/PyMS/data/gc01_0812_066.cdf'
 -> Reading netCDF file '/x/PyMS/data/gc01_0812_066.cdf'
Size of the intensity matrix is (n_scans, n_mz):Size of
the intensity matrix is (n_scans, n_mz): 9865 551
276 9865 551
1 277 2 278 3 279 4 280 5 281 6 282 7 283 8 284 9 285 10 286 11 287 12
288 13 289 14 290 15 291 16 292 17 293 18 294 19 295 20 296 21 297 22
298 23 299 24 300 25 301 26 302 27 303 28 304 2
... [ further output deleted ] ...
```

The above shows that two processes are active (each reading its own version of data). While the distribution of processing between the two processes has been achieved automatically by PyMS. Since both processes were started from the same terminal



```
likic@bean:~
top - 13:02:25 up  5:11,  1 user,  load average: 1.27, 0.53, 0.20
Tasks: 113 total,   3 running, 110 sleeping,   0 stopped,   0 zombie
Cpu(s): 96.5%us,  3.5%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3116092k total,   911340k used,  2204752k free,   140240k buffers
Swap:  2031608k total,        0k used,  2031608k free,   508444k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 4955 likic     25   0 61692  55m 2544 R  100  1.8   0:11.94 python
 4956 likic     25   0 61692  55m 2544 R   99  1.8   0:11.99 python
 4920 likic     16   0 11208 2744 1996 S    0  0.1   0:00.06 xterm
    1 root      15   0  2032  644  552 S    0  0.0   0:00.65 init
    2 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0
    4 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    5 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    6 root      34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
    8 root      10  -5     0    0    0 S    0  0.0   0:00.06 events/0
    9 root      10  -5     0    0    0 S    0  0.0   0:00.08 events/1
   10 root      10  -5     0    0    0 S    0  0.0   0:00.00 khelper
   11 root      11  -5     0    0    0 S    0  0.0   0:00.00 kthread
   15 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/0
   16 root      10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/1
   17 root      10  -5     0    0    0 S    0  0.0   0:00.00 kacpid
```

Figure 8.3: The xterm output of the program 'top' with PyMS running in parallel mode with two CPUs

their output is intermingled. This time the processing is using two CPUs, and this can be seen from the inspection of CPU utilisation, as shown in Figure 8.3. Also the execution of the script 'proc.py' is now two times faster.

This simple example shows how to speed-up PyMS processing on a common workstation with two CPUs. The MPI also allows PyMS processing to run on specialised computers with many CPUs, such as Linux clusters. The MPI implementation allows PyMS to be easily used in such distributed computing environments, much like in the example above. We have tested PyMS on Linux clusters, and the resulting speed-up is nearly linear with the number of CPUs employed (Figure 8.4).

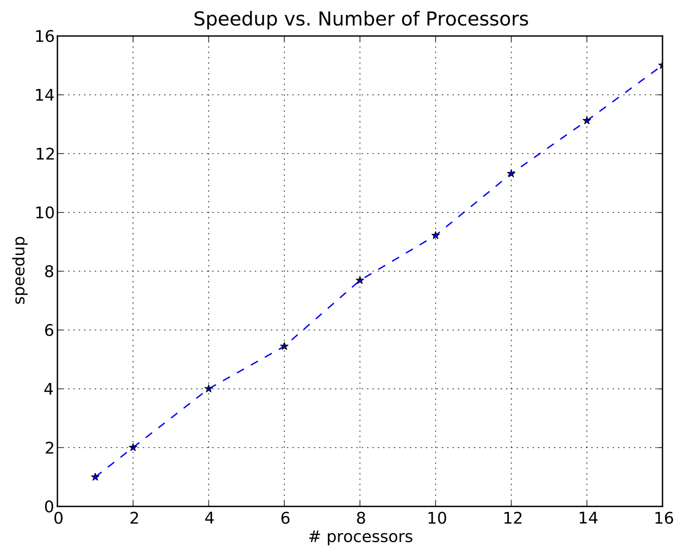


Figure 8.4: The speedup in PyMS processing when run in parallel on a Linux cluster as a function of a number of CPUs deployed

Chapter 9

GCMS Simulator

The PyMS GCMS data simulator is a useful tool for creating datasets with known peaks.

9.1 Setting up input data for the simulator

The PyMS GCMS data simulator package provides an Intensity Matrix object which can be processed by PyMS in the same way as any Intensity Matrix object derived from real data.

As input, the simulator requires a list of peaks, a list of times and a list of m/z channels. In this example we will use the time list, mass list and PyMS detected peak list from an existing real GCMS dataset.

For the purpose of demonstrating the simulator, only a portion of this dataset is examined. The following code selects the area of interest in the dataset and limits subsequent processing to this interval:

```
... data is a GCMS data object
>>> data.trim(4101, 4350)
```

9.1.1 Providing input to the simulator

[*This example is in pyms-test/90*]

The peaks are detected using the PyMS function `BillerBiemann()`, and subsequently filtered to remove those which do not meet certain requirements. This procedure is fully detailed in section 5.2.

Next the time list and mass list of the original data are copied to be used in the simulator

```
>>> mass_list = real_im.get_mass_list()
>>> time_list = real_im.get_time_list()
```

, where `real_im` is the Intensity Matrix object derived from the real data. (For more information about Intensity Matrix and data conversion refer to section (3.1).

9.2 Running the simulator

The function `gcms_sim()` takes three inputs: time list, mass list and peak list. It returns an Intensity Matrix object which can be manipulated in PyMS.

```
>>> sim_im = gcms_sim(time_list, mass_list, peak_list)
```

Using the PyMS display functionality described in chapter 7, the results of the simulator can be viewed. Figure 9.1 shows the IonChromatograms of the simulated Intensity Matrix.

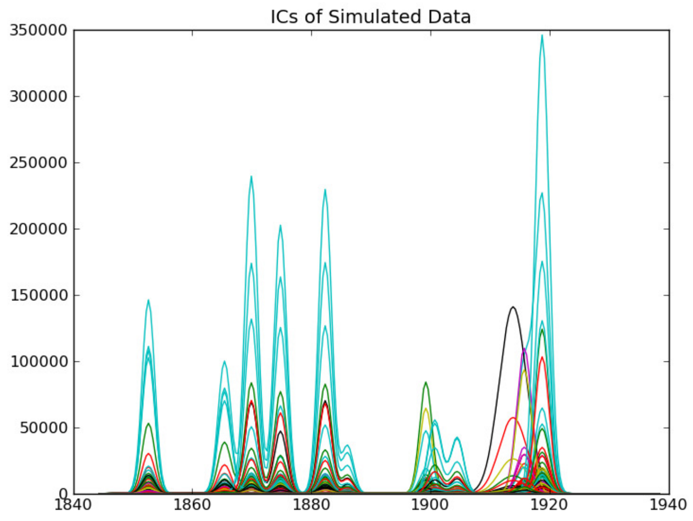


Figure 9.1: Graphics window displayed by the script 90/proc.py

9.3 Adding noise to the simulated data

Noise can be added to either the full Intenstity Matrix, or to individual Ion Chromatograms. Currently two noise models have been implemented in PyMS. These are:

1. Gaussian noise drawn from a normal distribution with constant scale across all scans.
2. Gaussian noise drawn from a normal distribution with scale varying with the intensity value at that point.

9.3.1 Constant scale noise

[*This example is in pyms-test/91*]

The Python package NumPy contains routines for pseudo random number generation. These "random" values are drawn from a normal distribution with a user defined scale over the same number of samples as the number of scans in a GC-MS experiment. This resulting list of numbers is a good approximation to gaussian noise on an m/z channel in the GC-MS experiment.

To add gaussian noise to an IC, the following code is used:

```
>>> ic = sim_im.get_ic_at_mass(73)
>>> scale = 1000
>>> add_gaussc_noise_ic(ic, scale)
```

The normal distribution from which the noise values are drawn in this example has a top value of 1000. This noisy IC can be displayed using the PyMS package Display, the resulting figure is shown in figure 9.2.

9.3.2 Variable scale noise

[*This example is in pyms-test/92*]

In reality, noise from a GC-MS experiment tends to have higher values in areas where peaks occur than in the valleys between peaks. To attempt to model this, a variable scale noise function has been implemented in PyMS. For a given time-point in an Ion Chromatogram, if the intensity at that time-point is greater than a user defined threshold value, the scale of the normal distribution from which the noise value is drawn will be proportional to the intensity at that point. The actual scale of the distribution at that point will be `scale * (intensity*proportion)`, if the intensity is above the `cutoff` value, and `scale` if below.

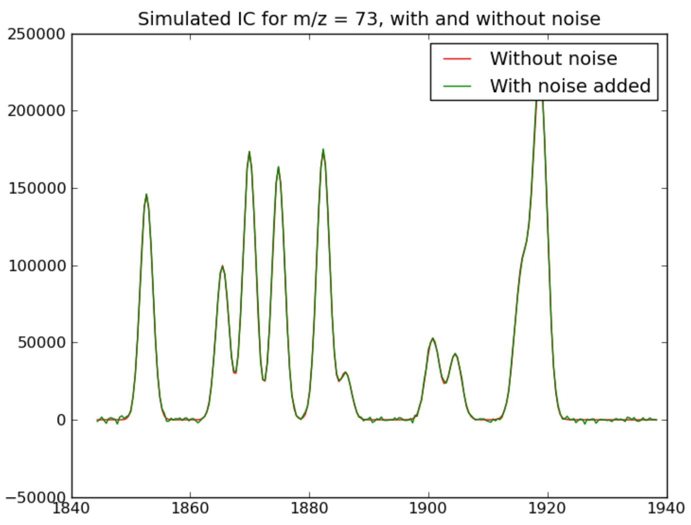


Figure 9.2: Graphics window displayed by the script 91/proc.py

To add variable scale gaussian noise to an IC, the following code is used:

```
>>> ic = sim_im.get_ic_at_mass(73)
>>> scale = 1000
>>> cutoff = 10000
>>> prop = 0.0003
>>> add_gaussv_noise_ic(ic, scale, cutoff, prop)
```

The resulting noisy IC is shown in figure 9.3

9.4 Adding noise to the whole simulated dataset

Often, the user may desire to add noise to all ICs in the dataset. This can be accomplished easily using the functions `add_gaussc_noise()` and `add_gaussv_noise()` The noise models in question are exactly the same as for the single IC noise functions above.

9.4.1 Constant scale gaussian noise

[*This example is in pyms-test/93*] For constant scale gaussian noise:

```
... sim_im is a simulated intensity matrix object
```

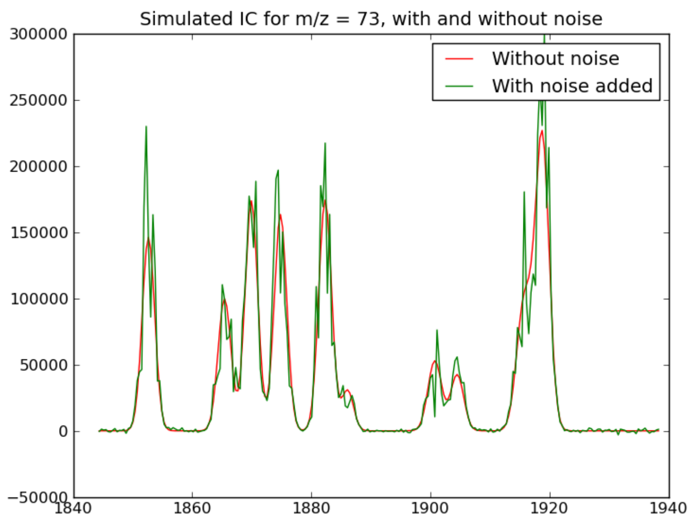


Figure 9.3: Graphics window displayed by the script 92/proc.py

```
>>> scale = 1000
>>> add_gaussc_noise(sim_im, scale)
```

This code adds noise to all Ion Chromatograms in the Intensity Matrix. These can be viewed in the PyMS package Display, with the resulting figure shown below (figure 9.4).

9.4.2 Variable scale gaussian noise

[*This example is in pyms-test/94*]

To apply variable scale gaussian noise to the whole simulated dataset:

```
>>> scale = 1000
>>> cutoff = 10000
>>> prop = 0.0003
>>> add_gaussv_noise(sim_im, scale, cutoff, prop)
```

The arguments supplied to `add_gaussv_noise()` relate to how the gaussian distribution supplying the noise values changes with the intensity. These parameters are further discussed in sub-section 9.3.2 above.

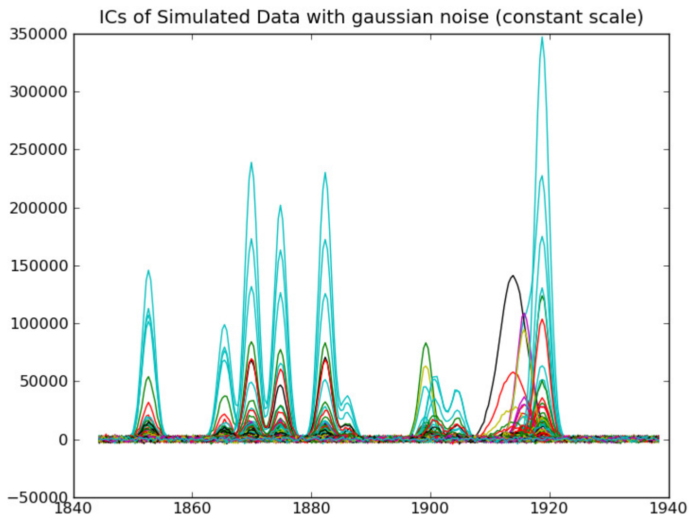


Figure 9.4: Graphics window displayed by the script 103/proc.py

9.5 Detecting Peaks in the Simulated data

[*This example is in pyms-test/95*]

The simulated intensity matrix may be processed in PyMS in the same way as an intensity matrix derived from real data. Applying the same techniques as outlined in section 5.2, the peaks of the simulated IM with added noise from 9.4.2 above can be found and displayed.

The found peaks are shown below in fig 9.6.

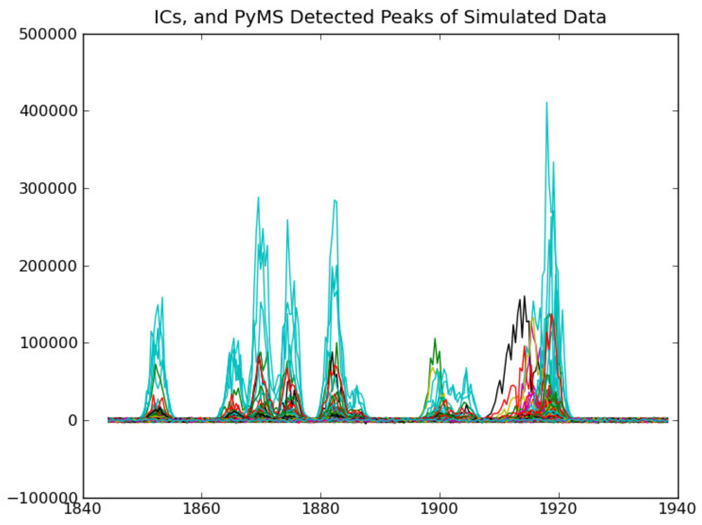


Figure 9.5: Graphics window displayed by the script 94/proc.py

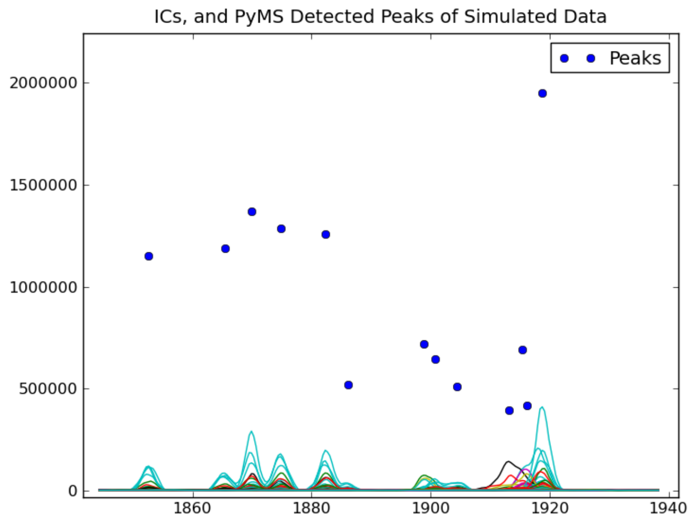


Figure 9.6: Graphics window displayed by the script 95/proc.py

Appendix A

Processing GC-MS data using PyMS

This Chapter presents different test examples where PyMS is used to process GC-MS data from different instruments.

A.1 Processing GC-Quad data

[*This example is in pyms-test/A1*]

A.1.1 Instrument

- Maker: Agilent
- Software: Chemstation

A.1.2 Raw Data

- File: gc01_0812_066.cdf
- Location: PyMS/data/
- Nature: Metabolite mix
- Processing: This is the raw cdf file generated by the instrument and no further processing was done using Chemstation.

A.1.3 Processing using PyMS

As detailed in the previous chapters, PyMS processes this data by binning the data, smoothing the data, removing the baseline, deconvolute peaks, filtering peaks, setting the mass range, removing uninformative ions and estimating peak areas.

GC-Quad data is made up of approximately 3 scans per second. As a result of this, the values of the following parameters in PyMS are unique for this dataset.

- *points* in Biller Biemann algorithm

  The parameter *points* in the Biller Biemann algorithm defines the width of the window that scans across the ion chromatogram in order to define a peak. The GC-Quad is made up of only a few scans per second and hence the peaks form over only a few number of scans resulting in a lower value defining the window width. In this example, this value is set to 3.

- *scans* in Biller Biemann algorithm

  The parameter *scans* in the Biller Biemann algorithm defines the number of consecutive scans across which a peak can be defined. The near optimal value of this parameter for GC-Quad data that is used in this example is 2.

- *threshold* in peak filtering

  In peak filtering, the parameter *threshold (t)* defines the intensity above which a certain number of ions needs to be represented to be accounted in and not to be filtered out. The near optimal value of this parameter for GC-Quad data that is used in this example is 10000.

A.2 Processing GC-TOF data

[*This example is in pyms-test/A2*]

A.2.1 Instrument

- Maker: Leco Pegasus
- Software: ChromaTOF

A.2.2 Raw Data

- File: MM-10.0_1_no_processing.cdf

- Location: PyMS/data/
- Nature: Metabolite mix
- Processing: This is the raw cdf file generated by the instrument and no further processing was done using ChromaTOF.

A.2.3 Processing using PyMS

PyMS processes this data through the following steps; binning the data, smoothing the data, removing the baseline, deconvolute peaks, filtering peaks, setting the mass range, removing uninformative ions and estimating peak areas.

GC-TOF data is made up of nearly 10 scans per second. As a result of this, the values of the following parameters in PyMS are unique for this dataset.

- *points* in Biller Biemann algorithm

  The GC-TOF is made up of more scans per second and as a result the data is more dense when compared to the GC-Quad data. Therefore the peaks form over more number of scans resulting in a higher value defining the window width. In this example, this value is set to 15.

- *scans* in Biller Biemann algorithm

  Due to the dense nature of the data, a peak can have its maximum value over more than 1 scan. In this example, this value is set to 3.

- *threshold* in peak filtering

  The near optimal value of this parameter for GC-TOF data that is used in this example is 4000.

Bibliography

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

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

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

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

[5] 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.

[6] 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.