pymzML Documentation

Release 0.7.4

Till Bald
Johannes Barth
Anna Niehues
Michael Specht
Michael Hippler
Christian Fufezan

January 14, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION

The latest Documentation was generated on: January 14, 2016

General information

Module to parse mzML data in Python based on cElementTree

T. Bald,
J. Barth,
A. Niehues,
M. Specht,
M. Hippler,
C. Fufezan

Contact information

Please refer to:

Dr. Christian Fufezan
Institute of Plant Biology and Biotechnology
Schlossplatz 7 , R 372
or
Hindenburgplatz 55 (mail)
University of Muenster
Germany
eMail: christian@fufezan.net
Tel: +049 251 83 24861

http://www.uni-muenster.de/Biologie.IBBP.AGFufezan

Summary

pymzML is an extension to Python that offers

- 1. easy access to mass spectrometry (MS) data that allows the rapid development of tools,
- 2. a very fast parser for mzML data, the standard in mass spectrometry data format and
- 3. a set of functions to compare or handle spectra.

Implementation

pymzML requires Python2.6.5+ and is fully compatible with Python3. The module is freely available on pymzml.github.com or pypi, published under LGPL and requires no additional modules to be installed.

Download

Get the latest version via github

https://github.com/pymzml/pymzML

or the latest package at

http://pymzml.github.com/dist/pymzml.tar.bz2
http://pymzml.github.com/dist/pymzml.zip

The complete Documentation can be found as pdf

http://pymzml.github.com/dist/pymzml.pdf

Citation

Please cite us when using pymzML in your work.

Bald, T., Barth, J., Niehues, A., Specht, M., Hippler, M., and Fufezan, C. (2012) pymzML - Python module for high throughput bioinformatics on mass spectrometry data, Bioinformatics, doi: 10.1093/bioinformatics/bts066

The original publication can be found here:

http://bioinformatics.oxfordjournals.org/content/early/2012/02/02/bioinformatics.bts066.abstract
http://bioinformatics.oxfordjournals.org/content/early/2012/02/02/bioinformatics.bts066.full.pdf

Installation

sudo python setup.py install

Introduction

Mass spectrometry has evolved into a very diverse field that relies heavily on high throughput bioinformatic tools. Due to the increasing complexity of the questions asked and biological problems addressed, standard tools might not be sufficient and tailored tools still have to be developed. However, the development of such tools has been hindered by proprietary data formats and the lack of an unified mass spectrometric data file standard. The latter has been overcome by the publication of the mzML standard by the HUPO Proteomics Standards Initiative (Deutsch, 2008) (http://www.psidev.info/) and soon all manufactures will hopefully offer a way to convert their format into this standardized one in order to stay comparable and competitive. Therefore in order to rapidly develop bioinformatic tools that can explore mass spectrometry data one needs a portable, robust, yet quick and easy interface to mzML files. The Python scripting language (http://python.org) is predestined for such a task.

Scripting languages carry several advantages compared to compiled programs and although compiled programs tend to be faster, scripting languages can already compete successfully in some tasks. For example, XML parsing is extremely optimized in Python due to the cElementTree module (http://effbot.org/zone/element-index.htm), which allows XML parsing in a fraction of classical C/C++ libraries, such as libxml2 or sgmlop. Therefore it seems natural that a well designed python mzML parser can successfully compete with C/C++ libraries currently available while offering the advantages of a scripting language.

CHAPTER

TWO

USAGE

This Chapter deals with some features of pymzml and explains the basic usage.

Basic usage

The Python scripting language is extended by pymzML to enable rapid evaluation, prototyping and even complex evaluation of large mass spectrometry datasets. The following examples are executed within the Python console (indicated by ">>>") but can equally be incorporated in standalone scripts.

The pymzML file object is declared as follows:

```
>>> import pymzml
>>> msrun = pymzml.run.Reader("big-1.0.0.mzML")
```

The input files can be plain or compressed mzML files. Initialization of the Run class accepts additional keywords, i.e. precision of MS1 or MSn runs and extra accessions (see: *obo*). The run object returns an iterator, hence one can loop over all spectra and/or chromatograms using classic Python syntax. Additionally, one can retrieve a specific spectrum by its nativeID via random access using msrun[1], msrun['TIC'] to access the chromatogram or in case of MRM experiments as e.g. msrun['transition_445-672']. Note that random access requires the mzML file not to be compressed or truncated by a conversion program.

```
>>> for spectrum in msrun:
>>>   if spectrum['ms level'] == 2:
```

```
>>> spectrum_with_nativeID_100 = msrun[100]
```

Each iteration returns a spectrum object (*spec.Spectrum*) which offers basic information on the spectrum. The information can be accessed like a Python dictionary. The keys in this dictionary are the accession numbers (e.g. MS:1000511) or the name of the accession (e.g. "ms level"), as they are defined by the HUPO Proteomics Standards Initiative in the mzML vocabulary, i.e. in the OBO files. Both keys of the spectrum dictionary point to the value that has been extracted from the mzML file (see: *obo*). Association of MS tag to name is done on the basis of the OBO files which are supplied in the pymzML package. The pymzML package offers a simple script (see: *queryOBO*) that can be used to translate between MS tags and names. Examples for accessing MS tags by their names can be found here: OBO parser Class. It is worth noting that the definition of MS tags and their attribute that the user wants to be associated with the tag and the trivial name is a feature. For example, the scan time is associated with two values, the actual time and its unit. From a programming point of view, access to the time as float by calling spectrum['scan time'] is desirable. Defining those attributes and their value one is interested in, facilitates live ;)

Data that is associated with the current spectrum, i.e. mass over charge (m/z) or intensity values can be accessed by the .mz (*spec.Spectrum.mz*) or .intensity (*spec.Spectrum.i*) properties, respectively, which are iterators themselves. The .peaks (*spec.Spectrum.peaks*) property also offers an iterator which returns mz and intensity as a tuple for each peak:

```
>>>     for mz, i in spectrum.peaks:
>>>       print(mz, i)
```

Since mass spectrometry data can be measured in profile mode, the .centroidedPeak (*spec.Spectrum.centroidedPeaks*) property offers an iterator which performs a simple Gauss fit on the profiled data, thereby returning the fitted m/z and maximum intensity of the bell shape curve as tuple. Basic visualization of spectra using XHTML and SVG can be done using the *plot* submodule.

Advanced usage

The spectrum class also offers other functions, such as deconvolution, estimation of similarity of spectra, simple noise reduction, addition of spectra and simple arithmetics on the intensity values by multiplication or division. An averaged spectrum can be created during parsing as follows:

```
>>> t = pymzml.spec.Spectrum()
>>> for s in msrun:
>>>   if s['ms level'] == 1:
>>>     t += s / s['total ion current']
```

The addition of spectra is done by creating Gaussian distributions around the centroided peak. This re-profiling is done internally as part of the addition and its values can be accessed by the .reprofiledPeaks (*spec.Spectrum.reprofiledPeaks*) property of the spectrum. To reduce the computational overhead one can remove noise by calling the .removeNoise() (*spec.Spectrum.removeNoise()*) method of the spectrum class. This method accepts different modes of noise reduction. Deconvoluted peaks can be accessed for high precision spectra by calling the .deconvolutedPeaks (*spec.Spectrum.deconvolutedPeaks*) property. The pymzML package contains detailed example scripts for all these methods. (Example scripts)

CHAPTER

THREE

MODULE RUN

The class *Reader* has been designed to selectivly extract data from a mzML file and to expose the data as a python object. Necessary information are read in and stored in a fast accesible format. The reader itself is an iterator, thus looping over all spectra follows the classical pythonian syntax. Additionally one can random access spectra by their nativeID if the file if not compressed or truncated by a conversion Program.

The class *Writer* is still in development.

class run.**Reader**

__init__(*path**[, *noiseThreshold = 0.0*, *extraAccessions = None*, *MS1_Precision = 5e-6*, *MSn_Precision = 20e-6*])

Initializes an mzML run and returns an iterator.

Parameters

- **path** (*string*) – path to mzML file. File can be gzipped.
- **extraAccessions** (*list of tuples*) – list of additional (accession,fieldName) tuples.

  For example, ('MS:1000285',['value']) will extract the "total ion current" and store it under two keys in the spectrum, i.e. spectrum["total ion current"] or spectrum['MS:1000285'].

  The translated name is extracted from the current OBO file, hence the name that is defined by the HUPO-PSI consortium is used. (http://www.psidev.info/).

  pymzML comes with an example script queryOBO.py which can be used to lookup the names or MS tags (see: *queryOBO*).

  The value, i.e. which xml property has to be extraced has to be provided by the user. Multiple values can be used as input, i.e. ('MS:1000016' , ['value','unitName']) will extract scan time and its unit.
- **MS1_Precision** (*float*) – measured precision of MS1 spectra
- **MSn_Precision** (*float*) – measured precision of MSn spectra

Example:

```
>>> run = pymzml.run.Reader("../mzML_example_files/100729_t300_100729172744.mzML.gz" ,
                            MS1_Precision = 20e-6 )
```

next()

Iterator in class Run:

will return an instance of *spec.Spectrum*, stored in run.spectrum.

Example:

```
        >>> for spectrum in run:
        ...     print(spectrum['id'], end='\r')
```

class run.**Writer**

__init__ (*filename**, *run**[, *overwrite = boolean*])

Initializes an mzML writer (beta stage).

Parameters

- **path** (*string*) – filename for the new mzML file.
- **run** (*pymzml.run.Reader*) – Currently a pymzml.run.Reader object is required since we do not write the header by ourselves, yet.
- **overwrite** (*boolean*) – force the re-initialization of mzML file, even if file exists.

Example:

```
>>> run = pymzml.run.Reader('../mzML_example_files/100729_t300_100729172744.mzML', MS1_Precision
>>> run2 = pymzml.run.Writer(filename = 'write_test.mzML', run= run , overwrite = True)
>>> spec = run[1000]
>>> run2.addSpec(spec)
>>> run2.save()
```

CHAPTER

FOUR

CLASS SPECTRUM

Spectrum class offers a python object for mass spectrometry data. The spectrum object holds the basic information on the spectrum and offers methods to interrogate properties of the spectrum. Data, i.e. mass over charge (m/z) and intensity decoding is performed on demand and can be accessed via their properties, e.g. *spec.Spectrum.peaks*.

The Spectrum class is used in the `run.Run` class. There each spectrum is accessible as a Spectrum object.

Theoretical spectra can also be created using the setter functions. For example, m/z values, intensities, and peaks can be set by the corresponding properties: *spec.Spectrum.mz*, *spec.Spectrum.i*, *spec.Spectrum.peaks*.

class `spec.`**Spectrum**

__init__ (*measuredPrecision = value**)
Initializes a pymzml.spec.Spectrum class.

Parameters **measuredPrecision** (*float*) – in m/z, mandatory

xmlTree
xmlTree property returns an iterator over the original xmlTree structure the spectrum was initilized with.

Example:

```
>>> for element in spectrum.xmlTree:
...   print( element, element.tag, element.items() )
```

please refer to the xml documentation of Python and cElementTree for more details.

mz
Returns the list of m/z values. If the m/z values are encoded, the function `_decode()` is used to decode the encoded data.

The mz property can also be setted, e.g. for theoretical data. However, it is recommended to use the peaks property to set mz and intesity tuples at same time.

Return type list

Returns Returns a list of mz from the actual analysed spectrum

i
Returns the list of the intensity values. If the intensity values are encoded, the function `_decode()` is used to decode the encoded data.

The i property can also be setted, e.g. for theoretical data.However, it is recommended to use the peaks property to set mz and intesity tuples at same time.

Return type list

Returns Returns a list of intensity values from the actual analysed spectrum.

peaks

Returns the list of peaks of the spectrum as tuples (m/z, intensity).

Return type list of tuples

Returns Returns list of tuples (m/z, intensity)

Example:

```
>>> import pymzml
>>> run = pymzml.run.Reader(spectra.mzMl.gz, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
...     for mz, i in spectrum.peaks:
...         print(mz, i)
```

Note: The peaks property can also be setted, e.g. for theoretical data. It requires a list of mz/intensity tuples.

centroidedPeaks

Returns the centroided version of a profile spectrum. Performs a Gauss fit to determine centroided mz and intensities, if the spectrum is in measured profile mode. Returns a list of tuples of fitted m/z-intesity values. If the spectrum peaks are already centroided, these peaks are returned.

Return type list of tuples

Returns Returns list of tuples (m/z, intensity)

Example:

```
>>> import pymzml
>>> run = pymzml.run.Reader(spectra.mzMl.gz, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
...     for mz, i in spectrum.centroidedPeaks:
...         print(mz, i)
```

reprofiledPeaks

Returns the reprofiled version of a centroided spectrum.

Return type list of reprofiled mz,i tuples

Returns Reprofiled peaks as tuple list

Example:

```
>>> import pymzml
>>> run = pymzml.run.Reader(spectra.mzMl.gz, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
...     for mz, i in spectrum.reprofiledPeaks:
...         print(mz, i)
```

reprofiledPeaks

Returns the reprofiled version of a centroided spectrum.

Return type list of reprofiled mz,i tuples

Returns Reprofiled peaks as tuple list

Example:

```
>>> import pymzml
>>> run = pymzml.run.Reader(spectra.mzMl.gz, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
```

```
...     for mz, i in spectrum.reprofiledPeaks:
...         print(mz, i)
```

measuredPrecision
Sets the measured and internal precision

Parameters **value** (*float*) – measured precision (e.g. 5e-6)

__add__ (*otherSpec*)
Adds two pymzml spectra together.

Parameters **otherSpec** (*object*) – Spectrum object

Example:

```
>>> import pymzml
>>> s = pymzml.spec.Spectrum( measuredPrescision = 20e-6 )
>>> file_to_read = "../mzML_example_files/xy.mzML.gz"
>>> run = pymzml.run.Reader(file_to_read , MS1_Precision = 5e-6 , MSn_Precision = 20e-6)
>>> for spec in run:
...     s += spec
```

__mul__ (*value*)
Multiplies each intensity with a float, i.e. scales the spectrum.

Parameters **value** (*float*) – Value to multiply the spectrum

__truediv__ (*value*)
Divides each intensity by a float, i.e. scales the spectrum.

Parameters **value** (*float, int*) – Value to divide the spectrum

strip (*scope='all'*)
Reduces the size of the spectrum. Interesting if specs need to be added or stored.

Parameters **scope** (*string*) – accepts currently ["all"]

"all" will remove the raw and profiled data and some internal lookup tables as well.

extremeValues (*key*)
Find extreme values, minimal and maximum mz and intensity

Parameters **key** (*string*) – m/z : "mz" or intensity : "i"

Return type tuple

Returns tuple of minimal and maximum m/z or intensity

reduce (*mzRange=(None, None)*)
Works on peaks and reduces spectrum to a m/z range.

Example:

```
>>> run = pymzml.run.Reader(file_to_read, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spec in run:
...     spec.reduce( mzRange = (100,200) )
```

deRef ()
Strip some heavy data and return deepcopy of spectrum.

Example:

```
>>> run = pymzml.run.Reader(file_to_read, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spec in run:
...     tmp = spec.deRef()
```

removeNoise(*mode='median'*, *noiseLevel=None*)

Function to remove noise from peaks, centroided peaks and reprofiled peaks.

Parameters **mode** (*string*) – define mode for removing noise. Default = "median" (other modes: "mean", "mad")

Return type list of tuples

Returns Returns a list with tuples of m/z-intensity pairs above the noise threshold

mad < median < mean

Threshold is calculated over the mad/median/mean of all intensity values. (mad = mean absolute deviation)

Example:

```
>>> import pymzml
>>> run = pymzml.run.Reader(spectra.mzML.gz, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
...     for mz, i in spectrum.removeNoise( mode = 'mean'):
...         print(mz, i)
```

highestPeaks(*n*)

Function to retrieve the n-highest centroided peaks of the spectrum.

Parameters **n** (*int*) – Number of n-highest peaks

Return type list

Returns list of centroided peaks (mz, intensity tuples)

Example:

```
>>> run = pymzml.run.Reader("../mzML_example_files/deconvolution.mzML.gz", MS1_Precision = 5
>>> for spectrum in run:
...     if spectrum["ms level"] == 2:
...         if spectrum["id"] == 1770:
...             for mz,i in spectrum.highestPeaks(5):
...                 print(mz,i)
```

estimatedNoiseLevel(*mode='median'*)

Calculates noise threshold for function *removeNoise()*

hasOverlappingPeak(*mz*)

Checks if a spetrum has more than one peak for a given m/z value and within the measured precision

Parameters **mz** (*float*) – m/z value which should be checked

Returns Returns `True` if a nearby peak is detected, otherwise `False`

Return type bool

hasPeak(*mz2find*)

Checks if a Spectrum has a certain peak. Needs a certain mz value as input and returns a list of peaks if a peak is found in the spectrum, otherwise `[]` is returned. Every peak is a tuple of m/z and intensity.

Parameters **mz2find** (*float*) – mz value which should be found

Return type list

Returns m/z and intensity as tuple in list

Example:

```
>>> import pymzml, get_example_file
>>> example_file = get_example_file.open_example('deconvolution.mzML.gz')
>>> run = pymzml.run.Reader(example_file, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
...     if spectrum["ms level"] == 2:
...             peak_to_find = spectrum.hasPeak(1016.5404)
...             print(peak_to_find)
[(1016.5404, 19141.735187697403)]
```

hasDeconvolutedPeak (*mass2find*)

Checks if a deconvoluted spectrum contains a certain peak. Needs a mass value as input and returns a list of peaks if a peak is found in the spectrum. If the mass is not found `[]` is returned. Every peak is a tuple of m/z and intensity.

Parameters **mass2find** (*float*) – mass value which should be found

Return type list

Returns mass and intensity as tuple in list if mass is found, otherwise `[]`

Example:

```
>>> import pymzml, get_example_file
>>> example_file = get_example_file.open_example('deconvolution.mzML.gz')
>>> run = pymzml.run.Reader(example_file, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
...     if spectrum["ms level"] == 2:
...             peak_to_find = spectrum.hasDeconvolutedPeak(1044.5804)
...             print(peak_to_find)
[(1044.5596, 3809.4356300564586)]
```

similarityTo (*spec2*)

Compares two spectra and returns cosine

Parameters **spec2** (*pymzml.spec.Spectrum*) – another pymzml spectrum that is compated to the current spectrum.

Returns value between 0 and 1, i.e. the cosine between the two spectra.

Return type float

Note: Spectra data is transformed into an n-dimensional vector, whereas mz values are binned in bins of 10 m/z and the intensities are added up. Then the cosine is calculated between those two vectors. The more similar the specs are, the closer the value is to 1.

---

tmzSet

Creates a set out of transformed m/z values (including all values in the defined imprecision).

Return type set

tmassSet

Creates a set out of transformed mass values (including all values in the defined imprecision).

Return type set

transformedPeaks

m/z value is multiplied by the internal precision

Return type list of tuples

Returns Returns a list of peaks (tuples of mz and intensity). Float m/z values are adjusted by the internal precision to integers.

transformed_deconvolutedPeaks
Deconvoluted mz value is multiplied by the internal precision

Return type list of tuples

Returns Returns a list of peaks (tuples of mz and intensity). Float m/z values are adjusted by the internal precision to integers.

deconvolute_peaks(*ppmFactor=4*, *minCharge=1*, *maxCharge=8*, *maxNextPeaks=100*)
Calculating uncharged masses and returning deconvoluted peaks.

The deconvolution of spectra is done by first identifying isotope envelopes and the charge state of this envelopes. The first peak of an isotope envelope is choosen as the monoisotopic peak for which the mass is calculated from the m/z ratio. Isotope envelopes are identified by searching the centroided spectrum for peaks which show no preceding isotope peak within a specified mass accuracy. To be sure, the measured mass accuracy is multiplied by a user adjustable factor (`ppmFactor`). When the current peak meets the criteria with no preceding peaks, the following peaks are analysed. The following peaks are considered to be part of the isotope envelope, as long as they fit within the measured precision and only one local maximum is present. The second local maximum is not considered as the starting point of a new isotope envelope as one cannot be sure were this isotope envelope starts. However, the last peak before the second local maximum is considered to be part of the isotope envelope from the first local maximum, as the intensity of this peak shouldn't have a big influence on the whole isotope envelope intensity. The charge range for detecting isotope envelopes can be specified (`minCharge`, `maxCharge`). An isotope envelope always gets the highest possible charge. With the charge the mass can be calculated from the m/z value of the first peak of the isotope envelope. The intensity of the deconvoluted peak results from the sum of all isotope envelope peaks. In a last step, deconvoluted peaks are grouped together within the measured precision. This is necessary because isotope envelopes from the same fragment but with different charge states can leed to slightly different deconvoluted peaks.

Parameters

- **ppmFactor** (*int*) – ppm factor (imprecision factor)
- **minCharge** (*int*) – minimum charge considered
- **maxCharge** (*int*) – maximum charge considered
- **maxNextPeaks** – maximum length for isotope envelope

Return type tuple (mass, intensity)

Returns Deconvoluted peaks, mass (instead of m/z) and intensity are returned

deconvolutedPeaks
Calling *spec.Spectrum.deconvolute_peaks()* with standard parameters, which calculates uncharged masses and returns deconvoluted peaks.

Return type list

Returns list of deconvoluted peaks (mass (instead of m/z) / intensity tuples)

CHAPTER

FIVE

OBO PARSER CLASS

Class to parse the obo file and set up the accessions library

The OBO parse has been designed to convert MS:xxxxx tags to their appropriate names. A minimal set of MS acession is used in pymzml, but additional accessions can easily added, using the extraAccession parameter during *run.Reader* initialization.

The obo translator is used internally to associate names with MS:xxxxxxx tags.

The oboTranslator Class generates a dictionary and several lookup tables. e.g.

```
>>> from pymzml.obo import oboTranslator as OT
>>> translator = OT()
>>> len(translator.id.keys()) # Numer of parsed entries
737
>>> translator['MS:1000127']
'centroid mass spectrum'
>>> translator['positive scan']
{'is_a': 'MS:1000465 ! scan polarity', 'id': 'MS:1000130', 'def': '"Polarity
of the scan is positive." [PSI:MS]', 'name': 'positive scan'}
>>> translator['scan']
{'relationship': 'part_of MS:0000000 ! Proteomics Standards Initiative Mass
Spectrometry Ontology', 'id': 'MS:1000441', 'def': '"Function or process of
the mass spectrometer where it records a spectrum." [PSI:MS]', 'name':
'scan'}
>>> translator['unit']
{'relationship': 'part_of MS:0000000 ! Proteomics Standards Initiative Mass
Spectrometry Ontology', 'id': 'MS:1000460', 'def': '"Terms to describe
units." [PSI:MS]', 'name': 'unit'}
```

pymzML comes with the queryOBO.py script that can be used to interogate the OBO file.

```
$ ./example_scripts/queryOBO.py "scan time"
MS:1000016
scan time
"The time taken for an acquisition by scanning analyzers." [PSI:MS]
Is a: MS:1000503 ! scan attribute
$
```

Accessing specific OBO MS tags

This section describes how to access some common MS tags by their names as they are defined in the OBO file.

First pymzML is imported and the run is defined.

```
>>> example_file = get_example_file.open_example('dta_example.mzML')
>>> import pymzml
>>> msrun = pymzml.run.Reader(example_file)
```

Now, we can fetch specific imformations from the spectrum object.

MS level:

```
>>> for spectrum in msrun:
...     print(spectrum['ms level'])
```

Total Ion current:

```
>>> for spectrum in msrun:
...     print(spectrum['total ion current'])
```

Furthermore we can also check for presence of parameters, therefore the proprties of the spectrum.

Differentiation of e.g. HCD and CID fractionation:

```
>>> for spectrum in msrun:
...     if spctrum['ms level'] == 2:
...         if 'collision-induced dissociation' in spectrum.keys():
...             print('Spectrum {0} is a CID spectrum'.format(spectrum['id']))
...         elif 'high-energy collision-induced dissociation' in spectrum.keys():
...             print('Spectrum {0} is a HCD spectrum'.format(spectrum['id']))
```

Minimal accession set

The following dictionary shows the minimal accession necessary to run pymzML.

```
MIN_REQ = [
#
#!NOTE!   exact names will be extracted of current OBO File, comments are just an orientation
#         pymzml comes with a little script (queryOBO.py) to query the obo file
#
#         $ ./example_scripts/queryOBO.py "scan time"
#         MS:1000016
#         scan time
#         "The time taken for an acquisition by scanning analyzers." [PSI:MS]
#         Is a: MS:1000503 ! scan attribute
#
('MS:1000016',['value']             ), #"scan time"
# -> Could also be ['value','unitName'] to retrieve a
# tuple of time and unit by calling spectrum['scan time']
('MS:1000040',['value']             ), #"m/z"
('MS:1000041',['value']             ), #"charge state"
('MS:1000127',['name']              ), #"centroid spectrum"
('MS:1000128',['name']              ), #"profile spectrum"
('MS:1000133',['name']              ), #"collision-induced dissociation"
('MS:1000285',['value']             ), #"total ion current"
('MS:1000422',['name']              ), #"high-energy collision-induced dissociation"
('MS:1000511',['value']             ), #"ms level"
('MS:1000512',['value']             ), #"filter string"
('MS:1000514',['name']              ), #"m/z array"
('MS:1000515',['name']              ), #"intensity array"
('MS:1000521',['name']              ), #"32-bit float"
```

```
('MS:1000523',['name']                ), #"64-bit float"
('MS:1000744',['value']               ), # legacy precursor mz value ...
]
```

CHAPTER

SIX

PLOTTING FUNCTIONS

Plotting functions for pymzML

class plot.**Factory**(*filename=None*)

Class to plot pymzml.spec.Spectrum as svg/xhtml.

Parameters **filename** (*string*) – Name for the output file. Default = "spectra.xhtml"

Example:

```
>>> import pymzml, get_example_file
>>> mzMLFile = 'profile-mass-spectrum.mzml'
>>> example_file = get_example_file.open_example(mzMLFile)
>>> run = pymzml.run.Run("../mzML_example_files/"+mzMLFile, precisionMSn = 250e-6)
>>> p = pymzml.plot.Factory()
>>> for spec in run:
>>>     p.newPlot()
>>>     p.add(spec.peaks, color=(200,00,00), style='circles')
>>>     p.add(spec.centroidedPeaks, color=(00,00,00), style='sticks')
>>>     p.add(spec.reprofiledPeaks, color=(00,255,00), style='circles')
>>>     p.save( filename="output/plotAspect.xhtml" , mzRange = [745.2,745.6] )
```

add(*data, color=(0, 0, 0), style='sticks', mzRange=[None, None]*)

Add data to the graph.

Parameters

- **data** (*list of tuples (mz,i)*) – The data added to the graph
- **color** (*tuple (R,G,B)*) – color encoded in RGB. Default = (0,0,0)
- **style** (*string*) – plotting style. Default = "circles".
- **mzRange** (*tuple of minMZ,maxMZ*) – Boundaries that should be added to the current plot

Currently supported styles are:

- 'emptycircles'
- 'circles'
- 'sticks'
- 'squares'
- 'area'
- 'triangle'
- 'label'

NOTE: The data format for label style is [(mz1, 'label1'), (mz2, 'label2'), (mz3, 'label3')].

info()
Returns summary about the plotting factory, i.e.how many plots and how many datasets per plot.

newPlot(*header=None, mzRange=[None, None], normalize=False*)
Add new plot to the plotFactory.

Parameters

- **header** (*string*) – an optional title that will be printed above the plot
- **mzRange** (*tuple of minMZ,maxMZ*) – Boundaries of the new plot
- **normalize** – whether or not the individal data sets are normalized in the plot

save(*filename=None, mzRange=[None, None]*)
Saves all plots and their data points that have been added to the plotFactory.

Parameters

- **filename** (*string*) – Name for the output file. Default = "spectra.xhtml"
- **mzRange** (*list*) – m/z range which should be considered [start, end]. Default = [None , None]

CHAPTER

SEVEN

EXAMPLE SCRIPTS

In this section, example scripts for the usage of pymzML can be found.

Finding a peak

hasPeak.py Testscript to demonstrate functionality of function *spec.Spectrum.hasPeak()* or *spec.Spectrum.hasDeconvolutedPeak()*

Example:

```
>>> import pymzml, get_example_file
>>> example_file = get_example_file.open_example('deconvolution.mzml.gz')
>>> run = pymzml.run.run(example_file, precisionms1 = 5e-6, precisionmsn = 20e-6)
>>> for spectrum in run:
...     if spectrum["ms level"] == 2:
...             peak_to_find = spectrum.haspeak(1016.5404)
...             print(peak_to_find)
(1016.5404, 19141.735187697403)
```

Plotting a spectrum

plotAspec.py This function shows how to plot a simple spectrum. It can be directly plotted via this script or using the python console

Example of plotting a spectrum:

```
>>> import pymzml, get_example_file
>>> mzMLFile = 'profile-mass-spectrum.mzml'
>>> example_file = get_example_file.open_example(mzMLFile)
>>> run = pymzml.run.Reader("../mzML_example_files/"+mzMLFile, MSn_Precision = 250e-6)
>>> p = pymzml.plot.Factory()
>>> for spec in run:
>>>     p.newPlot()
>>>     p.add(spec.peaks, color=(200,00,00), style='circles')
>>>     p.add(spec.centroidedPeaks, color=(00,00,00), style='sticks')
>>>     p.add(spec.reprofiledPeaks, color=(00,255,00), style='circles')
>>>     p.save( filename="output/plotAspect.xhtml" , mzRange = [744.7,747] )
```

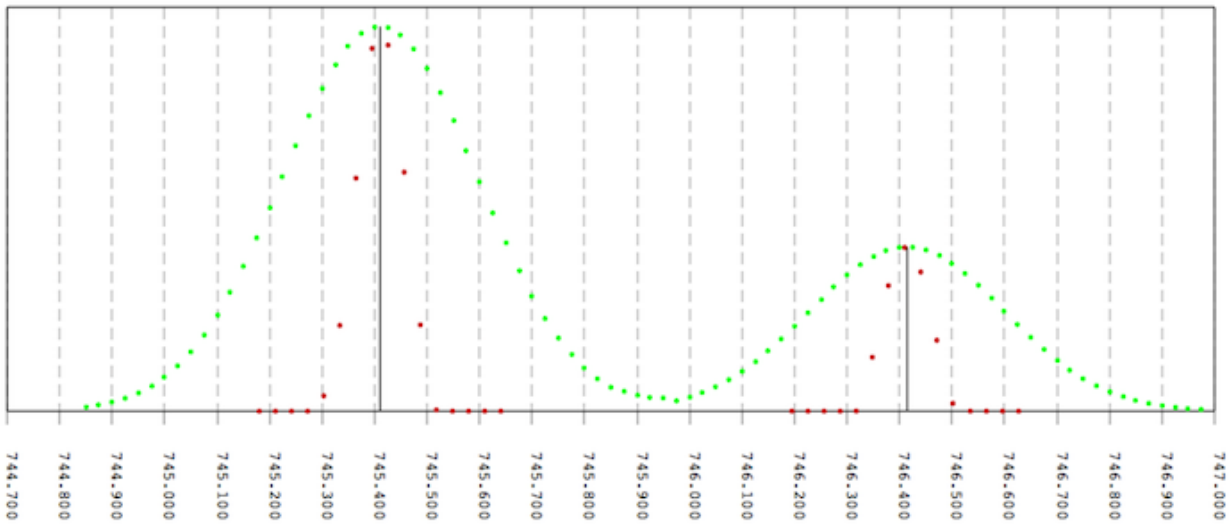

Fig. 7.1: output as xhtml :)

Abundant precursor

find_abundant_precursors.py Testscript to demonstrate an easy isoloation of the most abundant, isolated precusors for MSn spectra.Thus exclusion list for further MS runs can be generated. In this example all precursors which were isolated more than 5 times are found and printed.

Example:

```
>>> import pymzml, get_example_file, operator.item
>>> import collections import defaultdict as ddict
>>> from operator import itemgetter
>>> example_file = get_example_file.open_example('dta_example.mzML')
>>> run = pymzml.run.Reader(example_file , MS1_Precision = 5e-6 , MSn_Precision = 20e-6 )
>>> precursor_count_dict = ddict(int)
>>> for spectrum in run:
>>>    if spectrum["ms level"] == 2:
>>>        if "precursors" in spectrum.keys():
>>>            precursor_count_dict[round(float(spectrum["precursors"][0]["mz"]),3)] += 1
>>> for precursor, frequency in sorted(precursor_count_dict.items()):
>>>     print("{0}\t{1}".format(precursor, frequency))
```

Compare Spectra

compareSpectra.py Compare two spectra and return the cosine distance between them. The returned value is between 0 and 1, a returned value of 1 represents highest similarity.

Example:

```
>>> spec1 = pymzml.spec.Spectrum(measuredPrecision = 20e-5)
>>> spec2 = pymzml.spec.Spectrum(measuredPrecision = 20e-5)
>>> spec1.peaks = [ ( 1500,1 ), ( 1502,2.0 ), (1300,1 )]
>>> spec2.peaks = [ ( 1500,1 ), ( 1502,1.3 ), (1400,2 )]
>>> spec1.similarityTo( spec2 )
0.5682164685724541
```

```
>>> spec1.similarityTo( spec1 )
1.0000000000000002
```

Query Obo files

queryOBO.py pymzML comes with the queryOBO.py script that can be used to interogate the OBO file.

Usage:

```
$ ./example_scripts/queryOBO.py "scan time"
MS:1000016
scan time
"The time taken for an acquisition by scanning analyzers." [PSI:MS]
Is a: MS:1000503 ! scan attribute
$
```

Highest peaks

highestPeaks.py Testscript to isolate the n-highest peaks from an example file.

Example:

```
>>> example_file = get_example_file.open_example('deconvolution.mzML.gz')
>>> run = pymzml.run.Reader(example_file, MS1_Precision = 5e-6, MSn_Precision = 20e-6)
>>> for spectrum in run:
...     if spectrum["ms level"] == 2:
...         if spectrum["id"] == 1770:
...             for mz,i  in spectrum.highestPeaks(5):
...                 print(mz,i)
```

extract a specific Ion Chromatogram (EIC, XIC)

extractIonChromatogram.py Demonstration of the extraction of a specific ion chromatogram, i.e. XIC or EIC

Example:

```
>>> import pymzml, get_example_file
>>> example_file = get_example_file.open_example('small.pwiz.1.1.mzML')
>>> run = pymzml.run.Reader(example_file, MS1_Precision = 20e-6, MSn_Precision = 20e-6)
>>> timeDependentIntensities = []
>>> for spectrum in run:
...     if spectrum['ms level'] == 1:
...         matchList = spectrum.hasPeak(MASS_2_FOLLOW)
...         if matchList != []:
...             for mz,I in matchList:
...                 timeDependentIntensities.append( [ spectrum['scan time'], I , mz ])
>>> for rt, i, mz in timeDependentIntensities:
...     print('{0:5.3f} {1:13.4f}       {2:10}'.format( rt, i, mz ))
```

Accessing the original XML Tree of a spectrum

accessAllData.py accessAllData.py

Demos the usage of the spectrum.xmlTree iterator that can be used to extract all MS:tag for a given spectrum.

Example:

```
>>> example_file = get_example_file.open_example('small.pwiz.1.1.mzML')
>>> run = pymzml.run.Reader(example_file, MSn_Precision = 250e-6)
>>> spectrum = run[1]
>>> for element in spectrum.xmlTree:
...     print('-'*40)
...     print(element)
...     print(element.get('accession') )
...     print(element.tag)
...     print(element.items())
```

Write mzML

writeExample.py

This is part is still in development. Simple test of the mzML writing functionality.

This is very priliminary. The 'header' is copied into the new file with some addition in the softwareList XML tag, hence a pymzml.run.Reader Object needs to be passed over to the write function.

Example:

```
>>> example_file = get_example_file.open_example('small.pwiz.1.1.mzML')
>>> run  = pymzml.run.Reader(example_file, MS1_Precision = 5e-6)
>>> run2 = pymzml.run.Writer(filename = 'write_test.mzML', run= run , overwrite = True)
>>> specOfIntrest = run[2]
>>> run2.addSpec(spec)
>>> run2.save()
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

Symbols

A

C

D

E

F

H

I

M

N

O

P

Q

R

S

T

W

X