pyOpenMS Documentation

Release 2.3.0

OpenMS Team

Sep 25, 2018

Installation

pyOpenMS are the Python bindings to the OpenMS library which are available for Windows, Linux and OSX.

The pyOpenMS package contains Python bindings for a large part of the OpenMS library (http://www.open-ms.de) for mass spectrometry based proteomics. It thus provides providing facile access to a feature-rich, open-source algorithm library for mass-spectrometry based proteomics analysis. These Python bindings allow raw access to the data-structures and algorithms implemented in OpenMS, specifically those for file access (mzXML, mzML, TraML, mzIdentML among others), basic signal processing (smoothing, filtering, de-isotoping and peak-picking) and complex data analysis (including label-free, SILAC, iTRAQ and SWATH analysis tools).

Please see the appendix of the official pyOpenMS Manual for a comeplete documentation of the pyOpenMS API and all wrapped classes.

CHAPTER 1

pyOpenMS Installation

1.1 Binaries

To install pyOpenMS from the binaries, you can simply type

```
pip install numpy
pip install pyopenms
```

We have binary packages for OSX, Linux and Windows (64 bit only). Note that for Windows, we only support Python 3.5 and 3.6 in their 64 bit versions, therefore make sure to download the 64bit Python release. For OSX and Linux, we also support Python 2.7 as well as Python 3.4 (Linux only).

You can install Python first from here, again make sure to download the 64bit release. You can then open a shell and type the two commands above (on Windows you may potentially have to use `C:\Python36\Scripts\pip.exe` in case `pip` is not in your system path).

1.2 Source

To install pyOpenMS from source, you will first have to compile OpenMS successfully on your platform of choice and then follow the building from source instructions.

1.3 Wrap Classes

In order to wrap new classes in pyOpenMS, read the following guide.

CHAPTER 2

Getting Started

2.1 Import pyopenms

After installation, you should be able to import pyopenms as a package

```
import pyopenms
```

which should now give you access to all of pyopenms. You should now be able to interact with the OpenMS library and, for example, read and write mzML files:

```
import pyopenms
exp = pyopenms.MSExperiment()
pyopenms.MzMLFile().store("testfile.mzML", exp)
```

which will create an empty mzML file called *testfile.mzML*.

2.2 Getting help

There are multiple ways to get information about the available functions and methods. We can inspect individual pyOpenMS objects through the `help` function:

```
>>> import pyopenms
>>> help(pyopenms.MSExperiment)
Help on class MSExperiment in module pyopenms.pyopenms_2:

class MSExperiment(__builtin__.object)
 |  Methods defined here:
 |
 |  __copy__(...)
 |
 |  __deepcopy__(...)
 [...]
```

which lists the available functions. This full list indicates that `pyopenms.MSExperiment` has multiple methods, among them `__copy__`. The command also lists the signature for each function, allowing users to identify the function arguments and return types. In order to get more information about the wrapped functions, we can also consult the pyOpenMS manual which references to all wrapped functions. For a more complete documentation of the underlying wrapped methods, please consult the official OpenMS documentation, in this case the MSExperiment documentation.

2.3 First look at data

2.3.1 File reading

pyOpenMS supports a variety of different files through the implementations in OpenMS. In order to read mass spectrometric data, we can download the mzML example file

```
import urllib
from pyopenms import *
urllib.urlretrieve ("http://proteowizard.sourceforge.net/example_data/tiny.pwiz.1.1.
↪mzML", "tiny.pwiz.1.1.mzML")
exp = MSExperiment()
MzMLFile().load("tiny.pwiz.1.1.mzML", exp)
```

which will load the content of the "tiny.pwiz.1.1.mzML" file into the `exp` variable of type `MSExperiment`. We can now inspect the properties of this object:

```
>>> help(exp)
Help on MSExperiment object:

class MSExperiment(__builtin__.object)
 |  Methods defined here:
 ...
 |  getNrChromatograms(...)
 |      Cython signature: size_t getNrChromatograms()
 |
 |  getNrSpectra(...)
 |      Cython signature: size_t getNrSpectra()
 |
 ...
```

which indicates that the variable `exp` has (among others) the functions `getNrSpectra` and `getNrChromatograms`. We can now try these functions:

```
>>> exp.getNrSpectra()
4
>>> exp.getNrChromatograms()
2
```

and indeed we see that we get information about the underlying MS data. We can iterate through the spectra as follows:

2.3.2 Iteration

```
>>> for spec in exp:
...   print "MS Level:", spec.getMSLevel()
...
```

(continues on next page)

(continued from previous page)

```
MS Level: 1
MS Level: 2
MS Level: 1
MS Level: 1
```

This iterates through all available spectra, we can also access spectra through the `[]` operator:

```
>>> print "MS Level:", exp[1].getMSLevel()
MS Level: 2
```

Note that `spec[1]` will access the *second* spectrum (arrays start at `0`). We can access the raw peaks through `get_peaks()`:

```
>>> spec = exp[1]
>>> mz, i = spec.get_peaks()
>>> sum(i)
110
```

Which will access the data using a numpy array, storing the *m/z* information in the `mz` vector and the intensity in the `i` vector. Alternatively, we can also iterate over individual peak objects as follows (this tends to be slower):

```
>>> for peak in spec:
...   print peak.getIntensity()
...
20.0
18.0
16.0
14.0
12.0
10.0
8.0
6.0
4.0
2.0
```

2.3.3 TIC calculation

With this information, we can now calculate a total ion current (TIC) using the following function:

```
def calcTIC(exp):
    tic = 0
    for spec in exp:
        if spec.getMSLevel() == 1:
            mz, i = spec.get_peaks()
            tic += sum(i)
    return tic
```

To calculate a TIC we would now call the function:

```
>>> calcTIC(exp)
240.0
>>> sum([sum(s.get_peaks()[1]) for s in exp if s.getMSLevel() == 1])
240.0
```

Note how one can compute the same property using list comprehensions in Python (see the third line above).

CHAPTER 3

Reading Raw MS data

3.1 mzML files in memory

As discussed in the last section, the most straight forward way to load mass spectrometric data is using the `MzMLFile` class:

```
import urllib
from pyopenms import *
urllib.urlretrieve ("http://proteowizard.sourceforge.net/example_data/tiny.pwiz.1.1.
↪mzML", "test.mzML")
exp = MSExperiment()
MzMLFile().load("test.mzML", exp)
```

which will load the content of the "test.mzML" file into the `exp` variable of type `MSExperiment`. We can access the raw data and spectra through:

```
spectrum_data = exp.getSpectrum(0).get_peaks()
chromatogram_data = exp.getChromatogram(0).get_peaks()
```

Which will allow us to compute on spectra and chromatogram data. We can manipulate the spectra in the file for example as follows:

```
spec = []
for s in exp.getSpectra():
    if s.getMSLevel() != 1:
        spec.append(s)
exp.setSpectra(spec)
```

Which will only keep MS2 spectra in the `MSExperiment`. We can then store the modified data structure on disk:

```
MzMLFile().store("filtered.mzML", exp)
```

Putting this together, a small filtering program would look like this:

```
"""
Script to read mzML data and filter out all MS1 spectra
"""
from pyopenms import *
exp = MSExperiment()
MzMLFile().load("test.mzML", exp)

spec = []
for s in exp.getSpectra():
    if s.getMSLevel() != 1:
        spec.append(s)

exp.setSpectra(spec)

MzMLFile().store("filtered.mzML", exp)
```

3.2 mzML files as streams

In some instances it is impossible or inconvenient to load all data from an mzML file directly into memory. OpenMS offers streaming-based access to mass spectrometric data which uses a callback object that receives spectra and chromatograms as they are read from the disk. A simple implementation could look like

```
class MSCallback():
    def setExperimentalSettings(self, s):
        pass

    def setExpectedSize(self, a, b):
        pass

    def consumeChromatogram(self, c):
        print "Read a chromatogram"

    def consumeSpectrum(self, s):
        print "Read a spectrum"
```

which can the be used as follows:

```
>>> from pyopenms import *
>>> filename = "test.mzML"
>>> consumer = MSCallback()
>>> MzMLFile().transform(filename, consumer)
Read a spectrum
Read a spectrum
Read a spectrum
Read a spectrum
Read a chromatogram
Read a chromatogram
```

which provides an intuition on how the callback object works: whenever a spectrum or chromatogram is read from disk, the function `consumeSpectrum` or `consumeChromatogram` is called and a specific action is performed. We can use this to implement a simple filtering function for mass spectra:

```
class FilteringConsumer():
    """
```

(continues on next page)

(continued from previous page)

```
    Consumer that forwards all calls the internal consumer (after
    filtering)
    """

    def __init__(self, consumer, filter_string):
        self._internal_consumer = consumer
        self.filter_string = filter_string

    def setExperimentalSettings(self, s):
        self._internal_consumer.setExperimentalSettings(s)

    def setExpectedSize(self, a, b):
        self._internal_consumer.setExpectedSize(a, b)

    def consumeChromatogram(self, c):
        if c.getNativeID().find(self.filter_string) != -1:
            self._internal_consumer.consumeChromatogram(c)

    def consumeSpectrum(self, s):
        if s.getNativeID().find(self.filter_string) != -1:
            self._internal_consumer.consumeSpectrum(s)

###################################
filter_string = "DECOY"
inputfile = "in.mzML"
outputfile = "out.mzML"
###################################

consumer = pyopenms.PlainMSDataWritingConsumer(outputfile)
consumer = FilteringConsumer(consumer, filter_string)

pyopenms.MzMLFile().transform(inputfile, consumer)
```

where the spectra and chromatograms are filtered by their native ids. It is similarly trivial to implement filtering by other attributes. Note how the data are written to disk using the `PlainMSDataWritingConsumer` which is one of multiple available consumer classes – this specific class will simply take the spectrum `s` or chromatogram `c` and write it to disk (the location of the output file is given by the `outfile` variable).

This approach is memory efficient in cases where the whole data data may not fit into memory.

CHAPTER 4

Other MS data formats

4.1 Identification data (idXML, mzIdentML, pepXML, protXML)

You can store and load identification data from an *idXML* file as follows:

```
import urllib
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urllib.urlretrieve (gh +"/src/tests/class_tests/openms/data/IdXMLFile_whole.idXML",
↪"test.idXML")
protein_ids = []
peptide_ids = []
IdXMLFile().load("test.idXML", protein_ids, peptide_ids)
IdXMLFile().store("test.out.idXML", protein_ids, peptide_ids)
```

You can store and load identification data from an *mzIdentML* file as follows:

```
import urllib
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urllib.urlretrieve (gh + "/src/tests/class_tests/openms/data/MzIdentML_3runs.mzid",
↪"test.mzid")
protein_ids = []
peptide_ids = []
MzIdentMLFile().load("test.mzid", protein_ids, peptide_ids)
MzIdentMLFile().store("test.out.mzid", protein_ids, peptide_ids)
```

You can store and load identification data from a TPP *pepXML* file as follows:

```
import urllib
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urllib.urlretrieve (gh + "/src/tests/class_tests/openms/data/PepXMLFile_test.pepxml",
↪"test.pepxml")
```

(continues on next page)

(continued from previous page)

```
protein_ids = []
peptide_ids = []
PepXMLFile().load("test.pepxml", protein_ids, peptide_ids)
PepXMLFile().store("test.out.pepxml", protein_ids, peptide_ids)
```

You can load (storing is not supported) identification data from a TPP *protXML* file as follows:

```
import urllib
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urllib.urlretrieve (gh + "/src/tests/class_tests/openms/data/ProtXMLFile_input_1.
↪protXML", "test.protXML")
protein_ids = ProteinIdentification()
peptide_ids = PeptideIdentification()
ProtXMLFile().load("test.protXML", protein_ids, peptide_ids)
# storing protein XML file is not yet supported
```

note how each data file produces two vectors of type `ProteinIdentification` and `PeptideIdentification` which also means that conversion between two data types is trivial: load data from one data file and use the storage function of the other file.

4.2 Quantiative data (featureXML, consensusXML)

OpenMS stores quantitative information in the internal `featureXML` and `consensusXML` data formats. These can be accessed as follows:

```
import urllib
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urllib.urlretrieve (gh + "/src/tests/topp/FeatureFinderCentroided_1_output.featureXML
↪", "test.featureXML")
features = FeatureMap()
FeatureXMLFile().load("test.featureXML", features)
FeatureXMLFile().store("test.out.featureXML", features)
```

and for `consensusXML`

```
import urllib
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urllib.urlretrieve (gh + "/src/tests/class_tests/openms/data/ConsensusXMLFile_1.
↪consensusXML", "test.consensusXML")
features = ConsensusMap()
ConsensusXMLFile().load("test.consensusXML", features)
ConsensusXMLFile().store("test.out.consensusXML", features)
```

4.3 Transition data (TraML)

The TraML data format allows you to store transition information for targeted experiments (SRM / MRM / PRM / DIA).

```
import urllib
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urllib.urlretrieve (gh + "/src/tests/topp/ConvertTSVToTraML_output.TraML", "test.TraML
↪")
targeted_exp = TargetedExperiment()
TraMLFile().load("test.TraML", targeted_exp)
TraMLFile().store("test.out.TraML", targeted_exp)
```

CHAPTER 5

MS Data

5.1 Spectrum

The most important container for raw data and peaks is MSSpectrum which we have already worked with in the Getting Started tutorial. MSSpectrum is a container for 1-dimensional peak data (a container of Peak1D). You can access these objects directly, however it is faster to use the get_peaks() and set_peaks functions which use Python numpy arrays for raw data access. Meta-data is accessible through inheritance of the SpectrumSettings objects which handles meta data of a spectrum.

In the following example program, a MSSpectrum is filled with peaks, sorted according to mass-to-charge ratio and a selection of peak positions is displayed.

First we create a spectrum and insert peaks with descending mass-to-charge ratios:

```
1  from pyopenms import *
2  spectrum = MSSpectrum()
3  mz = range(1500, 500, -100)
4  i = [0 for mass in mz]
5  spectrum.set_peaks([mz, i])
6
7  # Sort the peaks according to ascending mass-to-charge ratio
8  spectrum.sortByPosition()
9
10 # Iterate over spectrum of those peaks
11 for p in spectrum:
12     print(p.getMZ(), p.getIntensity())
13
14 # More efficient peak access with get_peaks()
15 for mz, i in zip(*spectrum.get_peaks()):
16     print(mz, i)
17
18 # Access a peak by index
19 print(spectrum[2].getMZ(), spectrum[2].getIntensity())
```

Note how lines 11-12 (as well as line 19) use the direct access to the Peak1D objects (explicit iteration through the

`MSSpectrum` object, which is convenient but slow since a new `Peak1D` object needs to be created each time) while lines 15-16 use the faster access through numpy arrays. Direct iteration is only shown for demonstration purposes and should not be used in production code.

To discover the full set of functionality of `MSSpectrum`, we use the `help()` function. In particular, we find several important sets of meta information attached to the spectrum including retention time, the ms level (MS1, MS2, . . .), precursor ion, ion mobility drift time and extra data arrays.

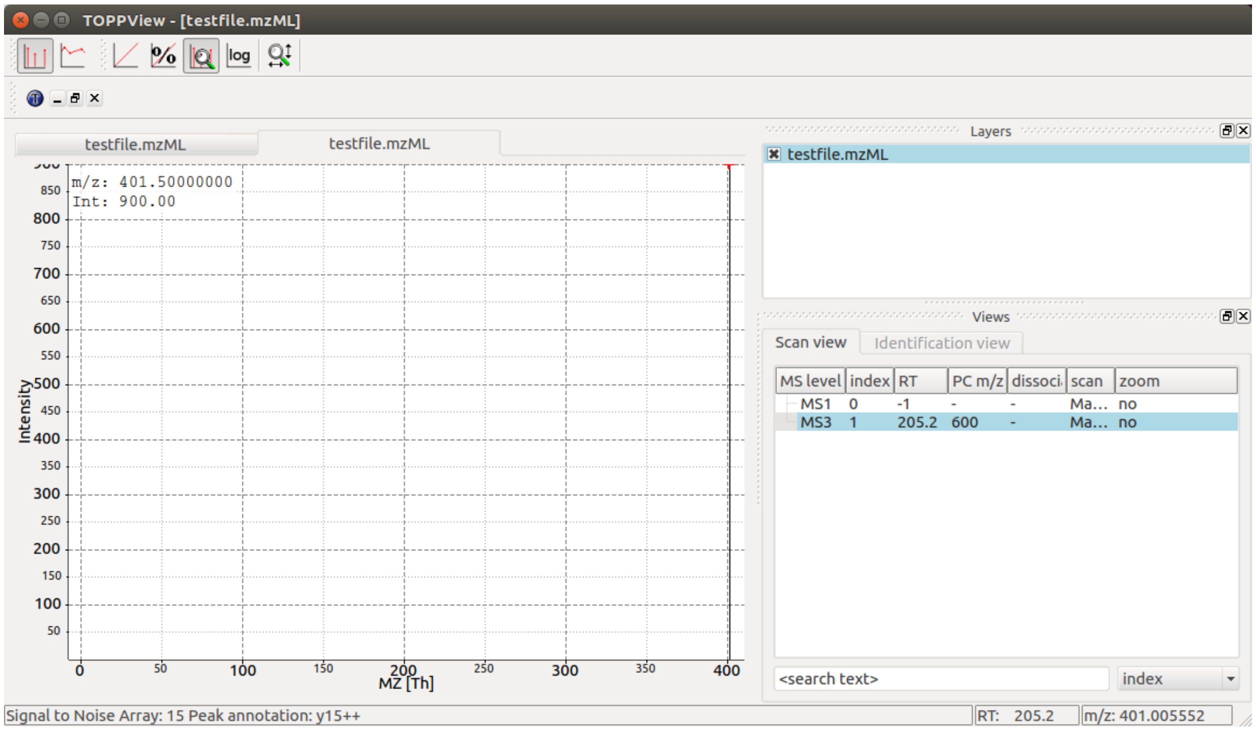
```
from pyopenms import *
help(MSSpectrum)
```

We now set several of these properties in a current MSSpectrum:

```
1  from pyopenms import *
2
3  spectrum = MSSpectrum()
4  spectrum.setDriftTime(25) # 25 ms
5  spectrum.setRT(205.2) # 205.2 s
6  spectrum.setMSLevel(3) # MS3
7  p = Precursor()
8  p.setIsolationWindowLowerOffset(1.5)
9  p.setIsolationWindowUpperOffset(1.5)
10 p.setMZ(600) # isolation at 600 +/- 1.5 Th
11 p.setActivationEnergy(40) # 40 eV
12 p.setCharge(4) # 4+ ion
13 spectrum.setPrecursors( [p] )
14
15 # Add raw data to spectrum
16 spectrum.set_peaks( ([401.5], [900]) )
17
18 # Additional data arrays / peak annotations
19 fda = FloatDataArray()
20 fda.setName("Signal to Noise Array")
21 fda.push_back(15)
22 sda = StringDataArray()
23 sda.setName("Peak annotation")
24 sda.push_back("y15++")
25 spectrum.setFloatDataArrays( [fda] )
26 spectrum.setStringDataArrays( [sda] )
27
28 # Add spectrum to MSExperiment
29 exp = MSExperiment()
30 exp.addSpectrum(spectrum)
31
32 # Add second spectrum and store as mzML file
33 spectrum2 = MSSpectrum()
34 spectrum2.set_peaks( ([1, 2], [1, 2]) )
35 exp.addSpectrum(spectrum2)
36
37 MzMLFile().store("testfile.mzML", exp)
```

We have created a single spectrum on line 3 and add meta information (drift time, retention time, MS level, precursor charge, isolation window and activation energy) on lines 4-13. We next add actual peaks into the spectrum (a single peak at 401.5 *m/z* and 900 intensity) on line 16 and on lines 19-26 add further meta information in the form of additional data arrays for each peak (e.g. one trace describes "Signal to Noise" for each peak and the second traces describes the "Peak annotation", identifying the peak at 401.5 *m/z* as a doubly charged y15 ion). Finally, we add the spectrum to a `MSExperiment` container on lines 29-30 and store the container in using the `MzMLFile` class in a file called "testfile.mzML" on line 37. To ensure our viewer works as expected, we add a second spectrum to the file before storing the file.

You can now open the resulting spectrum in a spectrum viewer. We use the OpenMS viewer `TOPPView` (which you will get when you install OpenMS from the official website) and look at our MS3 spectrum:



```
m/z: 401.50000000
Int: 900.00
```

MS level	index	RT	PC m/z	dissoci	scan	zoom
MS1	0	-1	-	-	Ma...	no
MS3	1	205.2	600	-	Ma...	no

```
Signal to Noise Array: 15 Peak annotation: y15++    RT:  205.2    m/z: 401.005552
```

TOPPView displays our MS3 spectrum with its single peak at 401.5 *m/z* and it also correctly displays its retention time at 205.2 seconds and precursor isolation target of 600.0 *m/z*. Notice how TOPPView displays the information about the S/N for the peak (S/N = 15) and its annotation as `y15++` in the status bar below when the user clicks on the peak at 401.5 *m/z* as shown in the screenshot.

5.2 LC-MS/MS Experiment

In OpenMS, LC-MS/MS injections are represented as so-called peak maps (using the `MSExperiment` class), which we have already encountered above. The `MSExperiment` class can hold a list of `MSSpectrum` object (as well as a list of `MSChromatogram` objects, see below). The `MSExperiment` object holds such peak maps as well as meta-data about the injection. Access to individual spectra is performed through `MSExperiment.getSpectrum` and `MSExperiment.getChromatogram`.

In the following code, we create an `MSExperiment` and populate it with several spectra:

```python
# The following examples creates an MSExperiment which holds six
# MSSpectrum instances.
exp = MSExperiment()
for i in range(6):
    spectrum = MSSpectrum()
    spectrum.setRT(i)
    spectrum.setMSLevel(1)
    for mz in range(500, 900, 100):
      peak = Peak1D()
      peak.setMZ(mz + i)
      peak.setIntensity(100 - 25*abs(i-2.5) )
```

(continues on next page)

(continued from previous page)

```
12         spectrum.push_back(peak)
13     exp.addSpectrum(spectrum)
14 
15 # Iterate over spectra
16 for spectrum in exp:
17     for peak in spectrum:
18         print (spectrum.getRT(), peak.getMZ(), peak.getIntensity())
```

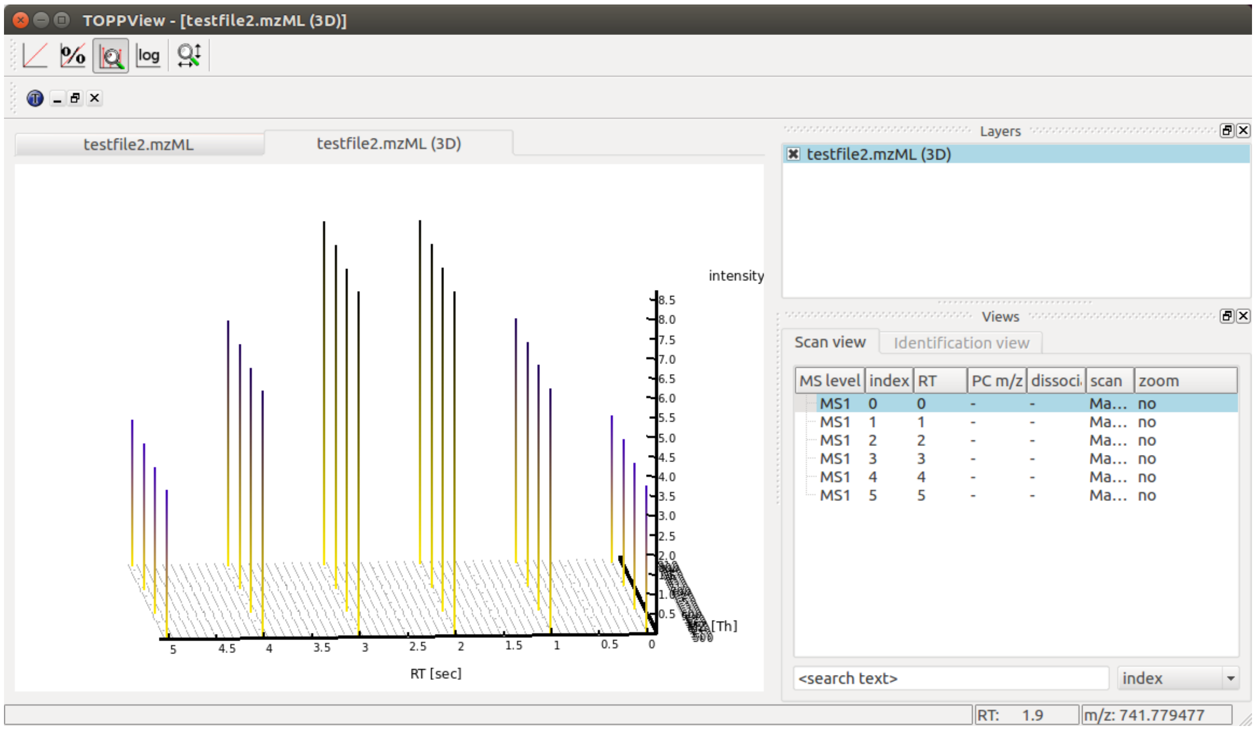
In the above code, we create six instances of `MSSpectrum` (line 4), populate it with three peaks at 500, 900 and 100 *m/z* and append them to the `MSExperiment` object (line 13). We can easily iterate over the spectra in the whole experiment by using the intuitive iteration on lines 16-18 or we can use list comprehensions to sum up intensities of all spectra that fulfill certain conditions:

```
>>> # Sum intensity of all spectra between RT 2.0 and 3.0
>>> print(sum([p.getIntensity() for s in exp
...                 if s.getRT() >= 2.0 and s.getRT() <= 3.0 for p in s]))
700.0
>>> 87.5 * 8
700.0
>>>
```

We can again store the resulting experiment containing the six spectra as mzML using the `MzMLFile` object:

```
# Store as mzML
MzMLFile().store("testfile2.mzML", exp)
```

Again we can visualize the resulting data using `TOPPView` using its 3D viewer capability, which shows the six scans over retention time where the traces first increase and then decrease in intensity:



MS level	index	RT	PC m/z	dissoci	scan	zoom
MS1	0	0	-	-	Ma...	no
MS1	1	1	-	-	Ma...	no
MS1	2	2	-	-	Ma...	no
MS1	3	3	-	-	Ma...	no
MS1	4	4	-	-	Ma...	no
MS1	5	5	-	-	Ma...	no

5.3 Chromatogram

An additional container for raw data is the `MSChromatogram` container, which is highly analogous to the `MSSpectrum` container, but contains an array of `ChromatogramPeak` and is derived from `ChromatogramSettings`:

```python
from pyopenms import *
import numpy as np

def gaussian(x, mu, sig):
    return np.exp(-np.power(x - mu, 2.) / (2 * np.power(sig, 2.)))

# Create new chromatogram
chromatogram = MSChromatogram()

# Set raw data (RT and intensity)
rt = range(1500, 500, -100)
i = [gaussian(rtime, 1000, 150) for rtime in rt]
chromatogram.set_peaks([rt, i])

# Sort the peaks according to ascending retention time
chromatogram.sortByPosition()

# Iterate over chromatogram of those peaks
for p in chromatogram:
    print(p.getRT(), p.getIntensity())

# More efficient peak access with get_peaks()
for rt, i in zip(*chromatogram.get_peaks()):
    print(rt, i)

# Access a peak by index
print(chromatogram[2].getRT(), chromatogram[2].getIntensity())
```

We now again add meta information to the chromatogram:

```python
chromatogram.setNativeID("Trace XIC@405.2")

# Store a precursor ion for the chromatogram
p = Precursor()
p.setIsolationWindowLowerOffset(1.5)
p.setIsolationWindowUpperOffset(1.5)
p.setMZ(405.2) # isolation at 405.2 +/- 1.5 Th
p.setActivationEnergy(40) # 40 eV
p.setCharge(2) # 2+ ion
p.setMetaValue("description", chromatogram.getNativeID())
p.setMetaValue("peptide_sequence", chromatogram.getNativeID())
chromatogram.setPrecursor(p)

# Also store a product ion for the chromatogram (e.g. for SRM)
p = Product()
p.setMZ(603.4) # transition from 405.2 -> 603.4
chromatogram.setProduct(p)

# Store as mzML
exp = MSExperiment()
exp.addChromatogram(chromatogram)
```

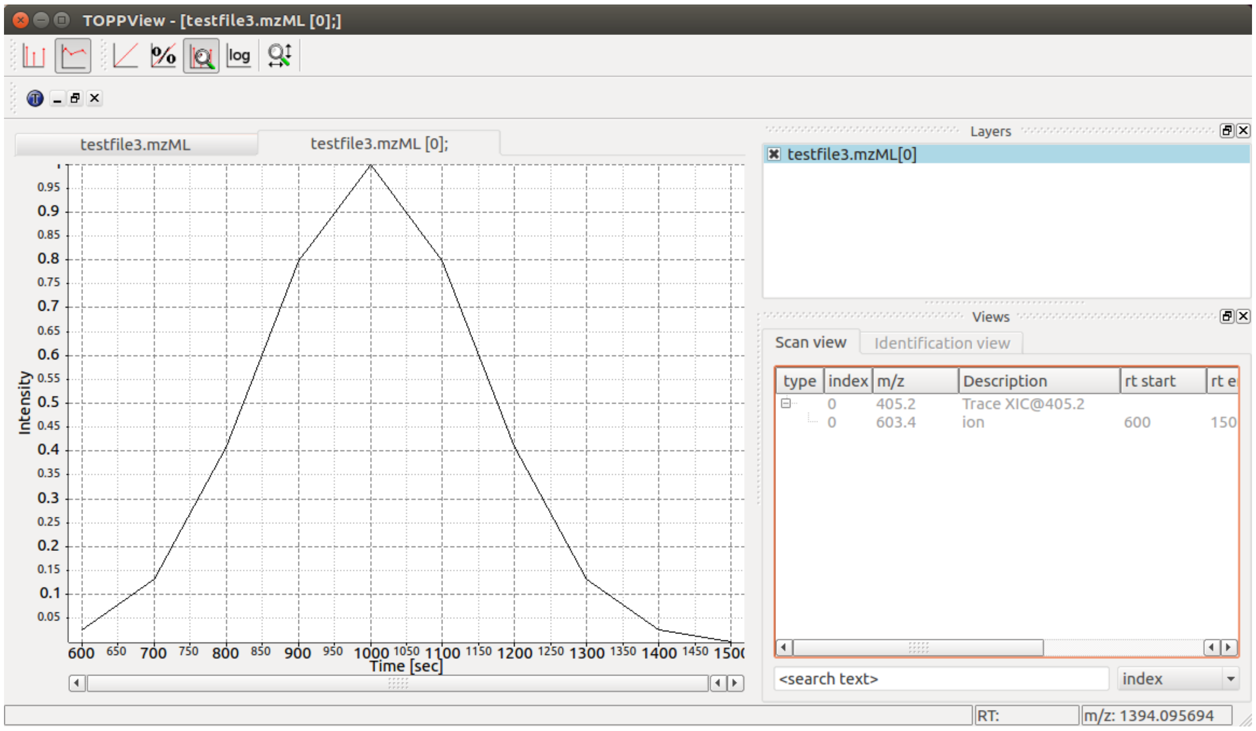
(continues on next page)

(continued from previous page)

```
MzMLFile().store("testfile3.mzML", exp)
```

This shows how the `MSExperiment` class can hold spectra as well as chromatograms.

Again we can visualize the resulting data using `TOPPView` using its chromatographic viewer capability, which shows the peak over retention time:



type	index	m/z	Description	rt start	rt e
⊟	0	405.2	Trace XIC@405.2		
	0	603.4	ion	600	150

Note how the annotation using precursor and production mass of our XIC chromatogram is displayed in the viewer.

CHAPTER 6

Chemistry

6.1 Elements

OpenMS has representations for various chemical concepts including molecular formulas, isotopes, amino acid sequences and modifications. First, we look at how elements are stored in OpenMS:

```
from pyopenms import *

edb = ElementDB()

edb.hasElement("O")
edb.hasElement("S")

oxygen = edb.getElement("O")
oxygen.getName()
oxygen.getSymbol()
oxygen.getMonoWeight()
isotopes = oxygen.getIsotopeDistribution()

sulfur = edb.getElement("S")
sulfur.getName()
sulfur.getSymbol()
sulfur.getMonoWeight()
isotopes = sulfur.getIsotopeDistribution()
for iso in isotopes.getContainer():
    print (iso)
```

As we can see, OpenMS knows common elements like Oxygen and Sulfur as well as their isotopic distribution. These values are stored in `Elements.xml` in the OpenMS share folder and can, in principle, be modified. The above code outputs the isotopes of sulfur and their abundance:

```
(32, 0.9493)
(33, 0.0076)
```

(continues on next page)

(continued from previous page)

```
(34, 0.0429)
(36, 0.0002)
```

6.2 Molecular Formula

Elements can be combined to molecular formulas (`EmpiricalFormula`) which can be used to describe small molecules or peptides. The class supports a large number of operations like addition and subtraction. A simple example is given in the next few lines of code.

```
from pyopenms import *

methanol = EmpiricalFormula("CH3OH")
water = EmpiricalFormula("H2O")
wm = EmpiricalFormula(str(water) + str(methanol))
print(wm)

isotopes = wm.getIsotopeDistribution(3)
for iso in isotopes.getContainer():
    print (iso)
```

which produces

```
C1H6O2
(50, 0.9838702160434344)
(51, 0.012069784261989644)
(52, 0.004059999694575987)
```

6.3 AA Residue

An amino acid residue is represented in OpenMS by the class `Residue`. It provides a container for the amino acids as well as some functionality. The class is able to provide information such as the isotope distribution of the residue, the average and monoisotopic weight. The residues can be identified by their full name, their three letter abbreviation or the single letter abbreviation. The residue can also be modified, which is implemented in the `Modification` class. Additional less frequently used parameters of a residue like the gas-phase basicity and pk values are also available.

```
>>> from pyopenms import *
>>> lys = ResidueDB().getResidue("Lysine")
>>> lys.getName()
'Lysine'
>>> lys.getThreeLetterCode()
'LYS'
>>> lys.getOneLetterCode()
'K'
>>> lys.getAverageWeight(Residue.ResidueType.Full)
146.18788276708443
>>> lys.getMonoWeight(Residue.ResidueType.Full)
146.1055284466
>>> lys.getPka()
2.16
```

As we can see, OpenMS knows common amino acids like lysine as well as some properties of them. These values are stored in `Residues.xml` in the OpenMS share folder and can, in principle, be modified.

6.4 Modifications

An amino acid residue modification is represented in OpenMS by the class `ResidueModification`. The known modifications are stored in the `ModificationsDB` object, which is capable of retrieving specific modifications. It contains UniMod as well as PSI modifications.

```
from pyopenms import *
ts = ResidueModification.TermSpecificity
ox = ModificationsDB().getModification("Oxidation", "", ts.ANYWHERE)
print(ox.getUniModAccession())
print(ox.getUniModRecordId())
print(ox.getDiffMonoMass())
print(ox.getId())
print(ox.getFullId())
print(ox.getFullName())
print(ox.getDiffFormula())
```

which outputs

```
UniMod:35
35
15.994915
Oxidation
Oxidation (N)
Oxidation or Hydroxylation
O1
```

thus providing information about the "Oxidation" modification. As above, we can investigate the isotopic distribution of the modification (which in this case is identical to the one of Oxygen by itself):

```
isotopes = ox.getDiffFormula().getIsotopeDistribution(5)
for iso in isotopes.getContainer():
    print (iso)
```

In the next section, we will look at how to combine amino acids and modifications to form amino acid sequences (peptides).

CHAPTER 7

Peptides and Proteins

7.1 AA Sequences

The `AASequence` class handles amino acid sequences in OpenMS. A string of amino acid residues can be turned into a instance of `AASequence` to provide some commonly used operations and data. The implementation supports mathematical operations like addition or subtraction. Also, average and mono isotopic weight and isotope distributions are accessible.

Weights, formulas and isotope distribution can be calculated depending on the charge state (additional proton count in case of positive ions) and ion type. Therefore, the class allows for a flexible handling of amino acid strings.

A very simple example of handling amino acid sequence with AASequence is given in the next few lines, which also calculates the weight of the `(M)` and `(M+2H)2+` ions.

```python
from pyopenms import *
seq = AASequence.fromString("DFPIANGER", True)
prefix = seq.getPrefix(4)
suffix = seq.getSuffix(5)
concat = seq + seq

print(seq)
print(concat)
print(suffix)
seq.getMonoWeight(Residue.ResidueType.Full, 0)
seq.getMonoWeight(Residue.ResidueType.Full, 2) / 2.0
concat.getMonoWeight(Residue.ResidueType.Full, 0)
```

We can now combine our knowledge of `AASequence` with what we learned above about `EmpiricalFormula` to get accurate mass and isotope distributions from the amino acid sequence:

```python
seq_formula = seq.getFormula(Residue.ResidueType.Full, 0)
print(seq_formula)

isotopes = seq_formula.getIsotopeDistribution(6)
```

(continues on next page)

(continued from previous page)

```
5  for iso in isotopes.getContainer():
6      print (iso)
7
8  suffix = seq.getSuffix(3) # y3 ion "GER"
9  print(suffix)
10 y3_formula = suffix.getFormula(Residue.ResidueType.YIon, 2) # y3++ ion
11 suffix.getMonoWeight(Residue.ResidueType.YIon, 2) / 2.0 # CORRECT
12 suffix.getMonoWeight(Residue.ResidueType.XIon, 2) / 2.0 # CORRECT
13 suffix.getMonoWeight(Residue.ResidueType.BIon, 2) / 2.0 # INCORRECT
14 print(y3_formula)
15 print(seq_formula)
```

Note on lines 11 to 13 we need to remember that we are dealing with an ion of the x/y/z series since we used a suffix of the original peptide and using any other ion type will produce a different mass-to-charge ratio (and while "GER" would also be a valid "x3" ion, note that it *cannot* be a valid ion from the a/b/c series and therefore the mass on line 13 cannot refer to the same input peptide "DFPIANGER" since its "b3" ion would be "DFP" and not "GER").

7.2 Modified AA Sequences

The `AASequence` class can also handle modifications, modifications are specified using a unique string identifier present in the `ModificationsDB` in round brackets after the modified amino acid or by providing the mass of the residue in square brackets. For example `AASequence.fromString(".DFPIAM(Oxidation)GER.", True)` creates an instance of the peptide "DFPIAMGER" with an oxidized methionine. There are multiple ways to specify modifications, and `AASequence.fromString("DFPIAM(UniMod:35)GER", True)`, `AASequence.fromString("DFPIAM[+16]GER", True)` and `AASequence.fromString("DFPIAM[147]GER", True)` are all equivalent).

N- and C-terminal modifications are represented by brackets to the right of the dots terminating the sequence. For example, `".(Dimethyl)DFPIAMGER."` and `".DFPIAMGER.(Label:18O(2))"` represent the labelling of the N- and C-terminus respectively, but `".DFPIAMGER(Phospho)."` will be interpreted as a phosphorylation of the last arginine at its side chain.

```
from pyopenms import *
seq = AASequence.fromString("PEPTIDESEKUEM(Oxidation)CER", True)
print(seq.toString())
print(seq.toUnmodifiedString())
print(seq.toBracketString(True, []))
print(seq.toBracketString(False, []))

print(AASequence.fromString("DFPIAM(UniMod:35)GER", True))
print(AASequence.fromString("DFPIAM[+16]GER", True))
print(AASequence.fromString("DFPIAM[+15.99]GER", True))
print(AASequence.fromString("DFPIAM[147]GER", True))
print(AASequence.fromString("DFPIAM[147.035405]GER", True))
```

The above code outputs:

```
PEPTIDESEKUEM(Oxidation)CER
PEPTIDESEKUEMCER
PEPTIDESEKUEM[147]CER
PEPTIDESEKUEM[147.0354000171]CER

DFPIAM(Oxidation)GER
DFPIAM(Oxidation)GER
```

(continues on next page)

(continued from previous page)

```
DFPIAM(Oxidation)GER
DFPIAM(Oxidation)GER
DFPIAM(Oxidation)GER
```

Note there is a subtle difference between `AASequence.fromString(".DFPIAM[+16]GER.")` and `AASequence.fromString(".DFPIAM[+15.9949]GER.")` - while the former will try to find the first modification matching to a mass difference of 16 +/- 0.5, the latter will try to find the closest matching modification to the exact mass. The exact mass approach usually gives the intended results while the first approach may or may not.

Arbitrary/unknown amino acids (usually due to an unknown modification) can be specified using tags preceded by X: "X[weight]". This indicates a new amino acid ("X") with the specified weight, e.g. `"RX[148.5]T"`. Note that this tag does not alter the amino acids to the left (R) or right (T). Rather, X represents an amino acid on its own. Be careful when converting such AASequence objects to an EmpiricalFormula using `getFormula()`, as tags will not be considered in this case (there exists no formula for them). However, they have an influence on `getMonoWeight()` and `getAverageWeight()`!

7.3 Proteins

Protein sequences can be accessed through the `FASTAEntry` object and can be read and stored on disk using a `FASTAFile`:

```python
from pyopenms import *
bsa = FASTAEntry()
bsa.sequence = "MKWVTFISLLLLFSSAYSRGVFRRDTHKSEIAHRFKDLGE"
bsa.description = "BSA Bovine Albumin (partial sequence)"
bsa.identifier = "BSA"
alb = FASTAEntry()
alb.sequence = "MKWVTFISLLFLFSSAYSRGVFRRDAHKSEVAHRFKDLGE"
alb.description = "ALB Human Albumin (partial sequence)"
alb.identifier = "ALB"

entries = [bsa, alb]

f = pyopenms.FASTAFile()
f.store("example.fasta", entries)
```

Afterwards, the protein sequences can be read again using:

```python
from pyopenms import *
entries = []
f = FASTAFile()
f.load("example.fasta", entries)
print( len(entries) )
for e in entries:
  print (e.identifier, e.sequence)
```

7.4 TheoreticalSpectrumGenerator

This class implements a simple generator which generates tandem MS spectra from a given peptide charge combination. There are various options which influence the occurring ions and their intensities.

```
from pyopenms import *

tsg = TheoreticalSpectrumGenerator()
spec1 = MSSpectrum()
spec2 = MSSpectrum()
peptide = AASequence.fromString("DFPIANGER", True)
# standard behavior is adding b- and y-ions of charge 1
p = Param()
p.setValue("add_b_ions", "false", "Add peaks of b-ions to the spectrum")
tsg.setParameters(p)
tsg.getSpectrum(spec1, peptide, 1, 1)
p.setValue("add_b_ions", "true", "Add peaks of a-ions to the spectrum")
p.setValue("add_metainfo", "true", "")
tsg.setParameters(p)
tsg.getSpectrum(spec2, peptide, 1, 2)
print("Spectrum 1 has", spec1.size(), "peaks.")
print("Spectrum 2 has", spec2.size(), "peaks.")

# Iterate over annotated ions and their masses
for ion, peak in zip(spec2.getStringDataArrays()[0], spec2):
    print(ion, peak.getMZ())
```

which outputs:

```
Spectrum 1 has 8 peaks.
Spectrum 2 has 30 peaks.

y1++ 88.0631146901
b2++ 132.05495569
y2++ 152.584411802
y1+ 175.118952913
[...]
```

The example shows how to put peaks of a certain type, y-ions in this case, into a spectrum. Spectrum 2 is filled with a complete spectrum of all peaks (a-, b-, y-ions and losses). The `TheoreticalSpectrumGenerator` has many parameters which have a detailed description located in the class documentation. For the first spectrum, no b ions are added. Note how the `add_metainfo` parameter in the second example populates the `StringDataArray` of the output spectrum, allowing us to iterate over annotated ions and their masses.

CHAPTER 8

Digestion

8.1 Proteolytic Digestion with Trypsin

OpenMS has classes for proteolytic digestion which can be used as follows:

```
from pyopenms import *
import urllib
urllib.urlretrieve ("http://www.uniprot.org/uniprot/P02769.fasta", "bsa.fasta")

dig = EnzymaticDigestion()
dig.getEnzymeName() # Trypsin
bsa = "".join([l.strip() for l in open("bsa.fasta").readlines()[1:]])
bsa = AASequence.fromString(bsa, True)
result = []
dig.digest(bsa, result)
print(result[4])
len(result) # 82 peptides
```

8.2 Proteolytic Digestion with Lys-C

We can of course also use different enzymes, these are defined `Enzyme.xml` file and can be accessed using the `EnzymesDB`

```
names = []
EnzymesDB().getAllNames(names)
len(names) # 25 by default
e = EnzymesDB().getEnzyme('Lys-C')
e.getRegExDescription()
e.getRegEx()
```

Now that we have learned about the other enzymes available, we can use it to cut out protein of interest:

```
from pyopenms import *
import urllib
urllib.urlretrieve ("http://www.uniprot.org/uniprot/P02769.fasta", "bsa.fasta")

dig = EnzymaticDigestion()
dig.setEnzyme('Lys-C')
bsa = "".join([l.strip() for l in open("bsa.fasta").readlines()[1:]])
bsa = AASequence.fromString(bsa, True)
result = []
dig.digest(bsa, result)
print(result[4])
len(result) # 57 peptides
```

We now get different digested peptides (57 vs 82) and the fourth peptide is now `GLVLIAFSQYLQQCPFDEHVK` instead of `DTHK` as with Trypsin (see above).

CHAPTER 9

Identification Data

In OpenMS, identifications of peptides, proteins and small molecules are stored in dedicated data structures. These data structures are typically stored to disc as idXML or mzIdentML file. The highest-level structure is `ProteinIdentification`. It stores all identified proteins of an identification run as ProteinHit objects plus additional metadata (search parameters, etc.). Each `ProteinHit` contains the actual protein accession, an associated score, and (optionally) the protein sequence.

A `PeptideIdentification` object stores the data corresponding to a single identified spectrum or feature. It has members for the retention time, m/z, and a vector of `PeptideHit` objects. Each `PeptideHit` stores the information of a specific peptide-to-spectrum match or PSM (e.g., the score and the peptide sequence). Each `PeptideHit` also contains a vector of `PeptideEvidence` objects which store the reference to one or more (in the case the peptide maps to multiple proteins) proteins and the position therein.

9.1 ProteinIdentification

We can create an object of type `ProteinIdentification` and populate it with `ProteinHit` objects as follows:

```
from pyopenms import *
from __future__ import print_function

# Create new protein identification object corresponding to a single search
protein_id = ProteinIdentification()
protein_id.setIdentifier("IdentificationRun1")

# Each ProteinIdentification object stores a vector of protein hits
protein_hit = ProteinHit()
protein_hit.setAccession("MyAccession")
protein_hit.setSequence("PEPTIDEPEPTIDEPEPTIDEPEPTIDER")
protein_hit.setScore(1.0)

protein_id.setHits([protein_hit])
```

We have now added a single `ProteinHit` with the accession `MyAccession` to the `ProteinIdentification` object (note how on line 14 we directly added a list of size 1). We can continue to add meta-data for the whole identification run (such as search parameters):

```
now = DateTime.now()
date_string = now.getDate()
protein_id.setDateTime(now)

# Example of possible search parameters
search_parameters = SearchParameters() # ProteinIdentification::SearchParameters
search_parameters.db = "database"
search_parameters.charges = "+2"
protein_id.setSearchParameters(search_parameters)

# Some search engine meta data
protein_id.setSearchEngineVersion("v1.0.0")
protein_id.setSearchEngine("SearchEngine")
protein_id.setScoreType("HyperScore")

# Iterate over all protein hits
for hit in protein_id.getHits():
  print("Protein hit accession:", hit.getAccession())
  print("Protein hit sequence:", hit.getSequence())
  print("Protein hit score:", hit.getScore())
```

9.2 PeptideIdentification

Next, we can also create a `PeptideIdentification` object and add corresponding `PeptideHit` objects:

```
peptide_id = PeptideIdentification()

peptide_id.setRT(1243.56)
peptide_id.setMZ(440.0)
peptide_id.setScoreType("ScoreType")
peptide_id.setHigherScoreBetter(False)
peptide_id.setIdentifier("IdentificationRun1")

# define additional meta value for the peptide identification
peptide_id.setMetaValue("AdditionalMetaValue", "Value")

# create a new PeptideHit (best PSM)
peptide_hit = PeptideHit()
peptide_hit.setScore(1.0)
peptide_hit.setRank(1)
peptide_hit.setCharge(2)
peptide_hit.setSequence(AASequence.fromString("DLQM(Oxidation)TQSPSSLSVSVGDR", True))

# create a new PeptideHit (second best PSM)
peptide_hit2 = PeptideHit()
peptide_hit2.setScore(0.5)
peptide_hit2.setRank(2)
peptide_hit2.setCharge(2)
peptide_hit2.setSequence(AASequence.fromString("QDLM(Oxidation)TQSPSSLSVSVGDR", True))

# add PeptideHit to PeptideIdentification
```

(continues on next page)

(continued from previous page)

```
peptide_id.setHits([peptide_hit, peptide_hit2])

# Iterate over PeptideIdentification
peptide_ids = [peptide_id]
for peptide_id in peptide_ids:
  # Peptide identification values
  print ("Peptide ID m/z:", peptide_id.getMZ())
  print ("Peptide ID rt:", peptide_id.getRT())
  print ("Peptide ID score type:", peptide_id.getScoreType())
  # PeptideHits
  for hit in peptide_id.getHits():
    print(" - Peptide hit rank:", hit.getRank())
    print(" - Peptide hit sequence:", hit.getSequence())
    print(" - Peptide hit score:", hit.getScore())
```

This allows us to represent single spectra (`PeptideIdentification` at *m/z* 440.0 and *rt* 1234.56) with possible identifications that are ranked by score. In this case, apparently two possible peptides match the spectrum which have the first three amino acids in a different order “DLQ” vs “QDL”).

9.3 Storage on disk

Finally, we can store the peptide and protein identification data in a `idXML` file (a OpenMS internal file format which we have previously discussed here) which we would do as follows:

```
# Store the identification data in an idXML file
IdXMLFile().store("out.idXML", [protein_id], peptide_ids)
# and load it back into memory
prot_ids = []; pep_ids = []
IdXMLFile().load("out.idXML", prot_ids, pep_ids)

# Iterate over all protein hits
for protein_id in prot_ids:
  for hit in protein_id.getHits():
    print("Protein hit accession:", hit.getAccession())
    print("Protein hit sequence:", hit.getSequence())
    print("Protein hit score:", hit.getScore())

# Iterate over PeptideIdentification
for peptide_id in pep_ids:
  # Peptide identification values
  print ("Peptide ID m/z:", peptide_id.getMZ())
  print ("Peptide ID rt:", peptide_id.getRT())
  print ("Peptide ID score type:", peptide_id.getScoreType())
  # PeptideHits
  for hit in peptide_id.getHits():
    print(" - Peptide hit rank:", hit.getRank())
    print(" - Peptide hit sequence:", hit.getSequence())
    print(" - Peptide hit score:", hit.getScore())
```

CHAPTER 10

Quantitative Data

10.1 Feature

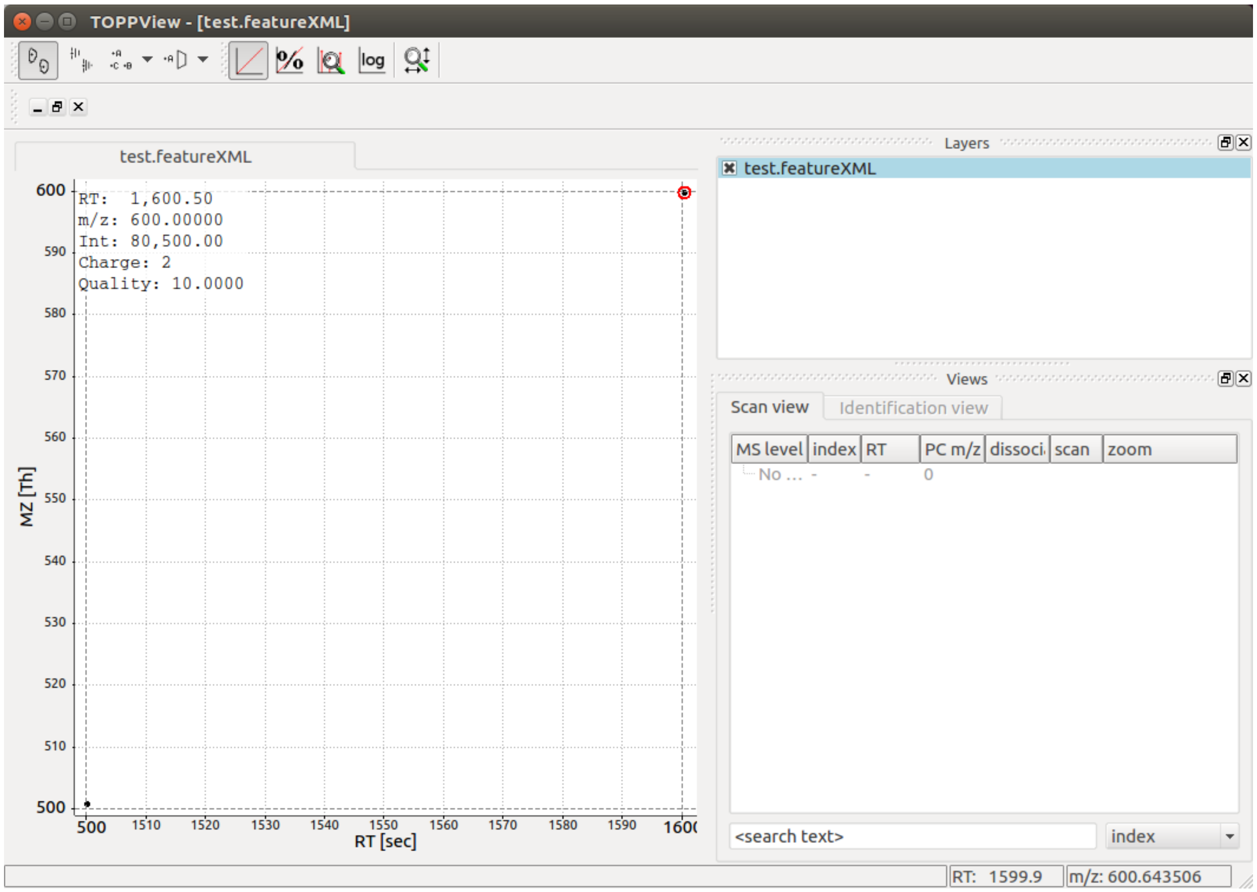
In OpenMS, information about quantitative data is stored in a so-called `Feature` which we have previously discussed here. Each `Feature` represents a region in RT and *m/z* space use for quantitative analysis.

```python
from pyopenms import *
feature = Feature()
feature.setMZ( 500.9 )
feature.setCharge(2)
feature.setRT( 1500.1 )
feature.setIntensity( 30500 )
feature.setOverallQuality( 10 )
```

Usually, the quantitative features would be produced by a so-called "FeatureFinder" algorithm, which we will discuss in the next chapter. The features can be stored in a `FeatureMap` and written to disk.

```python
fm = FeatureMap()
fm.push_back(feature)
feature.setRT(1600.5 )
feature.setCharge(2)
feature.setMZ( 600.0 )
feature.setIntensity( 80500.0 )
fm.push_back(feature)
FeatureXMLFile().store("test.featureXML", fm)
```

Visualizing the resulting map in `TOPPView` allows detection of the two features stored in the `FeatureMap` with the visualization indicating charge state, *m/z*, RT and other properties:



```
RT:   1,600.50
m/z:  600.00000
Int:  80,500.00
Charge:  2
Quality:  10.0000
```

Note that in this case only 2 features are present, but in a typical LC-MS/MS experiments, thousands of features are present.

10.2 FeatureMap

The resulting `FeatureMap` can be used in various ways to extract quantitative data directly and it supports direct iteration in Python:

```python
from pyopenms import *
fmap = FeatureMap()
FeatureXMLFile().load("test.featureXML", fmap)
for feature in fmap:
   print("Feature: ", feature.getIntensity(), feature.getRT(), feature.getMZ())
```

10.3 ConsensusFeature

Often LC-MS/MS experiments are run to compare quantitative features across experiments. In OpenMS, linked features from individual experiments are represented by a `ConsensusFeature`

```python
from pyopenms import *
feature = ConsensusFeature()
feature.setMZ( 500.9 )
```

(continues on next page)

(continued from previous page)

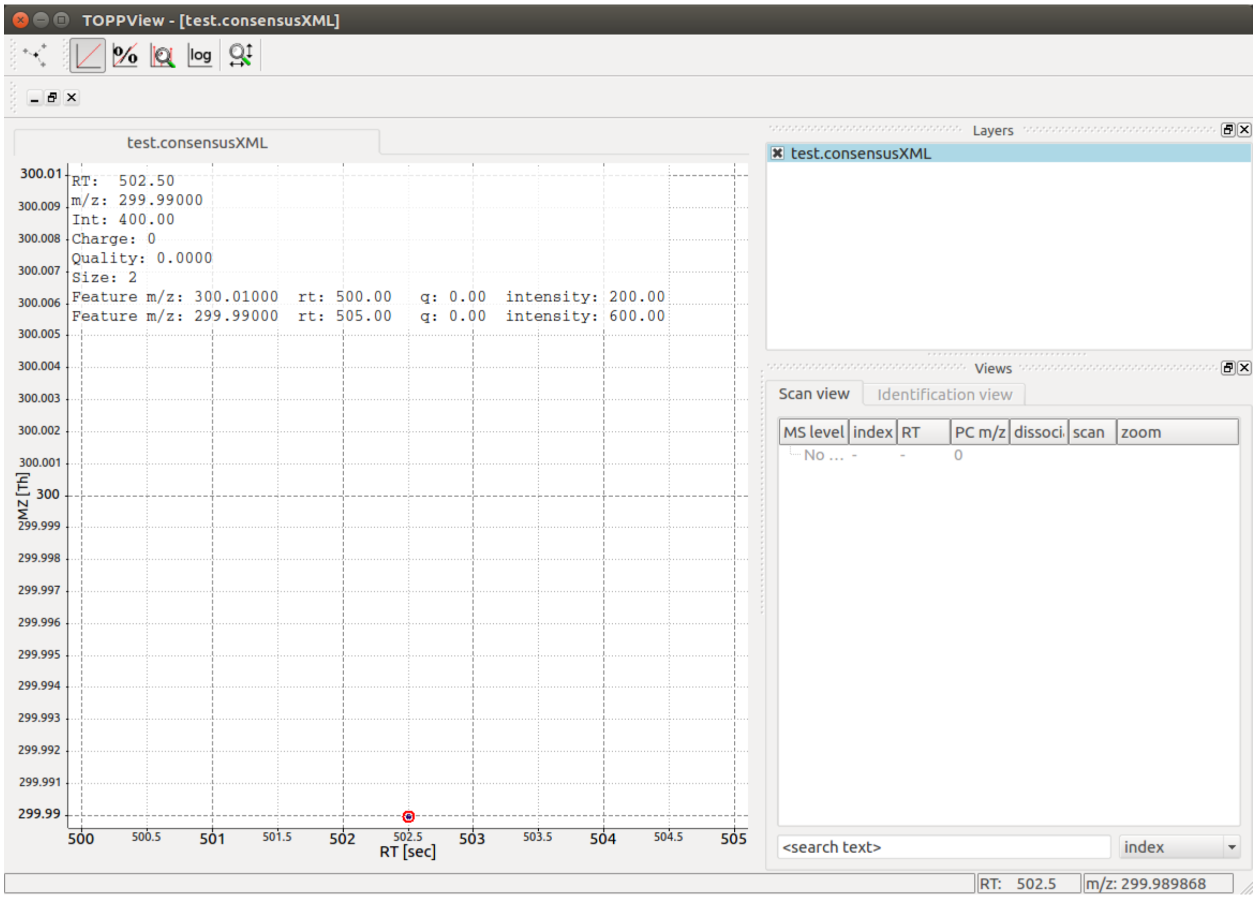
```python
feature.setCharge(2)
feature.setRT( 1500.1 )
feature.setIntensity( 80500 )

# Generate ConsensusFeature and features from two maps (with id 1 and 2)
### Feature 1
f_m1 = ConsensusFeature()
f_m1.setRT(500)
f_m1.setMZ(300.01)
f_m1.setIntensity(200)
f_m1.ensureUniqueId()
### Feature 2
f_m2 = ConsensusFeature()
f_m2.setRT(505)
f_m2.setMZ(299.99)
f_m2.setIntensity(600)
f_m2.ensureUniqueId()
feature.insert(1, f_m1 )
feature.insert(2, f_m2 )
```

We have thus added two features from two individual maps (which have the unique identifier `1` and `2`) to the `ConsensusFeature`. Next, we inspect the consensus feature, compute a "consensus" *m/z* across the two maps and output the two linked features:

```python
# The two features in map 1 and map 2 represent the same analyte at
# slightly different RT and m/z
for fh in feature.getFeatureList():
  print(fh.getMapIndex(), fh.getIntensity(), fh.getRT())

print(feature.getMZ())
feature.computeMonoisotopicConsensus()
print(feature.getMZ())

# Generate ConsensusMap and add two maps (with id 1 and 2)
cmap = ConsensusMap()
fds = { 1 : FileDescription(), 2 : FileDescription() }
fds[1].filename = b"file1"
fds[2].filename = b"file2"
cmap.setFileDescriptions(fds)

feature.ensureUniqueId()
cmap.push_back(feature)
ConsensusXMLFile().store("test.consensusXML", cmap)
```

Inspection of the generated `test.consensusXML` reveals that it contains references to two LC-MS/MS runs (`file1` and `file2`) with their respective unique identifier. Note how the two features we added before have matching unique identifiers.

Visualization of the resulting output file reveals a single `ConsensusFeature` of size 2 that links to the two individual features at their respective positions in RT and *m/z*:



```
RT:   502.50
m/z: 299.99000
Int: 400.00
Charge: 0
Quality: 0.0000
Size: 2
Feature m/z: 300.01000  rt: 500.00   q: 0.00  intensity: 200.00
Feature m/z: 299.99000  rt: 505.00   q: 0.00  intensity: 600.00
```

10.4 ConsensusMap

The resulting `ConsensusMap` can be used in various ways to extract quantitative data directly and it supports direct iteration in Python:

```python
from pyopenms import *
cmap = ConsensusMap()
ConsensusXMLFile().load("test.consensusXML", cmap)
for cfeature in cmap:
   cfeature.computeConsensus()
   print("ConsensusFeature", cfeature.getIntensity(), cfeature.getRT(), cfeature.getMZ())
   # The two features in map 1 and map 2 represent the same analyte at
   # slightly different RT and m/z
   for fh in cfeature.getFeatureList():
     print(" -- Feature", fh.getMapIndex(), fh.getIntensity(), fh.getRT())
```

CHAPTER 11

Simple Data Manipulation

Here we will look at a few simple data manipulation techniques on spectral data, such as filtering. First we will download some sample data:

```
import urllib
from pyopenms import *
urllib.urlretrieve ("http://proteowizard.sourceforge.net/example_data/tiny.pwiz.1.1.
↪mzML",
                    "test.mzML")
```

11.1 Filtering Spectra

We will filter the "test.mzML" file by only retaining spectra that match a certain identifier:

```
from pyopenms import *
inp = MSExperiment()
MzMLFile().load("test.mzML", inp)

e = MSExperiment()
for s in inp:
  if s.getNativeID().startswith("scan="):
    e.addSpectrum(s)

MzMLFile().store("test_filtered.mzML", e)
```

11.1.1 Filtering by MS level

Similarly, we can filter the `test.mzML` file by MS level:

```
from pyopenms import *
inp = MSExperiment()
```

(continues on next page)

(continued from previous page)

```
3  MzMLFile().load("test.mzML", inp)
4
5  e = MSExperiment()
6  for s in inp:
7    if s.getMSLevel() > 1:
8      e.addSpectrum(s)
9
10 MzMLFile().store("test_filtered.mzML", e)
```

Note that we can easily replace line 7 with more complicated criteria, such as filtering by MS level and scan identifier at the same time:

```
7  if s.getMSLevel() > 1 and s.getNativeID().startswith("scan="):
```

11.1.2 Filtering by scan number

Or we could use an external list of scan numbers to filter by scan numbers:

```
1  from pyopenms import *
2  inp = MSExperiment()
3  MzMLFile().load("test.mzML", inp)
4  scan_nrs = [0, 2, 5, 7]
5
6  e = MSExperiment()
7  for k, s in enumerate(inp):
8    if k in scan_nrs and s.getMSLevel() == 1:
9      e.addSpectrum(s)
10
11 MzMLFile().store("test_filtered.mzML", e)
```

It would also be easy to read the scan numbers from a file where each scan number is on its own line, thus replacing line 4 with:

```
4  scan_nrs = [int(k) for k in open("scan_nrs.txt")]
```

11.2 Filtering Spectra and Peaks

We can now move on to more advanced filtering, suppose you are interested in only a part of all fragment ion spectra. m/z. We can easily filter our data accordingly:

```
1  from pyopenms import *
2  inp = MSExperiment()
3  MzMLFile().load("test.mzML", inp)
4
5  mz_start = 6.0
6  mz_end = 12.0
7  e = MSExperiment()
8  for s in inp:
9    if s.getMSLevel() > 1:
10     filtered_mz = []
11     filtered_int = []
12     for mz, i in zip(*s.get_peaks()):
```

(continues on next page)

(continued from previous page)

```
13        if mz > mz_start and mz < mz_end:
14          filtered_mz.append(mz)
15          filtered_int.append(i)
16      s.set_peaks((filtered_mz, filtered_int))
17      e.addSpectrum(s)
18
19  MzMLFile().store("test_filtered.mzML", e)
```

Note that in a real-world application, we would set the `mz_start` and `mz_end` parameter to an actual area of interest, for example the area between 125 and 132 which contains quantitative ions for a TMT experiment.

Similarly we could change line 13 to only report peaks above a certain intensity or to only report the top N peaks in a spectrum.

11.3 Memory management

On order to save memory, we can avoid loading the whole file into memory and use the OnDiscMSExperiment for reading data.

```python
from pyopenms import *
od_exp = OnDiscMSExperiment()
od_exp.openFile("test.mzML")

e = MSExperiment()
for k in range(od_exp.getNrSpectra()):
  s = od_exp.getSpectrum(k)
  if s.getNativeID().startswith("scan="):
    e.addSpectrum(s)

MzMLFile().store("test_filtered.mzML", e)
```

Note that using the approach the output data `e` is still completely in memory and may end up using a substantial amount of memory. We can avoid that by using

```python
from pyopenms import *
od_exp = OnDiscMSExperiment()
od_exp.openFile("test.mzML")

consumer = PlainMSDataWritingConsumer("test_filtered.mzML")

e = MSExperiment()
for k in range(od_exp.getNrSpectra()):
  s = od_exp.getSpectrum(k)
  if s.getNativeID().startswith("scan="):
    consumer.consumeSpectrum(s)

del consumer
```

Make sure you do not forget `del consumer` since otherwise the final part of the mzML may not get written to disk (and the consumer is still waiting for new data).

CHAPTER 12

Parameter Handling

Parameter handling in OpenMS and pyOpenMS is usually implemented through inheritance from `DefaultParamHandler` and allow access to parameters through the `Param` object. This means, the classes implement the methods `getDefaults`, `getParameters`, `setParameters` which allows access to the default parameters, the current parameters and allows to set the parameters.

The `Param` object that is returned can be manipulated through the `setValue` and `getValue` methods (the `exists` method can be used to check for existence of a key). Using the `getDescription` method, it is possible to get a help-text for each parameter value in an interactive session without consulting the documentation.

```
from pyopenms import *
p = Param()
p.setValue("param1", 4.0, "This is value 1")
p.setValue("param2", 5.0, "This is value 2")
```

The parameters can then be accessed as

```
>>> p.asDict()
{'param2': 4.0, 'param1': 4.0}
>>> p.values()
[4.0, 4.0]
>>> p.keys()
['param1', 'param2']
>>> p.items()
[('param1', 4.0), ('param2', 4.0)]
```

CHAPTER 13

Algorithms

Most signal processing algorithms follow a similar pattern in OpenMS.

```
filter = FilterObject()
exp = MSExperiment()
# populate exp
filter.filterExperiment(exp)
```

Since they work on a single MSExperiment object, little input is needed to execute a filter directly on the data. Examples of filters that follow this pattern are `GaussFilter`, `SavitzkyGolayFilter` as well as the spectral filters `BernNorm`, `MarkerMower`, `NLargest`, `Normalizer`, `ParentPeakMower`, `Scaler`, `SpectraMerger`, `SqrtMower`, `ThresholdMower`, `WindowMower`.

using the same example file as before in

```
from pyopenms import *
exp = MSExperiment()
gf = GaussFilter()
MzMLFile().load("test.mzML", exp)
gf.filterExperiment(exp)
MzMLFile().store("test.filtered.mzML", exp)
```

CHAPTER 14

Feature Detection

One very common task in Mass Spectrometric research is the detection of 2D features in a series of MS1 scans (MS1 feature detection). OpenMS has multiple tools that can achieve these tasks, these tools are called *FeatureFinder*. Currently the following FeatureFinders are available in OpenMS:

- FeatureFinderMultiplex
- FeatureFinderMRM
- FeatureFinderCentroided
- FeatureFinderIdentification
- FeatureFinderIsotopeWavelet
- FeatureFinderMetabo
- FeatureFinderSuperHirn

One of the most commonly used FeatureFinders is the FeatureFinderCentroided which works on (high resolution) centroided data. We can use the following code to find `Features` in MS data:

```
from pyopenms import *

# Prepare data loading (save memory by only
# loading MS1 spectra into memory)
options = PeakFileOptions()
options.setMSLevels([1])
fh = MzMLFile()
fh.setOptions(options)

# Load data
input_map = MSExperiment()
fh.load("test.mzML", input_map)
input_map.updateRanges()

ff = FeatureFinder()
ff.setLogType(LogType.CMD)
```

(continues on next page)

(continued from previous page)

```

# Run the feature finder
features = FeatureMap()
seeds = FeatureMap()
params = FeatureFinder().getParameters("centroided")
ff.run(name, input_map, features, params, seeds)

features.setUniqueIds()
fh = FeatureXMLFile()
fh.store("output.featureXML", features)
print("Found", features.size(), "features")
```

You can get a sample file for analysis directly from here.

With a few lines of Python, we are able to run powerful algorithms available in OpenMS. The resulting data is held in memory (a `FeatureMap` object) and can be inspected directly using the `help(features)` comment. It reveals that the object supports iteration (through the `__iter__` function) as well as direct access (through the `__getitem__` function). We can also inspect the entry for `FeatureMap` in the pyOpenMS manual and learn about the same functions. This means we write code that uses direct access and iteration in Python as follows:

```
f0 = features[0]
for f in features:
    print f.getRT(), f.getMZ()
```

Each entry in the `FeatureMap` is a so-called `Feature` and allows direct access to the *m/z* and *RT* value from Python. Again, we can lear this by inspecting `help(f)` or by consulting the Manual.

Note: the output file that we have written (`output.featureXML`) is an OpenMS-internal XML format for storing features. You can learn more about file formats in the Reading MS data formats section.

CHAPTER 15

Build from source

To install pyOpenMS from source, you will first have to compile OpenMS successfully on your platform of choice (note that for MS Windows you will need to match your compiler and Python version). Please follow the official documenation in order to compile OpenMS for your platform. Next you will need to install the following software packages

On Microsoft Windows: you need the 64 bit C++ compiler from Visual Studio 2015 to compile the newest pyOpenMS for Python 3.5 or 3.6. This is important, else you get a different clib than Python 2.7 is built with, and pyOpenMS will crash on import.

You can install all necessary Python packages on which pyOpenMS depends through

```
pip install -U setuptools
pip install -U pip
pip install -U autowrap
pip install -U nose
pip install -U numpy
pip install -U wheel
```

Depending on your systems setup, it may make sense to do this inside a virtual environment

```
virtualenv pyopenms_venv
source pyopenms_venv/bin/activate
```

Next, configure OpenMS with pyOpenMS: execute cmake as usual, but with parameters `DPYOPENMS=ON`. Also, if using virtualenv or using a specific Python version, add `-DPYTHON_EXECUTABLE:FILEPATH=/path/to/python` to ensure that the correct Python executable is used. Compiling pyOpenMS can use a lot of memory and take some time, however you can reduce the memory consumption by breaking up the compilation into multiple units and compiling in parallel, for example `-DPY_NUM_THREADS=2 -DPY_NUM_MODULES=4` will build 4 modules with 2 threads. You can then configure pyOpenMS:

```
cmake -DPYOPENMS=ON
make pyopenms
```

Build pyOpenMS (now there should be pyOpenMS specific build targets). Afterwards, test that all wen well by running the tests:

```
ctest -R pyopenms
```

Which should execute all the tests and return with all tests passing.

15.1 Further questions

In case the above instructions did not work, please refer to the Wiki Page, contact the development team on github or send an email to the OpenMS mailing list.

CHAPTER 16

Wrapping Workflow and wrapping new Classes

16.1 How pyOpenMS wraps Python classes

General concept of how the wrapping is done (all files are in `src/pyOpenMS/`):

- Step 1: The author declares which classes and which functions of these classes s/he wants to wrap (expose to Python). This is done by writing the function declaration in a file in the `pxds/` folder.
- Step 2: The Python tool "autowrap" (developed for this project) creates the wrapping code automatically from the function declaration - see https://github.com/uweschmitt/autowrap for an explanation of the autowrap tool. Since not all code can be wrapped automatically, also manual code can be written in the `addons/` folder. Autowrap will create an output file at `pyopenms/pyopenms.pyx` which can be interpreted by Cython.
- Step 3: Cython translates the `pyopenms/pyopenms.pyx` to C++ code at `pyopenms/pyopenms.cpp`
- Step 4: A compiler compiles the C++ code to a Python module which is then importable in Python with `import pyopenms`

Maintaining existing wrappers: If the C++ API is changed, then pyOpenMS will not build any more. Thus, find the corresponding file in the `pyOpenMS/pxds/` folder and adjust the function declaration accordingly.

16.2 How to wrap new methods in existing classes

Lets say you have written a new method for an existing OpenMS class and you would like to expose this method to pyOpenMS. First, identify the correct `.pxd` file in the `src/pyOpenMS/pxds` folder (for example for `Adduct` that would be Adduct.pxd). Open it and add your new function *with the correct indentation*:

- Place the full function declaration into the file (indented as the other functions)
- Check whether you are using any classes that are not yet imported, if so add a corresponding `cimport` statement to the top of the file. E.g. if your method is using using `MSExperiment`, then add `from MSExerpiment cimport *` to the top (note its cimport, not import).
- Remove any qualifiers (e.g. *const*) from the function signature and add *nogil except* + to the end of the signature

  - Ex: `void setType(Int a);` becomes `void setType(Int a) nogil except +`
  - Ex: `const T& getType() const;` becomes `T getType() nogil except +`
- Remove any qualifiers (e.g. *const*) from the argument signatures, but leave reference and pointer indicators
  - Ex: `const T&` becomes `T`, preventing an additional copy operation
  - Ex: `T&` will stay `T&` (indicating `T` needs to copied back to Python)
  - Ex: `T*` will stay `T*` (indicating `T` needs to copied back to Python)
  - One exception is `OpenMS::String`, you can leave `const String&` as-is
- STL constructs are replaced with Cython constructs: `std::vector<X>` becomes `libcpp_vector[ X ]` etc.
- Most complex STL constructs can be wrapped even if they are nested, however mixing them with user-defined types does not always work, see *Limitations* below. Nested `std::vector` constructs work well even with user-defined (OpenMS-defined) types. However, `std::map<String, X>` does not work (since `String` is user-defined, however a primitive C++ type such as `std::map<std::string, X>` would work).
- Python cannot pass primitive data types by reference (therefore no `int& res1`)
- Replace `boost::shared_ptr<X>` with `shared_ptr[X]` and add `from smart_ptr cimport shared_ptr` to the top
- Public members are simply added with `Type member_name`
- You can inject documentation that will be shown when calling `help()` in the function by adding `wrap-doc:Your documentation` as a comment after the function:
  - Ex: `void modifyWidget() nogil except + #wrap-doc:This changes your widget`

See the next section for a *SimpleExample* and a more *AdvancedExample* of a wrapped class with several functions.

16.3 How to wrap new classes

16.3.1 A simple example

To wrap a new OpenMS class: Create a new ".pxd" file in the folder `./pxds`. As a small example, look at the Adduct.pxd to get you started. Start with the following structure:

```
from xxx cimport *
cdef extern from "<OpenMS/path/to/header/Classname.h>" namespace "OpenMS":

    cdef cppclass ClassName(DefaultParamHandler):
        # wrap-inherits:
        #    DefaultParamHandler

        ClassName() nogil except +
        ClassName(ClassName) nogil except +

        Int getValue() nogil except + #wrap-doc:Gets value (between 0 and 5)
        void setValue(Int v) nogil except + #wrap-doc:Sets value (between 0 and 5)
```

- make sure to use `ClassName:` instead of `ClassName(DefaultParamHandler):` to wrap a class that does not inherit from another class and also remove the two comments regarding inheritance below that line.
- always use `cimport` and not Python `import`

- always add default constructor AND copy constructor to the code (note that the C++ compiler will add a default copy constructor to any class)
- to expose a function to Python, copy the signature to your pxd file, e.g. `DataValue getValue()` and make sure you `cimport` all corresponding classes. Replace `std::vector` with the corresponding Cython vector, in this case `libcpp_vector` (see for example PepXMLFile.pxd)
- Remember to include a copy constructor (even if none was declared in the C++ header file) since Cython will need it for certain operations. Otherwise you might see error messages like `item2.inst = shared_ptr[_ClassName](new _ClassName(deref(it_terms))) Call with wrong number of arguments`.
- you can add documentation that will show up in the interactive Python documentation (using `help()`) using the `wreap-doc` qualifier

16.3.2 A further example

A slightly more complicated class could look like this, where we demonstrate how to handle a templated class with template `T` and static methods:

```
from xxx cimport *
from AbstractBaseClass cimport *
from AbstractBaseClassImpl1 cimport *
from AbstractBaseClassImpl2 cimport *
cdef extern from "<OpenMS/path/to/header/Classname.h>" namespace "OpenMS":

    cdef cppclass ClassName[T](DefaultParamHandler):
        # wrap-inherits:
        #    DefaultParamHandler
        #
        # wrap-instances:
        #   ClassName := ClassName[X]
        #   ClassNameY := ClassName[Y]

        ClassName() nogil except +
        ClassName(ClassName[T]) nogil except + # wrap-ignore

        void method_name(int param1, double param2) nogil except +
        T method_returns_template_param() nogil except +

        size_t size() nogil except +
        T operator[](int) nogil except + # wrap-upper-limit:size()

        libcpp_vector[T].iterator begin() nogil except +  # wrap-iter-begin:__iter__
↪(T)
        libcpp_vector[T].iterator end()   nogil except +  # wrap-iter-end:__iter__(T)

        void getWidgets(libcpp_vector[String] & keys) nogil except +
        void getWidgets(libcpp_vector[unsigned int] & keys) nogil except + # wrap-
↪as:getWAsInt

        # C++ signature: void process(AbstractBaseClass * widget)
        void process(AbstractBaseClassImpl1 * widget) nogil except +
        void process(AbstractBaseClassImpl2 * widget) nogil except +

cdef extern from "<OpenMS/path/to/header/Classname.h>" namespace "OpenMS::Classname
↪<OpenMS::X>":
```

(continues on next page)

(continued from previous page)

```
    void static_method_name(int param1, double param2) nogil except + # wrap-
↪attach:ClassName

cdef extern from "<OpenMS/path/to/header/Classname.h>" namespace "OpenMS::Classname
↪<OpenMS::Y>":

    void static_method_name(int param1, double param2) nogil except + # wrap-
↪attach:ClassNameY
```

Here the copy constructor will not be wrapped but the Cython parser will import it from C++ so that is is present (using `wrap-ignore`). The `operator[]` will return an object of type `X` or `Y` depending on the template argument `T` and contain a guard that the number may not be exceed `size()`.

The wrapping of iterators allows for iteration over the objects inside the `Classname` container using the appropriate Python function (here `__iter__` with the indicated return type `T`).

The `wrap-as` keyword allows the Python function to assume a different name.

Note that pointers to abstract base classes can be passed as arguments but the classes have to be known at compile time, e.g. the function `process` takes a pointer to `AbstractBaseClass` which has two known implementations `AbstractBaseClassImpl1` and `AbstractBaseClassImpl2`. Then, the function needs to declared and overloaded with both implementations as arguments as shown above.

16.3.3 An example with handwritten addon code

A more complex examples requires some hand-written wrapper code (`pxds/Classname.pxd`), for example for singletons that implement a `getInstance()` method that returns a pointer to the singleton resource. Note that in this case it is quite important to not let autowrap take over the pointer and possibly delete it when the lifetime of the Python object ends. This is done through `wrap-manual-memory` and failing to doing so could lead to segmentation faults in the program.

```
from xxx cimport *
cdef extern from "<OpenMS/path/to/header/Classname.h>" namespace "OpenMS":

    cdef cppclass ModificationsDB "OpenMS::ModificationsDB":
        # wrap-manual-memory
        # wrap-hash:
        #   getFullId().c_str()

        ClassName(ClassName[T]) nogil except + # wrap-ignore

        void method_name(int param1, double param2) nogil except +

        int process(libcpp_vector[Peak1D].iterator, libcpp_vector[Peak1D].iterator)␣
↪nogil except + # wrap-ignore

cdef extern from "<OpenMS/path/to/header/Classname.h>" namespace "OpenMS::Classname":

    const ClassName* getInstance() nogil except + # wrap-ignore
```

Here the `wrap-manual-memory` keyword indicates that memory management will be handled manually and autowrap can assume that a member called `inst` will be provided which implements a `gets()` method to obtain a pointer to an object of C++ type `Classname`.

We then have to provide such an object (`addons/Classname.pyx`):

```
# This will go into the header (no empty lines below is *required*)
# NOTE: _Classname is the C++ class while Classname is the Python class
from Classname cimport Classname as _Classname
cdef class ClassnameWrapper:
    # A small utility class holding a ptr and implementing get()
    cdef const _Classname* wrapped
    cdef setptr(self, const _Classname* wrapped): self.wrapped = wrapped
    cdef const _Classname* get(self) except *: return self.wrapped

    # This will go into the class (after the first empty line)
    # NOTE: we use 4 spaces indent
    # NOTE: using shared_ptr for a singleton will lead to segfaults, use raw ptr␣
↪instead
    cdef ClassnameWrapper inst

    def __init__(self):
      self.inst = ClassnameWrapper()
      # the following require some knowledge of the internals of autowrap:
      # we call the getInstance method to obtain raw ptr
      self.inst.setptr(_getInstance_Classname())

    def __dealloc__(self):
      # Careful here, the wrapped ptr is a single instance and we should not
      # reset it (which is why we used 'wrap-manual-dealloc')
      pass

    def process(self, Container c):
      # An example function here (processing Container c):
      return self.inst.get().process(c.inst.get().begin(), c.inst.get().end())
```

Note how the manual wrapping of the process functions allows us to access the `inst` pointer of the argument as well as of the object itself, allowing us to call C++ functions on both pointers. This makes it easy to generate the required iterators and process the container efficiently.

16.3.4 Considerations and limitations

Further considerations and limitations:

- Inheritance: there are some limitations, see for example `Precursor.pxd`
- Reference: arguments by reference may be copied under some circumstances. For example, if they are in an array then not the original argument is handed back, so comparisons might fail. Also, simple Python types like int, float etc cannot be passed by reference.
- operator+=: see for example `AASequence.iadd` in `AASequence.pxd`
- operator==, !=, <=, <, >=, > are wrapped automatically
- Iterators: some limitations apply, see MSExperiment.pxd for an example
- copy-constructor becomes __copy__ in Python
- shared pointers: is handled automatically, check DataAccessHelper using `shared_ptr[Spectrum]`. Use `from smart_ptr cimport shared_ptr` as import statement

These hints can be given to autowrap classes (also check the autowrap documentation):

- `wrap-ignore` is a hint for autowrap to not wrap the class (but the declaration might still be important for Cython to know about)

- `wrap-instances`: for templated classes (see MSSpectrum.pxd)
- `wrap-hash`: hash function to use for `__hash__` (see Residue.pxd)
- `wrap-manual-memory`: hint that memory management will be done manually

These hints can be given to autowrap functions (also check the autowrap documentation):

- `wrap-ignore` is a hint for autowrap to not wrap the function (but the declaration might still be important for Cython to know about)
- `wrap-as`: see for example AASequence
- `wrap-iter-begin:`, `wrap-iter-end`: (see ConsensusMap.pxd)
- `wrap-attach`: enums, static methods (see for example VersionInfo.pxd)
- `wrap-upper-limit:size()` (see MSSpectrum.pxd)

16.3.5 Wrapping code yourself in ./addons

Not all code can be wrapped automatically (yet). Place a file with the same (!) name in the addons folder (e.g. `myClass.pxd` in `pxds/` and `myClass.pyx` in `addons/`) and leave two lines empty on the top (this is important). Start with 4 spaces of indent and write your additional wrapper functions, adding a wrap-ignore comment to the pxd file. See the example above, some additional examples, look into the `src/pyOpenMS/addons/` folder:

- IDRipper.pyx
  - for an example of both input and output of a complex STL construct (`map< String, pair<vector<>, vector<> >`)
- MSQuantifications.pyx
  - for a `vector< vector< pair <String,double > > >` as input in registerExperiment
  - for a `map< String, Ratio>` in getRatios to get returned
- QcMLFile.pyx - for a `map< String, map< String,String> >` as input
- SequestInfile.pyx
  - for a `map< String, vector<String> >` to get returned
- Attachment.pyx
  - for a `vector< vector<String> >` to get returned
- ChromatogramExtractorAlgorithm.pxd
  - for an example of an abstract base class (`ISpectrumAccess`) in the function `extractChromatograms` - this is solved by copy-pasting the function multiple times for each possible implementation of the abstract base class.

Make sure that you *always* declare your objects (all C++ and all Cython objects need to be declared) using `cdef` Type name. Otherwise you get `Cannot convert ... to Python object` errors.

CHAPTER 17

Indices and tables

- genindex

Index