# Table of Contents

7.2 Modified Sequences ........................................... 39
7.3 Proteins ......................................................... 40

8 Oligonucleotides: RNA ........................................ 43
8.1 Nucleic Acid Sequences ..................................... 43
8.2 Modified oligonucleotides .................................. 44
8.3 DNA, RNA and Protein ...................................... 45

9 TheoreticalSpectrumGenerator .............................. 47
9.1 Y-ion spectrum ................................................. 47
9.2 Full fragment ion spectrum .................................. 48
9.3 Visualization .................................................... 49

10 Digestion ........................................................ 51
10.1 Proteolytic Digestion with Trypsin ..................... 51
10.2 Proteolytic Digestion with Lys-C ....................... 51
10.3 Oligonucleotide Digestion ................................. 52

11 Identification Data ............................................ 53
11.1 ProteinIdentification ....................................... 53
11.2 PeptideIdentification ...................................... 54
11.3 Storage on disk ............................................... 55

12 Quantitative Data ................................................ 57
12.1 Feature .......................................................... 57
12.2 FeatureMap ...................................................... 58
12.3 ConsensusFeature .......................................... 58
12.4 ConsensusMap ................................................. 60

13 Simple Data Manipulation .................................... 61
13.1 Filtering Spectra .............................................. 61
13.2 Filtering Spectra and Peaks ............................. 62
13.3 Memory management ....................................... 63

14 Parameter Handling ........................................... 65

15 Algorithms ....................................................... 67

16 Smoothing ........................................................ 69

17 Mass Decomposition .......................................... 71
17.1 Fragment mass to amino acid composition ............ 71
17.2 Naive algorithm .............................................. 72
17.3 Stand-alone Program ....................................... 72
17.4 Spectrum Tagger ............................................. 73

18 Charge and Isotope Deconvolution ....................... 75
18.1 Single peak example ........................................ 75
18.2 Full spectral de-isotoping ............................... 76
18.3 Visualization .................................................. 77

19 Feature Detection ............................................... 81

20 Peptide Search .................................................. 83
20.1 SimpleSearch ................................................ 83
20.2 PSM inspection .............................................. 84
pyOpenMS is an open-source Python library for mass spectrometry, specifically for the analysis of proteomics and metabolomics data in Python. pyOpenMS implements a set of Python bindings to the OpenMS library for computational mass spectrometry and is available for Windows, Linux and OSX.

PyOpenMS provides functionality that is commonly used in computational mass spectrometry. 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 facile access to a feature-rich, open-source algorithm library for mass-spectrometry based proteomics analysis.

pyOpenMS facilitates the execution of common tasks in proteomics (and other mass spectrometric fields) such as

- file handling (mzXML, mzML, TraML, mzTab, fasta, pepxml, protxml, mzIdentML among others)
- chemistry (mass calculation, peptide fragmentation, isotopic abundances)
- signal processing (smoothing, filtering, de-isotoping, retention time correction and peak-picking)
- identification analysis (including peptide search, PTM analysis, Cross-linked analytes, FDR control, RNA oligonucleotide search and small molecule search tools)
- quantitative analysis (including label-free, metabolomics, SILAC, iTRAQ and SWATH/DIA analysis tools)
- chromatogram analysis (chromatographic peak picking, smoothing, elution profiles and peak scoring for SRM/MRM/PRM/SWATH/DIA data)
- interaction with common tools in proteomics and metabolomics
  - search engines such as Comet, Crux, Mascot, MSGFPlus, MSFragger, Myrimatch, OMSSA, Sequest, SpectraST, XTandem
  - post-processing tools such as percolator, MSStats, Fido
  - metabolomics tools such as SIRIUS, CSI:FingerId

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

Note: the current documentation relates to the 2.6.0 release of pyOpenMS.
1.1 Binaries

1.1.1 Spyder

On Microsoft Windows, we recommend to use pyopenms together with Anaconda and the Spyder interface which you can download from the Official Anaconda repository. After installation, select the “Anaconda Powershell Prompt” from the start menu and enter the following command:

```
pip install pyopenms
```

which should result in the following output:
Once successfully installed, you can open the “Spyder” graphical user interface and pyopenms will be available:

Note the console window (lower right) with the `import pyopenms` command, which was executed without error. Next, the Python script on the right was executed and the output is also shown on the console window. You can now use pyopenms within the Spyder environment, either moving on with the pyOpenMS Tutorial or familiarize yourself first with the Spyder environment using the Online Spyder Documentation.
## 1.1.2 Command Line

To install pyOpenMS from the command line using the binary wheels, you can type

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

We have binary packages for OSX, Linux and Windows (64 bit only) available from PyPI. Note that for Windows, we only support Python 3.5, 3.6 and 3.7 in their 64 bit versions, therefore make sure to download the 64bit Python release for Windows. For OSX and Linux, we additionally 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:\Python37\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. Note that this may be non-trivial and is not recommended for most users.

## 1.3 Wrap Classes

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

```
from pyopenms import *
exp = MSExperiment()
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:

```
>>> from pyopenms import *
>>> help(MSExperiment)
```

`class MSExperiment(builtins.object)`

<p>| Cython implementation of <code>MSExperiment</code> |
| -- Inherits from ['ExperimentalSettings', 'RangeManager2'] |</p>
<table>
<thead>
<tr>
<th>In-Memory representation of a mass spectrometry experiment.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contains the data and metadata of an experiment performed with an MS (or HPLC and MS). This representation of an MS experiment is organized as list</td>
</tr>
</tbody>
</table>

(continues on next page)
of spectra and chromatograms and provides an in-memory representation of
popular mass-spectrometric file formats such as mzXML or mzML. The
meta-data associated with an experiment is contained in
ExperimentalSettings (by inheritance) while the raw data (as well as
spectra and chromatogram level meta data) is stored in objects of type
MS Spectrum and MS Chromatogram, which are accessible through the get Spectrum
and get Chromatogram functions.

---

Spectra can be accessed by direct iteration or by get Spectrum(),
while chromatograms are accessed through get Chromatogram().
See help (ExperimentalSettings) for information about meta-data.

Methods defined here:

[...]

which lists information on the pyopenms.MSExperiment class, including a description of the main purpose of the
class and how the class is intended to be used. Additional useful information is presented in the Inherits from
section which points to additional classes that act as base classes to pyopenms.MSExperiment and that contain
further information. The list of available methods is long (but does not include methods from the base classes) and
reveals that the class exposes methods such as get Nr Spectra() and get Spectrum(id) where the argument
id indicates the spectrum identifier. The command also lists the signature for each function, allowing users to identify
the function arguments and return types. We can gain further information about exposed methods by investigating the
documentation of the base classes:

>>> from pyopenms import *
>>> help(ExperimentalSettings)
Help on class ExperimentalSettings in module pyopenms.pyopenms_4:

class ExperimentalSettings(builtins.object)
   | Cython implementation of _ExperimentalSettings
   | -- Inherits from ['DocumentIdentifier', 'MetaInfoInterface']
   |
   | Description of the experimental settings, provides meta-information
   | about an LC-MS/MS injection.
   |
   | Methods defined here:
   |
[...]

We could now continue our investigation by reading the documentation of the base classes DocumentIdentifier
and MetaInfoInterface, but we will leave this exercise for the interested reader. 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 docu-
mentation, 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 spec-
trometric data, we can download the mzML example file
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
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:

```python
>>> help(exp)
```

class MSExperiment(builtins.object):
    | Cython implementation of _MSExperiment
    | -- Inherits from ['ExperimentalSettings', 'RangeManager2']

    [...
    |
    | 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:

```python
>>> exp.getNrSpectra()
4
```  
```python
>>> 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

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

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():

```python
>>> spec = exp[1]
>>> mz, intensity = spec.get_peaks()
>>> sum(intensity)
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):

```python
>>> 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

Here, we will apply what we have learned to perform a common function in mass spectrometry: calculating the total ion current. While some mzML files already contain a pre-computed TIC, we will show you how to calculate the TIC for MS1 and MS2 spectra separately (generally only MS1 TIC is provided).

With this information, we can write a function that calculates the total ion current (TIC) for a given ms level:

```python
# Calculates total ion chromatogram of an LC-MS/MS experiment
def calcTIC(exp, mslevel):
    tic = 0
    # Iterate through all spectra of the experiment
    for spec in exp:
        # Only calculate TIC for matching (MS1) spectra
        if spec.getMSLevel() == mslevel:
            mz, i = spec.get_peaks()
            tic += sum(i)
    return tic
```

To calculate a TIC we would now call the function:

```python
>>> calcTIC(exp, 1)
240.0
>>> sum([sum(s.get_peaks()[1]) for s in exp if s.getMSLevel() == 1])
240.0
>>> calcTIC(exp, 2)
110.0
```
Note how one can compute the same property using list comprehensions in Python (see line number 3 in the above code which computes the TIC using filtering properties of Python list comprehensions `s.getMSLevel() == 1`) and computes the sum over all peaks (right `sum`) and the sum over all spectra (left `sum`) to retrieve the TIC.
3.1 mzML files in memory

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

```python
from pyopenms import *
from urllib.request import urlretrieve

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:

```python
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:

```python
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:

```python
MzMLFile().store("filtered.mzML", exp)
```
Putting this together, a small filtering program would look like this:

```python
'''
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 indexed mzML files

Since pyOpenMS 2.4, you can open, read and inspect files that use the indexedMzML standard. This allows users to read MS data without loading all data into memory:

```python
from pyopenms import *
od_exp = OnDiscMSExperiment()
od_exp.openFile("test.mzML")
meta_data = od_exp.getMetaData()
meta_data.getNrChromatograms()# data is not present in meta_data experiment
sum(meta_data.getChromatogram(0).get_peaks()[1]) # no data!
sum(od_exp.getChromatogram(0).get_peaks()[1]) # data is here!

# meta data is present and identical in both data structures:
meta_data.getChromatogram(0).getNativeID() # fast
od_exp.getChromatogram(0).getNativeID() # slow
```

Note that the OnDiscMSExperiment allows users to access meta data through the getMetaData function, which allows easy selection and filtering on meta data attributes (such as MS level, precursor m/z, retention time etc.) in order to select spectra and chromatograms for analysis. Only once selection on the meta data has been performed, will actual data be loaded into memory using the getChromatogram and getSpectrum functions.

This approach is memory efficient in cases where computation should only occur on part of the data or the whole data may not fit into memory.

### 3.3 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")

text which can be used as follows:

>>> from pyopenms import *
>>> filename = b"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:

from pyopenms import *
class FilteringConsumer():
    
    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)

#-----------------------

(continues on next page)
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).

Note that this approach is memory efficient in cases where computation should only occur on part of the data or the whole data may not fit into memory.

### 3.4 cached mzML files

In addition, since pyOpenMS 2.4 the user can efficiently cache mzML files to disk which provides very fast access with minimal overhead in memory. Basically the data directly mapped into memory when requested. You can use this feature as follows:

```python
from pyopenms import *

# First load data and cache to disk
exp = MSExperiment()
MzMLFile().load("test.mzML", exp)
CachedmzML.store("myCache.mzML", exp)

# Now load data
cfile = CachedmzML()
CachedmzML.load("myCache.mzML", cfile)

meta_data = cfile.getMetaData()
cfile.getNrChromatograms()
cfile.getNrSpectra()

# data is not present in meta_data experiment.
sum(meta_data.getChromatogram(0).get_peaks()[1]) # no data!
sum(cfile.getChromatogram(0).get_peaks()[1]) # data is here!

# meta data is present and identical in both data structures:
meta_data.getChromatogram(0).getNativeID() # fast
cfile.getChromatogram(0).getNativeID() # slow
```

Note that the **CachedmzML** allows users to access meta data through the `getMetaData` function, which allows easy selection and filtering on meta data attributes (such as MS level, precursor m/z, retention time etc.) in order to select spectra and chromatograms for analysis. Only once selection on the meta data has been performed, will actual data be loaded into memory using the `getChromatogram` and `getSpectrum` functions.

Note that in the example above all data is loaded into memory first and then cached to disk. This is not very efficient and we can use the **MSDataCachedConsumer** to directly cache to disk (without loading any data into memory):
from pyopenms import *

# First cache to disk
# Note: writing meta data to myCache2.mzML is required
cacher = MSDataCachedConsumer("myCache2.mzML.cached")
exp = MSExperiment()
MzMLFile().transform(b"test.mzML", cacher, exp)
CachedMzMLHandler().writeMetadata(exp, "myCache2.mzML")

del cacher

# Now load data
cfile = CachedMzML()
CachedMzML.load("myCache2.mzML", cfile)

meta_data = cfile.getMetadata()
# data is not present in meta_data experiment
sum(meta_data.getChromatogram(0).get_peaks()[1]) # no data!
sum(cfile.getChromatogram(0).get_peaks()[1]) # data is here!

This approach is now memory efficient in cases where computation should only occur on part of the data or the whole data may not fit into memory.
4.1 Identification data (idXML, mzIdentML, pepXML, protXML)

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

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
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:

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
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:

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
```
You can load (storing is not supported) identification data from a TPP *protXML* file as follows:

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
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 Quantitative data (*featureXML, consensusXML*)

OpenMS stores quantitative information in the internal *featureXML* and *consensusXML* data formats. The *featureXML* format is used to store quantitative data from a single LC-MS/MS run while the *consensusXML* is used to store quantitative data from multiple LC-MS/MS runs. These can be accessed as follows:

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
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*

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urlretrieve (gh + "/src/tests/class_tests/openms/data/ConsensusXMLFile_1.consensusXML",
    "/test.consensusXML")
consensus_features = ConsensusMap()
ConsensusXMLFile().load("test.consensusXML", consensus_features)
ConsensusXMLFile().store("test.out.consensusXML", consensus_features)
```
4.3 Transition data (TraML)

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

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve  # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
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)
```
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:

```python
from pyopenms import *
spectrum = MSSpectrum()
mz = range(1500, 500, -100)
i = [0 for mass in mz]
spectrum.set_peaks([mz, i])

# Sort the peaks according to ascending mass-to-charge ratio
spectrum.sortByPosition()

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

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

# Access a peak by index
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.

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

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

```python
from pyopenms import *
spectrum = MSSpectrum()
spectrum.setDriftTime(25) # 25 ms
spectrum.setRT(205.2) # 205.2 s
spectrum.setMSLevel(3) # MS3
p = Precursor()
p.setIsolationWindowLowerOffset(1.5)
p.setIsolationWindowUpperOffset(1.5)
p.setMZ(600) # isolation at 600 +/- 1.5 Th
p.setActivationEnergy(40) # 40 eV
p.setCharge(4) # 4+ ion
spectrum.setPrecursors([p])

# Add raw data to spectrum
spectrum.set_peaks(((401.5), [900]))

# Additional data arrays / peak annotations
fda = FloatDataArray()
fda.setName("Signal to Noise Array")
fda.push_back(15)
sda = StringDataArray()
sda.setName("Peak annotation")
sda.push_back("y15++")
spectrum.setFloatDataArrays([fda])
spectrum.setStringDataArrays([sda])

# Add spectrum to MSExperiment
exp = MSExperiment()
exp.addSpectrum(spectrum)

# Add second spectrum and store as mzML file
spectrum2 = MSSpectrum()
spectrum2.set_peaks(((1, 2), [1, 2]))
exp.addSpectrum(spectrum2)
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:

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) )
```
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:

```python
>>> # 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
```
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)
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:

Note how the annotation using precursor and production mass of our XIC chromatogram is displayed in the viewer.
OpenMS has representations for various chemical concepts including molecular formulas, isotopes, ribonucleotide and amino acid sequences as well as common modifications of amino acids or ribonucleotides.

6.1 Constants

OpenMS has many chemical and physical constants built in:

```python
>>> import pyopenms

>>> help(pyopenms.Constants)

>>> print("Avogadro's number is", pyopenms.Constants.AVOGADRO)
```

which provides access to constants such as Avogadro’s number or the electron mass.

6.2 Elements

In OpenMS, elements are stored in ElementDB which has entries for dozens of elements commonly used in mass spectrometry. The database is stored in the pyopenms/share/OpenMS/CHEMISTRY/Elements.xml file (where a user can add new elements if necessary).

```python
from pyopenms import *

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

oxygen =edb.getElement("O")
print(oxygen.getName())
print(oxygen.getSymbol())
print(oxygen.getMonoWeight())
```

(continues on next page)
print(oxygen.getAverageWeight())

sulfur = edb.getElement("S")
print(sulfur.getName())
print(sulfur.getSymbol())
print(sulfur.getMonoWeight())
print(sulfur.getAverageWeight())
isotopes = sulfur.getIsotopeDistribution()

print("One mole of oxygen weighs", 2*oxygen.getAverageWeight(), "grams")
print("One mole of 16O2 weighs", 2*oxygen.getMonoWeight(), "grams")

As we can see, the OpenMS ElementDB has entries for common elements like Oxygen and Sulfur as well as information on their average and monoisotopic weight. Note that the monoisotopic weight is the weight of the most abundant isotope while the average weight is the sum across all isotopes, weighted by their natural abundance. Therefore, one mole of oxygen (O2) weighs slightly more than a mole of only its monoisotopic isotope since natural oxygen is a mixture of multiple isotopes.

Oxygen
O
15.994915
15.999405323160001
Sulfur
S
31.97207073
32.066084735289
One mole of oxygen weighs 31.998810646320003 grams
One mole of 16O2 weighs 31.98983 grams

6.2.1 Isotopes

We can also inspect the full isotopic distribution of oxygen and sulfur:

```python
from pyopenms import *
edb = ElementDB()
oxygen = edb.getElement("O")
isotopes = oxygen.getIsotopeDistribution()
for iso in isotopes.getContainer():
    print ("Oxygen isotope", iso.getMZ(), "has abundance", iso.getIntensity()*100, "/")
sulfur = edb.getElement("S")
isotopes = sulfur.getIsotopeDistribution()
for iso in isotopes.getContainer():
    print ("Sulfur isotope", iso.getMZ(), "has abundance", iso.getIntensity()*100, "/")
```

OpenMS can compute isotopic distributions for individual elements which contain information for all stable elements. The current values in the file are average abundances found in nature, which may differ depending on location. The above code outputs the isotopes of oxygen and sulfur as well as their abundance:

Oxygen isotope 15.994915 has abundance 99.75699782371521 %
Oxygen isotope 16.999132 has abundance 0.03800000122282654 %

(continues on next page)
Oxygen isotope 17.999169 has abundance 0.20500000100582838 %
Sulfur isotope 31.97207073 has abundance 94.92999911308289 %
Sulfur isotope 32.971458 has abundance 0.7600000128149986 %
Sulfur isotope 33.967867 has abundance 4.2899999767541885 %
Sulfur isotope 35.967081 has abundance 0.019999999494757503 %

6.2.2 Mass Defect

Note: While all isotopes are created by adding one or more neutrons to the nucleus, this leads to different observed masses due to the mass defect, which describes the difference between the mass of an atom and the mass of its constituent particles. For example, the mass difference between 12C and 13C is slightly different than the mass difference between 14N and 15N, even though both only differ by a neutron from their monoisotopic element:

```python
from pyopenms import *
edb = ElementDB()
isotopes = edb.getElement("C").getIsotopeDistribution().getContainer()
carbon_isotope_difference = isotopes[1].getMZ() - isotopes[0].getMZ()
isotopes = edb.getElement("N").getIsotopeDistribution().getContainer()
nitrogen_isotope_difference = isotopes[1].getMZ() - isotopes[0].getMZ()

def mass_difference(element, isotopes):
    mass_sum = 2*PROTON_MASS_U + 2*ELECTRON_MASS_U + 2*NEUTRON_MASS_U
    return 100*(mass_sum - isotopes[1].getMZ())/mass_sum

print("Mass difference between 12C and 13C: ", mass_difference("C", carbon_isotope_difference))
print("Mass difference between 14N and 15N: ", mass_difference("N", nitrogen_isotope_difference))
print("Relative deviation: ", 100*(carbon_isotope_difference - nitrogen_isotope_difference)/carbon_isotope_difference, "\%")
```

Mass difference between 12C and 13C: 1.003355
Mass difference between 14N and 15N: 0.997035
Relative deviation: 0.6298867300208343 %

This difference can actually be measured by a high resolution mass spectrometric instrument and is used in the tandem mass tag (TMT) labelling strategy.

For the same reason, the helium atom has a slightly lower mass than the mass of its constituent particles (two protons, two neutrons and two electrons):

```python
from pyopenms import *
from pyopenms.Constants import *
helium = ElementDB().getElement("He")
isotopes = helium.getIsotopeDistribution()
mass_sum = 2*PROTON_MASS_U + 2*ELECTRON_MASS_U + 2*NEUTRON_MASS_U
helium4 = isotopes.getContainer()[1].getMZ()

def mass_difference(element, isotopes):
    mass_sum = 2*PROTON_MASS_U + 2*ELECTRON_MASS_U + 2*NEUTRON_MASS_U
    return 100*(mass_sum - helium4)/mass_sum

print("Sum of masses of 2 protons, neutrons and electrons: ", mass_difference("He", helium4))
print("Mass of He4: ", helium4)
print("Difference between the two masses: ", 100*(mass_sum - helium4)/mass_sum, "\%")
```

Sum of masses of 2 protons, neutrons and electrons: 4.032979924670597
Mass of He4: 4.00260325415
Difference between the two masses: 0.7532065888743016 %

The difference in mass is the energy released when the atom was formed (or in other words, it is the energy required
to disassemble the nucleus into its particles).

### 6.3 Molecular Formulae

Elements can be combined to molecular formulas (EmpiricalFormula) which can be used to describe molecules such as metabolites, amino acid sequences or oligonucleotides. The class supports a large number of operations like addition and subtraction. A simple example is given in the next few lines of code.

```python
from pyopenms import *

methanol = EmpiricalFormula("CH3OH")
water = EmpiricalFormula("H2O")
ethanol = EmpiricalFormula("CH2") + methanol

print("Ethanol chemical formula: ", ethanol.toString())
print("Ethanol composition: ", ethanol.getElementalComposition())
print("Ethanol has ", ethanol.getElementalComposition()[b"H"], "hydrogen atoms")
```

which produces

Ethanol chemical formula: C2H6O1
Ethanol composition: {b'C': 2, b'H': 6, b'O': 1}
Ethanol has 6 hydrogen atoms

Note how in line 5 we were able to make a new molecule by adding existing molecules (for example by adding two EmpiricalFormula objects). In this case, we illustrated how to make ethanol by adding a CH2 methyl group to an existing methanol molecule. Note that OpenMS describes sum formulae with the EmpiricalFormula object and does store structural information in this class.

### 6.4 Isotopic Distributions

OpenMS can also generate theoretical isotopic distributions from analytes represented as EmpiricalFormula. Currently there are two algorithms implemented, CoarseIsotopePatternGenerator which produces unit mass isotope patterns and FineIsotopePatternGenerator which is based on the IsoSpec algorithm:

```python
from pyopenms import *

methanol = EmpiricalFormula("CH3OH")
ethanol = EmpiricalFormula("CH2") + methanol

print("Coarse Isotope Distribution:")
isotopes = ethanol.getIsotopeDistribution( CoarseIsotopePatternGenerator(4) )
prob_sum = sum([iso.getIntensity() for iso in isotopes.getContainer()])
print("This covers ", prob_sum, " probability")
for iso in isotopes.getContainer():
    print ("Isotope", iso.getMZ(), "has abundance", iso.getIntensity()*100, "%")

print("Fine Isotope Distribution:")
isotopes = ethanol.getIsotopeDistribution( FineIsotopePatternGenerator(1e-3) )
prob_sum = sum([iso.getIntensity() for iso in isotopes.getContainer()])
```

(continues on next page)

---

print("This covers", prob_sum, "probability")
for iso in isotopes.getContainer():
    print ("Isotope", iso.getMZ(), "has abundance", iso.getIntensity()*100, ")
which produces

Coarse Isotope Distribution:
This covers 0.999999753596569 probability
Isotope 46.0418651914 has abundance 97.56630063056946 %
Isotope 47.04522029199996 has abundance 2.21499539911747 %
Isotope 48.048574867 has abundance 0.2142168115824461 %
Isotope 49.0519297048 has abundance 0.004488634294830263 %

Fine Isotope Distribution:
This covers 0.9994461630121805 probability
Isotope 46.0418651914 has abundance 97.5662887096405 %
Isotope 47.0452201914 has abundance 2.11050106245613 %
Isotope 47.0481419395 has abundance 0.06732848123647273 %
Isotope 48.04611919139995 has abundance 0.20049810409545898 %

The result calculated with the FineIsotopePatternGenerator contains the hyperfine isotope structure with heavy isotopes of Carbon and Hydrogen clearly distinguished while the coarse (unit resolution) isotopic distribution contains summed probabilities for each isotopic peak without the hyperfine resolution.

Please refer to our previous discussion on the mass defect to understand the results of the hyperfine algorithm and why different elements produce slightly different masses. In this example, the hyperfine isotopic distribution will contain two peaks for the nominal mass of 47: one at 47.045 for the incorporation of one heavy 13C with a delta mass of 1.003355 and one at 47.048 for the incorporation of one heavy deuterium with a delta mass of 1.006277. These two peaks also have two different abundances (the heavy carbon one has 2.1% abundance and the deuterium one has 0.07% abundance). This can be understood given that there are 2 carbon atoms and the natural abundance of 13C is about 1.1%, while the molecule has six hydrogen atoms and the natural abundance of deuterium is about 0.02%. The fine isotopic generator will not generate the peak at nominal mass 49 since we specified our cutoff at 0.1% total abundance and the four peaks above cover 99.9% of the isotopic abundance.

We can also decrease our cutoff and ask for more isotopes to be calculated:

```
from pyopenms import *

methanol = EmpiricalFormula("CH3OH")
ethanol = EmpiricalFormula("CH2") + methanol

print("Fine Isotope Distribution:")
isotopes = ethanol.getIsotopeDistribution( FineIsotopePatternGenerator(1e-6) )
prob_sum = sum({iso.getIntensity() for iso in isotopes.getContainer()})
print("This covers", prob_sum, "probability")
for iso in isotopes.getContainer():
    print ("Isotope", iso.getMZ(), "has abundance", iso.getIntensity()*100, ")
```

which produces

Fine Isotope Distribution:
This covers 0.9999993089130612 probability
Isotope 46.0418651914 has abundance 97.5662887096405 %
Isotope 47.0452201914 has abundance 2.11050106245613 %
Isotope 47.046082191400004 has abundance 0.03716550418175757 %
Isotope 47.0481419395 has abundance 0.06732848123647273 %

(continues on next page)
Isotope 48.046119191399995 has abundance 0.20049810409545898 %
Isotope 48.0485751914 has abundance 0.011413302854634821 %
Isotope 48.0494371914 has abundance 0.0008039440217544325 %
Isotope 48.0514969395 has abundance 0.0014564131561201066 %
Isotope 49.049474191399995 has abundance 0.004337066275184043 %
Isotope 49.0523959395 has abundance 0.00013835959862262825 %

Here we can observe more peaks and now also see the heavy oxygen peak at 47.04608 with a delta mass of 1.004217 (difference between 16O and 17O) at an abundance of 0.04%, which is what we would expect for a single oxygen atom. Even though the natural abundance of deuterium (0.02%) is lower than 17O (0.04%), since there are six hydrogen atoms in the molecule and only one oxygen, it is more likely that we will see a deuterium peak than a heavy oxygen peak. Also, even for a small molecule like ethanol, the differences in mass between the hyperfine peaks can reach more than 110 ppm (48.046 vs 48.051). Note that the FinelIsotopePatternGenerator will generate peaks until the total error has decreased to 1e-6, allowing us to cover 0.999999 of the probability.

OpenMS can also produce isotopic distribution with masses rounded to the nearest integer:

```python
isotopes = ethanol.getIsotopeDistribution( CoarseIsotopePatternGenerator(5, True) )
for iso in isotopes.getContainer():
    print("Isotope", iso.getMZ(), "has abundance", iso.getIntensity()*100, ", ")
```

<table>
<thead>
<tr>
<th>Isotope</th>
<th>Abundance</th>
</tr>
</thead>
<tbody>
<tr>
<td>46.0</td>
<td>97.56627082824707 %</td>
</tr>
<tr>
<td>47.0</td>
<td>2.214994840323925 %</td>
</tr>
<tr>
<td>48.0</td>
<td>0.214216741733253 %</td>
</tr>
<tr>
<td>49.0</td>
<td>0.0044886332034366205 %</td>
</tr>
<tr>
<td>50.0</td>
<td>0.00013835959862262825 %</td>
</tr>
</tbody>
</table>

### 6.5 Amino Acids

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.

```python
>>> from pyopenms import *
>>> lys = ResidueDB().getResidue("Lysine")
>>> lys.getName()
'Lysine'
>>> lys.getThreeLetterCode()
'LYS'
>>> lys.getOneLetterCode()
'K'
>>> lys.getAverageWeight()
146.18788276708443
>>> lys.getMonoWeight()
146.1055284466
>>> lys.getPka()
2.16
>>> lys.getFormula().toString()
'\u0027C6H14N2O2\u2019'
```

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.
6.6 Amino Acid 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.

```python
from pyopenms import *
ox = ModificationsDB().getModification("Oxidation")
print(ox.getUniModAccession())
print(ox.getUniModRecordId())
print(ox.getDiffMonoMass())
print(ox.getId())
print(ox.getFullId())
print(ox.getFullName())
print(ox.getDiffFormula())
```

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):

```python
isotopes = ox.getDiffFormula().getIsotopeDistribution(CoarseIsotopePatternGenerator(5))
for iso in isotopes.getContainer():
    print (iso.getMZ(), ":", iso.getIntensity())
```

Which will print the isotopic pattern of the modification (Oxygen):

```
15.994915 : 0.9975699782371521
16.998269837800002 : 0.0003800000122282654
18.0016246756 : 0.002050000010058284
```

6.7 Ribonucleotides

A ribonucleotide describes one of the building blocks of DNA and RNA. In OpenMS, a ribonucleotide in its modified or unmodified form is represented by the `Ribonucleotide` class in OpenMS. 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. Modified ribonucleotides are represented by the same class. Currently, support for RNA is implemented.

```python
>>> from pyopenms import *
>>> uridine = RibonucleotideDB().getRibonucleotide(b"U")
>>> uridine.getName()
'uridine'
>>> uridine.getCode()
'U'
>>> uridine.getAvgMass()
(continues on next page)```
>>> uridine.getMonoMass()
244.0695
>>> uridine.getFormula().toString()
'C9H12N2O6'
>>> uridine.isModified()
False

>>> methyladenosine = RibonucleotideDB().getRibonucleotide(b"m1A")
>>> methyladenosine.getName()
'1-methyladenosine'
>>> methyladenosine.isModified()
True
CHAPTER 7

Peptides and Proteins

7.1 Amino Acid Sequences

The AASequence class handles amino acid sequences in OpenMS. A string of amino acid residues can be turned into an 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")
prefix = seq.getPrefix(4)
suffix = seq.getSuffix(5)
concat = seq + seq

print(seq)
print(concat)
print(suffix)
mfull = seq.getMonoWeight()  # weight of M
mprecursor = seq.getMonoWeight(Residue.ResidueType.Full, 2)  # weight of M+2H
mz = seq.getMonoWeight(Residue.ResidueType.Full, 2) / 2.0  # m/z of M+2H

print Московский mass of peptide [M] is", mfull)
print Московскый mass of peptide precursor [M+2H]^{2+} is", mprecursor)
print Московскый mass of m/z of [M+2H]^{2+} is", mz)
```

Which prints the amino acid sequence on line 7 as well as the result of concatenating two sequences or taking the suffix of a sequence (lines 8 and 9). On line 10 we compute the mass of the full peptide ([M]), on line 11 we compute the mass of the peptide precursor ([M+2H]^{2+}) while on line 12 we compute the m/z value of the peptide precursor.
The mass of the peptide precursor is shifted by two protons that are now attached to the molecules as charge carriers (note that the proton mass of 1.007276 u is slightly different from the mass of an uncharged hydrogen atom at 1.007825 u).

The AASequence object also allows iterations directly in Python:

```python
from pyopenms import *
seq = AASequence.fromString("DFPIANGER")
print("The peptide", str(seq), "consists of the following amino acids:")
for aa in seq:
  print(aa.getName(), ":", aa.getMonoWeight())
```

Which will print

The peptide DFPIANGER consists of the following amino acids:
Aspartate : 133.0375092233
Phenylalanine : 165.0789793509
Proline : 115.0633292871
Isoleucine : 131.0946294147
Alanine : 89.04767922330001
Asparagine : 132.0534932552
Glycine : 75.0320291595
Glutamate : 147.05315928710002
Arginine : 174.1116764466

### 7.1.1 Fragment ions

Note how we can easily calculate the charged weight of a \((\text{M+2H})^2+\) ion on line 11 and compute \(m/z\) on line 12 – simply dividing by the charge. 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
from pyopenms import *
seq = AASequence.fromString("DFPIANGER")
seq_formula = seq.getFormula()
print("Peptide", seq, "has molecular formula", seq_formula)
print("*
*
isotopes = seq_formula.getIsotopeDistribution( CoarseIsotopePatternGenerator(6) )
for iso in isotopes.getContainer():
  print ("Isotope", iso.getMZ(), "has abundance", iso.getIntensity()*100, ",\%")
suffix = seq.getSuffix(3) # y3 ion "GER"
print("*
*
y3_formula = suffix.getFormula(Residue.ResidueType.YIon, 2) # y3++ ion
suffix.getMonoWeight(Residue.ResidueType.YIon, 2) / 2.0 # CORRECT
suffix.getMonoWeight(Residue.ResidueType.XIon, 2) / 2.0 # CORRECT
```

(continues on next page)
17 suffix.getMonoWeight(Residue.ResidueType.BIon, 2) / 2.0 # INCORRECT
18 print("y3 mz:", suffix.getMonoWeight(Residue.ResidueType.YIon, 2) / 2.0 )
19 print("y3 molecular formula:", y3_formula)

Which will produce

Peptide DFPIANGER has molecular formula C44H67N13O15
===================================
Isotope 1017.4879641373 has abundance 56.81651830673218 %
Isotope 1018.4913189751 has abundance 30.52912950515747 %
Isotope 1019.4946738129 has abundance 9.80210527776718 %
Isotope 1020.4980286507 has abundance 2.329206666748047 %
Isotope 1021.5013834885 has abundance 0.4492596257477999 %
Isotope 1022.5047383263 has abundance 0.07378293084912002 %
===================================
y3 ion sequence: GER
y3 mz: 181.09514385
y3 molecular formula: C13H24N6O6

Note on lines 15 to 17 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 17 cannot refer to the same input peptide “DFPIANGER” since its “b3” ion would be “DFP” and not “GER”).

7.2 Modified 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. ") 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"), AASequence.fromString("DFPIAM[+16]GER") and AASequence.fromString("DFPIAM[147]GER") are all equivalent).

```python
from pyopenms import *
seq = AASequence.fromString("PEPTIDESEKUEM(Oxidation)CER")
print(seq.toUnmodifiedString())
print(seq.toString())
print(seq.toUniModString())
print(seq.toBracketString())
print(seq.toBracketString(False))
print(AASequence.fromString("DFPIAM(UniMod:35)GER"))
print(AASequence.fromString("DFPIAM[+16]GER"))
print(AASequence.fromString("DFPIAM[+15.99]GER"))
print(AASequence.fromString("DFPIAM[147]GER"))
print(AASequence.fromString("DFPIAM[147.035405]GER"))
```

The above code outputs:

PEPTIDESEKUEMCER
PEPTIDESEKUEM(Oxidation)CER
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. In all instances, it is better to use an exact description of the desired modification, such as UniMod, instead of mass differences.

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:

```python
from pyopenms import *
s = AASequence.fromString(".(Dimethyl)DFPIAMGER.")
print(s, s.hasNTerminalModification())
s = AASequence.fromString("DFPIAMGER.(Label:18O(2))")
print(s, s.hasCTerminalModification())
s = AASequence.fromString("DFPIAMGER(Phospho).")
print(s, s.hasCTerminalModification())
```

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 = "MKWVTFISLLLLFSSAYSRGVFRDDTHKSEIAHRFKDLGE"
bsa.description = "BSA Bovine Albumin (partial sequence)"
bsa.identifier = "BSA"
alb = FASTAEntry()
alb.sequence = "MKWVTFISLLLLFSSAYSRGVFRDDAHKSEVAHRFKDLGE"
alb.description = "ALB Human Albumin (partial sequence)"
alb.identifier = "ALB"
entries = [bsa, alb]
```
f = FASTAFile()
f.store("example.fasta", entries)

Afterwards, the example.fasta file can be read again from disk:

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

OpenMS also supports the representation of RNA oligonucleotides using the \texttt{NASequence} class:

```python
from pyopenms import *

oligo = NASequence.fromString("AAUGCAUGG")
prefix = oligo.getPrefix(4)
suffix = oligo.getSuffix(4)

print(oligo)
print(prefix)
print(suffix)

print("Oligo length", oligo.size())
print("Total precursor mass", oligo.getMonoWeight())
print("y1+ ion mass of", str(prefix), ":", prefix.getMonoWeight(NASequence.
  \rightarrow NASFragmentType.YIon, 1))

seq_formula = oligo.getFormula()
print("RNA Oligo", oligo, "has molecular formula", seq_formula)

isotopes = seq_formula.getIsotopeDistribution( CoarseIsotopePatternGenerator(6) )
for iso in isotopes.getContainer():
    print("Isotope", iso.getMZ(), ":", iso.getIntensity())
```

Which will output

```
AAUGCAUGG
AAUG
```
AUGG

Oligo length 10
Total precursor mass 3206.4885302061
y1+ ion mass of AAUG : 1248.2298440331

RNA Oligo AAUGCAAUGG has molecular formula C97H119N42O66P9

Isotope 3206.4885302061 : 0.25567981600761414
Isotope 3207.4918850439003 : 0.31783154606819153
Isotope 3208.4952398817004 : 0.23069815337657928
Isotope 3209.4985947195 : 0.12306403368711472
Isotope 3210.5019495573 : 0.053163252770900726
Isotope 3211.5053043951 : 0.01956319250166416

The NASequence object also allows iterations directly in Python:

```python
from pyopenms import *
oligo = NASequence.fromString("AAUGCAAUGG")
print("The oligonucleotide", str(oligo), "consists of the following nucleotides:")
for ribo in oligo:
    print(ribo.getName())
```

### 8.1.1 Fragment ions

Similarly to before for amino acid sequences, we can also generate internal fragment ions:

```python
from pyopenms import *
oligo = NASequence.fromString("AAUGCAAUGG")
suffix = oligo.getSuffix(4)
print("RNA Oligo w4++ ion", suffix, "has mz", suffix.getMonoWeight(), "/ charge")
charge = 2
mass = suffix.getMonoWeight(NASequence.NASFragmentType.WIon, charge)
w4_formula = suffix.getFormula(NASequence.NASFragmentType.WIon, charge)
mz = mass / charge
print("RNA Oligo w4++ ion", suffix, "has molecular formula", w4_formula)
```

### 8.2 Modified oligonucleotides

Modified nucleotides can also represented by the Ribonucleotide class and are specified using a unique string identifier present in the RibonucleotideDB in square brackets. For example, [m1A] represents 1-methyladenosine. We can create a NASequence object by parsing a modified sequence as follows:

```python
from pyopenms import *
oligo_mod = NASequence.fromString("A[m1A][Gm]A")
```
seq_formula = oligo_mod.getFormula()
print("RNA Oligo", oligo_mod, "has molecular formula",
    seq_formula, "and length", oligo_mod.size())
print("*"*35)
oligo_list = [oligo_mod[i].getOrigin() for i in range(oligo_mod.size())]
print("RNA Oligo", oligo_mod.toString(), "has unmodified sequence", ".join(oligo_ 
    list))
for i in range(oligo_mod.size()):
    print (oligo_mod[i].isModified())

---

8.3 DNA, RNA and Protein

We can also work with DNA and RNA sequences in combination with the BioPython library (you can install BioPython with pip install biopython):

```python
from pyopenms import *
from Bio.Seq import Seq
from Bio.Alphabet import IUPAC
bsa = FASTAEntry()
bsa.sequence = 'ATGAAGTGGGTGACTTTTATTTCTCTTCTCTTCTCAGTCTCCTATTTCAAGGCTGGGTGTTTCTG'
bsa.description = "BSA Bovine Albumin (partial sequence)"
bsa.identifier = "BSA"
entries = [bsa]
f = FASTAFile()
f.store("example_dna.fasta", entries)
coding_dna = Seq(bsa.sequence, IUPAC.unambiguous_dna)
coding_rna = coding_dna.transcribe()
protein_seq = coding_rna.translate()
oligo = NASequence.fromString(str(coding_rna))
aaseq = AASequence.fromString(str(protein_seq))
print("The RNA sequence", str(oligo), "has mass", oligo.getMonoWeight(), "and"
    "translates to the protein sequence", str(aaseq), "which has mass", aaseq.
    getMonoWeight() )
```
CHAPTER 9

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.

9.1 Y-ion spectrum

First, we will generate a simple spectrum that only contains y ions

```python
from pyopenms import *

tsg = TheoreticalSpectrumGenerator()
spec1 = MSSpectrum()
peptide = AASequence.fromString("DFPIANGER")
# standard behavior is adding b- and y-ions of charge 1
p = Param()
p.setValue("add_b_ions", "false")
p.setValue("add_metainfo", "true")
tsg.setParameters(p)
tsg.getSpectrum(spec1, peptide, 1, 1)

# Iterate over annotated ions and their masses
print("Spectrum 1 of", peptide, "has", spec1.size(), "peaks.")
for ion, peak in zip(spec1.getStringDataArrays()[0], spec1):
    print(ion.decode(), "is generated at m/z", peak.getMZ())
```

which produces all y single charged ions:

Spectrum 1 of DFPIANGER has 8 peaks.
y1+ is generated at m/z 175.118952913371
y2+ is generated at m/z 304.161547136671
y3+ is generated at m/z 361.18301123237103
y4+ is generated at m/z 475.225939423771
y5+ is generated at m/z 546.2630535832709

(continues on next page)
9.2 Full fragment ion spectrum

We can also produce additional peaks in the fragment ion spectrum, such as isotopic peaks, higher charge states, additional ion series or common neutral losses:

```python
spec2 = MSSpectrum()
p.setValue("add_b_ions", "true")
p.setValue("add_a_ions", "true")
p.setValue("add_losses", "true")
p.setValue("add_metainfo", "true")
tsg.setParameters(p)
tsg.getSpectrum(spec2, peptide, 1, 2)

# Iterate over annotated ions and their masses
print("Spectrum 2 of", peptide, "has", spec2.size(), "peaks.")
for ion, peak in zip(spec2.getStringDataArrays()[0], spec2):
    print(ion.decode(), "is generated at m/z", peak.getMZ())
```

which outputs all 146 peaks that are generated (this is without isotopic peaks), here we will just show the first few peaks:

<table>
<thead>
<tr>
<th>Ion</th>
<th>M/z</th>
</tr>
</thead>
<tbody>
<tr>
<td>y1-C1H2N1O1++</td>
<td>66.05629515817103</td>
</tr>
<tr>
<td>y1-C1H2N2++</td>
<td>67.05221565817102</td>
</tr>
<tr>
<td>y1-H3N1++</td>
<td>79.54984014222102</td>
</tr>
<tr>
<td>y1+</td>
<td>88.06311469007102</td>
</tr>
<tr>
<td>a2-H2O1++</td>
<td>109.05221565817101</td>
</tr>
<tr>
<td>a2+</td>
<td>118.05749819007102</td>
</tr>
<tr>
<td>b2-H2O1++</td>
<td>123.04967315817101</td>
</tr>
<tr>
<td>y2-C1H2N1O1++</td>
<td>130.57759226982103</td>
</tr>
<tr>
<td>y1-C1H2N1O1+</td>
<td>131.10531384957102</td>
</tr>
<tr>
<td>y2-C1H2N2++</td>
<td>131.57351276982102</td>
</tr>
<tr>
<td>b2+</td>
<td>132.05495569007101</td>
</tr>
<tr>
<td>y1-C1H2N2+</td>
<td>133.09715484957101</td>
</tr>
<tr>
<td>y2-H2O1++</td>
<td>143.57912926982102</td>
</tr>
<tr>
<td>y2-H3N1++</td>
<td>144.07113725387103</td>
</tr>
<tr>
<td>y2+</td>
<td>152.58441180172102</td>
</tr>
</tbody>
</table>

The first example shows how to put peaks of a certain type, y-ions in this case, into a spectrum. The second spectrum is filled with a complete fragment ion spectrum of all peaks (a-, b-, y-ions and losses). The losses are based on commonly observed fragment ion losses for specific amino acids and are defined in the Residues.xml file, which means that not all fragment ions will produce all possible losses, as can be observed above: water loss is not observed for the y1 ion but for the y2 ion since glutamic acid can have a neutral water loss but arginine cannot. Similarly, only water
loss and no ammonia loss is simulated in the \(a/b/c\) ion series with the first fragment capable of ammonia loss being asparagine at position 6.

The `TheoreticalSpectrumGenerator` has many parameters which have a detailed description located in the class documentation. Note how the `add_metainfo` parameter populates the `StringDataArray` of the output spectrum, allowing us to iterate over annotated ions and their masses.

### 9.3 Visualization

We can now visualize the resulting spectra using TOPPView when we open the DFPIANGER.mzML file that we produced above in TOPPView:

![Visualization of DFPIANGER.mzML file](image)

We can see all eight \(y\) ion peaks that are produced in the `TheoreticalSpectrumGenerator` and when we hover over one of the peaks (546 mz in this example) there is an annotation in the bottom left corner that indicates charge state and ion name (\(y^5+\) for every peak). The larger spectrum with 146 peaks can also be interactively investigated with TOPPView (the second spectrum in the file):
There are substantially more peaks here and the spectrum is much busier, with singly and double charged peaks of the b, y and a series creating 44 different individual fragment ion peaks as well as neutral losses adding an additional 102 peaks (neutral losses easily recognizable by their 10-fold lower intensity in the simulated spectrum).
10.1 Proteolytic Digestion with Trypsin

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

```python
from pyopenms import *
from urllib.request import urlretrieve
# from urllib import urlretrieve  # use this code for Python 2.x
urlretrieve("http://www.uniprot.org/uniprot/P02769.fasta", "bsa.fasta")

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

10.2 Proteolytic Digestion with Lys-C

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

```python
from pyopenms import *
names = []
ProteaseDB().getAllNames(names)
len(names)  # at least 25 by default
e = ProteaseDB().getEnzyme('Lys-C')
e.getRegExDescription()
e.getRegEx()
```
Now that we have learned about the other enzymes available, we can use it to cut our protein of interest:

```python
from pyopenms import *
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
urlretrieve ('http://www.uniprot.org/uniprot/P02769.fasta', "bsa.fasta")

dig = ProteaseDigestion()
dig.setEnzyme('Lys-C')
bsa = ".".join([l.strip() for l in open("bsa.fasta").readlines()[1:]])
bsa = AASequence.fromString(bsa)
result = []
dig.digest(bsa, result)
print(result[4].toString())

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).

### 10.3 Oligonucleotide Digestion

There are multiple cleavage enzymes available for oligonucleotides, these are defined in Enzymes_RNA.xml file and can be accessed using the RNaseDB object

```python
from pyopenms import *
db = RNaseDB()
names = []
db.getAllNames(names)

# Will print out all available enzymes:
# ['RNase_U2', 'RNase_T1', 'RNase_H', 'unspecific cleavage', 'no cleavage', 'RNase_MC1', 'RNase_A', 'cusativin']
e = db.getEnzyme("RNase_T1")
e.getRegEx()
e.getThreePrimeGain()
```

We can now use it to cut an oligo:

```python
from pyopenms import *
oligo = NASequence.fromString("pAUGUCGCAG");

dig = RNaseDigestion()
dig.setEnzyme("RNase_T1")

result = []
dig.digest(oligo, result)
for fragment in result:
    print (fragment)

print("Looking closer at", result[0])
print(" Five Prime modification:", result[0].getFivePrimeMod().getCode())
print(" Three Prime modification:", result[0].getThreePrimeMod().getCode())
for ribo in result[0]:
    print (ribo.getCode(), ribo.getMonoMass(), ribo.isModified())
```

52 Chapter 10. Digestion
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.

### 11.1 ProteinIdentification

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

```python
from pyopenms import *

# 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("sp|MyAccession")
protein_hit.setSequence("PEPTIDERDLQMTQSPSSLSVSVGDRPEPTIDE")
protein_hit.setScore(1.0)
protein_hit.setMetaValue("target_decoy", b"target")  # its a target protein

protein_id.setHits([protein_hit])
```
We have now added a single ProteinHit with the accession sp|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):

```python
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())
```

### 11.2 PeptideIdentification

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

```python
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, best score)
peptide_hit = PeptideHit()
peptide_hit.setScore(1.0)
peptide_hit.setRank(1)
peptide_hit.setCharge(2)
peptide_hit.setSequence(AASequence.fromString("DLQM(Oxidation)TQSPSSLSVSVGDR"))

ev = PeptideEvidence()
ev.setProteinAccession("sp|MyAccession")
ev.setAABefore(b"R")
ev.setAAAfter(b"P")
peptide_hit.setPeptideEvidences([ev])

# create a new PeptideHit (second best PSM, lower score)
peptide_hit2 = PeptideHit()
```

(continues on next page)
peptide_hit2.setScore(0.5)
peptide_hit2.setRank(2)
peptide_hit2.setCharge(2)
peptide_hit2.setSequence(AASequence.fromString("QDLMTQSPSSLSVSVGDR"))
peptide_hit2.setPeptideEvidences([ev])

# add PeptideHit to PeptideIdentification
peptide_id.setHits([peptide_hit, peptide_hit2])

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”).

We can now display the peptides we just stored:

```python
# 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())
        print(" - Mapping to proteins:", [ev.getProteinAccession() for ev in hit.getPeptideEvidences() ] )
```

11.3 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:

```python
# 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())
        print("Protein hit target/decoy:", hit.getMetaValue("target_decoy"))

# 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())
```

(continues on next page)
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())
    print(" - Mapping to proteins:", [ev.getProteinAccession() for ev in hit.getPeptideEvidences()])

You can inspect the out.idXML XML file produced here, and you will find a <ProteinHit> entry for the protein that we stored and two <PeptideHit> entries for the two peptides stored on disk.
12.1 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:
Note that in this case only 2 features are present, but in a typical LC-MS/MS experiments, thousands of features are present.

12.2 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())
```

12.3 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 *
f = ConsensusFeature()
f.setMZ(500.9)
```
```python
def feature.setCharge(2)
def feature.setRT(1500.1)
def 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()
files = { 1 : ColumnHeader(), 2 : ColumnHeader() }
files[1].filename = "file1"
files[2].filename = "file2"
cmap.setColumnHeaders(files)
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:
12.4 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())
```
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.

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve  # use this code for Python 2.x
urlretrieve ("http://proteowizard.sourceforge.net/example_data/tiny.pwiz.1.1.mzML", "test.mzML")
```

13.1 Filtering Spectra

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

```python
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)
```

13.1.1 Filtering by MS level

Similarly, we can filter the test.mzML file by MS level, retaining only spectra that are not MS1 spectra (e.g. MS2, MS3 or MSn spectra):
from pyopenms import *
inp = MSExperiment()
MzMLFile().load("test.mzML", inp)
e = MSExperiment()
for s in inp:
    if s.getMSLevel() > 1:
        e.addSpectrum(s)
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:

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

### 13.1.2 Filtering by scan number

Or we could use an external list of scan numbers to filter by scan numbers, thus only retaining MS scans in which we are interested in:

from pyopenms import *
inp = MSExperiment()
scan_nrs = [0, 2, 5, 7]
e = MSExperiment()
for k, s in enumerate(inp):
    if k in scan_nrs and s.getMSLevel() == 1:
        e.addSpectrum(s)
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:

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

### 13.2 Filtering Spectra and Peaks

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

from pyopenms import *
inp = MSExperiment()
MzMLFile().load("test.mzML", inp)
mz_start = 6.0
mz_end = 12.0
e = MSExperiment()
for s in inp:
    if s.getMSLevel() > 1:
        filtered_mz = []
(continues on next page)
filtered_int = []
for mz, i in zip(*s.get_peaks()):
    if mz > mz_start and mz < mz_end:
        filtered_mz.append(mz)
        filtered_int.append(i)
    s.set_peaks((filtered_mz, filtered_int))
e.addSpectrum(s)
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.

### 13.3 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).
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.

```python
from pyopenms import *
p = Param()
p.setValue("param1", 4.0, "This is value 1")
p.setValue("param2", 5.0, "This is value 2")
print( p[b"param1"] )
p[b"param1"] += 3 # add three to the parameter value
print( p[b"param1"] )
```

The parameters can then be accessed as

```python
>>> p.asDict()
{'param2': 4.0, 'param1': 7.0}
>>> p.values()
[4.0, 7.0]
>>> p.keys()
['param1', 'param2']
>>> p.items()
[('param1', 7.0), ('param2', 4.0)]
>>> p.exists("param1")
True
```
Most signal processing algorithms follow a similar pattern in OpenMS.

```python
class Algorithm:
    def __init__(self):
        pass

def filterExperiment(self, exp):
    # perform some processing on exp
```

In many cases, the processing algorithms have a set of parameters that can be adjusted. These are accessible through `getParameters()` and yield a `Param` object (see 'Parameter handling<parameter_handling.html>') which can be manipulated. After changing parameters, one can use `setParameters()` to propagate the new parameters to the algorithm:

```python
class Algorithm:
    def __init__(self):
        pass

def getParameters(self):
    return Param()  # return a Param object

def setParameters(self, param):
    pass
```

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, we can apply this approach as follows:

```python
from pyopenms import *
exp = MSExperiment()
gf = GaussFilter()
MzMLFile().load("test.mzML", exp)
```
gf.filterExperiment(exp)
MzMLFile().store("test.filtered.mzML", exp)
In many applications, mass spectrometric data should be smoothed first before further analysis.

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve  # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urlretrieve (gh +"/share/OpenMS/examples/peakpicker_tutorial_1_baseline_filtered.mzML 
˓→", "tutorial.mzML")

exp = MSExperiment()
gf = GaussFilter()
param = gf.getParameters()
param.setValue("gaussian_width", 1.0)  # needs wider width
gf.setParameters(param)

MzMLFile().load("tutorial.mzML", exp)
gf.filterExperiment(exp)
MzMLFile().store("tutorial.smoothed.mzML", exp)
```

We can now load our data into TOPPView to observe the effect of the smoothing, which becomes apparent when we overlay the two files (drag onto each other) and then zoom into a given mass range using Ctrl-G and select 4030 to 4045:
In the screenshot above we see the original data (red) and the smoothed data (black), indicating that the smoothing does clean up noise in the data significantly and will prepare the data for downstream processing, such as peak-picking.
**17.1 Fragment mass to amino acid composition**

One challenge often encountered in mass spectrometry is the question of the composition of a specific mass fragment only given its mass. For example, for the internal fragment mass 262.0953584466 there are three different interpretations within a narrow mass band of 0.05 Th:

```python
>>> AASequence.fromString("MM").getMonoWeight(Residue.ResidueType.Internal, 0)
262.08097003420005
>>> AASequence.fromString("VY").getMonoWeight(Residue.ResidueType.Internal, 0)
262.1317435742
>>> AASequence.fromString("DF").getMonoWeight(Residue.ResidueType.Internal, 0)
262.0953584466
```

As you can see, already for relatively simple two-amino acid combinations, multiple explanations may exist. OpenMS provides an algorithm to compute all potential amino acid combinations that explain a certain mass in the `MassDecompositionAlgorithm` class:

```python
from pyopenms import *
md_alg = MassDecompositionAlgorithm()
param = md_alg.getParameters()
param.setValue("tolerance", 0.05)
param.setValue("residue_set", b"Natural19WithoutI")
md_alg.setParameters(param)
decomps = []
md_alg.getDecompositions(decomps, 262.0953584466)
for d in decomps:
    print(d.toExpandedString().decode())
```

Which outputs the three potential compositions for the mass 262.0953584466. Note that every single combination of amino acids is only printed once, e.g. only DF is reported while the isobaric FD is not reported. This makes the algorithm more efficient.
17.2 Naive algorithm

We can compare this result with a more naive algorithm which simply iterates through all combinations of amino acid residues until the sum of all residues equals the target mass:

```python
mass = 262.0953584466
residues = ResidueDB().getResidues(b"Natural19WithoutI")
def recursive_mass_decomposition(mass_sum, peptide):
    if abs(mass_sum - mass) < 0.05:
        print(peptide + "\t" + str(mass_sum))
    for r in residues:
        new_mass = mass_sum + r.getMonoWeight(Residue.ResidueType.Internal)
        if new_mass < mass + 0.05:
            recursive_mass_decomposition(new_mass, peptide + r.getOneLetterCode().decode())

print("Mass explanations by naive algorithm:")
recursive_mass_decomposition(0, "")
```

Note that this approach is substantially slower than the OpenMS algorithm and also does not treat DF and FD as equivalent, instead outputting them both as viable solutions.

17.3 Stand-alone Program

We can use pyOpenMS to write a short program that takes a mass and outputs all possible amino acid combinations for that mass within a given tolerance:

```python
from pyopenms import *
import sys

# Example for mass decomposition (mass explanation)
# Internal residue masses (as observed e.g. as mass shifts in tandem mass spectra)
# are decomposed in possible amino acid strings that match in mass.

mass = float(sys.argv[1])
tol = float(sys.argv[2])

md_alg = MassDecompositionAlgorithm()
param = md_alg.getParameters()
param.setValue("tolerance", tol)
param.setValue("residue_set", b"Natural19WithoutI")
md_alg.setParameters(param)
decomps = []
md_alg.getDecompositions(decomps, mass)
for d in decomps:
    print(d.toExpandedString().decode())
```

If we copy the above code into a script, for example `mass_decomposition.py`, we will have a stand-alone software that takes two arguments: first the mass to be decomposed and secondly the tolerance to be used (which are collected on line 8 and 9). We can call it as follows:

```
python mass_decomposition.py 999.4773990735001 1.0
python mass_decomposition.py 999.4773990735001 0.001
```

Try to change the tolerance parameter. The parameter has a very large influence on the reported results, for example for 1.0 tolerance, the algorithm will produce 80 463 results while for a 0.001 tolerance, only 911 results are expected.
from pyopenms import *

tsg = TheoreticalSpectrumGenerator()
param = tsg.getParameters()
param.setValue("add_metainfo", "false")
param.setValue("add_first_prefix_ion", "true")
param.setValue("add_a_ions", "true")
param.setValue("add_losses", "true")
param.setValue("add_precursor_peaks", "true")
tsg.setParameters(param)

# spectrum with charges +1 and +2

spec = MSSpectrum()
tsg.getSpectrum(spec, test_sequence, 1, 2)

print(spec.size())  # should be 357

# tagger searching only for charge +1

tagger = Tagger(2, 10.0, 5, 1, 1, [], [])
tagger.getTag(spec, tags)

print(len(tags))  # should be 890

b"EPTID" in tags  # True
b"PTIDE" in tags  # True
b"PTIDEF" in tags  # False
A single mass spectrum contains measurements of one or more analytes and the m/z values recorded for these analytes. Most analytes produce multiple signals in the mass spectrometer, due to the natural abundance of carbon 13 (naturally occurring at ca. 1% frequency) and the large amount of carbon atoms in most organic molecules, most analytes produce a so-called isotopic pattern with a monoisotopic peak (all carbon are C12) and a first isotopic peak (exactly one carbon atom is a C13), a second isotopic peak (exactly two atoms are C13) etc. Note that also other elements can contribute to the isotope pattern, see the Chemistry section for further details.

In addition, each analyte may appear in more than one charge state and adduct state, a singly charge analyte $[M+H]^+$ may be accompanied by a doubly charged analyte $[M+2H]^{++}$ or a sodium adduct $[M+Na]^+$. In the case of a multiply charged peptide, the isotopic traces are spaced by $\text{PROTON\_M MASS} / \text{charge\_state}$ which is often close to 0.5 m/z for doubly charged analytes, 0.33 m/z for triply charged analytes etc. Note: tryptic peptides often appear at least doubly charged, while small molecules often carry a single charge but can have adducts other than hydrogen.

18.1 Single peak example

```python
from pyopenms import *

charge = 2
seq = AASequence.fromString("DFPIANGER")
seq_formula = seq.getFormula() + EmpiricalFormula("H" + str(charge))
isotopes = seq_formula.getIsotopeDistribution(CoarseIsotopePatternGenerator(6))
print("[M+H]+ weight:", seq.getMonoWeight(Residue.ResidueType.Full, 1))

# Append isotopic distribution to spectrum
s = MSSpectrum()
for iso in isotopes.getContainer():
    iso.setMZ( iso.getMZ() / charge )
    s.push_back(iso)
    print("Isotope", iso.getMZ(), ":", iso.getIntensity())

Deisotoper.deisotopeAndSingleChargeDefault(s, 10, True)
```

(continues on next page)
for p in s:
    print(p.getMZ(), p.getIntensity() )

Note that the algorithm presented here as some heuristics built into it, such as assuming that the isotopic peaks will decrease after the first isotopic peak. This heuristic can be tuned by changing the parameter use_decreasing_model and start_intensity_check. In this case, the second isotopic peak is the highest in intensity and the start_intensity_check parameter needs to be set to 3.

from pyopenms import *
charge = 4
seq = AASequence.fromString("DFPIANGERDFPIANGERDFPIANGERDFPIANGER")
seq_formula = seq.getFormula() + EmpiricalFormula("H" + str(charge))
isotopes = seq_formula.getIsotopeDistribution(CoarseIsotopePatternGenerator(8))
print("[M+H]+ weight:", seq.getMonoWeight(Residue.ResidueType.Full, 1))

# Append isotopic distribution to spectrum
s = MSSpectrum()
for iso in isotopes.getContainer():
    iso.setMZ( iso.getMZ() / charge )
    s.push_back(iso)
    print("Isotope", iso.getMZ(), ":", iso.getIntensity())

min_charge = 1
min_isotopes = 2
max_isotopes = 10
use_decreasing_model = True
start_intensity_check = 3
Deisotoper.deisotopeAndSingleCharge(s, 10, True, min_charge, charge, True,
min_isotopes, max_isotopes,
True, True, True,
use_decreasing_model, start_intensity_check,,
False)
for p in s:
    print(p.getMZ(), p.getIntensity() )

18.2 Full spectral de-isotoping

In the following code segment, we will use a sample measurement of BSA (Bovine Serum Albumin), and apply a simple algorithm in OpenMS for “deisotoping” a mass spectrum, which means grouping peaks of the same isotopic pattern charge state:

from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urlretrieve (gh +"/share/OpenMS/examples/BSA/BSA1.mzML", "BSA1.mzML")

from pyopenms import *
e = MSExperiment()
MzMLFile().load("BSA1.mzML", e)
s = e[214]
s.setFloatDataArrays([])

(continues on next page)
Deisotoper.deisotopeAndSingleCharge(s, 0.1, False, 1, 3, True, 2, 10, True, True)

print(e[214].size())
print(s.size())

e2 = MSExperiment()
e2.addSpectrum(e[214])
MzMLFile().store("BSA1_scan214_full.mzML", e2)
e2 = MSExperiment()
e2.addSpectrum(s)
MzMLFile().store("BSA1_scan214_deisotoped.mzML", e2)

maxvalue = max([p.getIntensity() for p in s])
for p in s:
  if p.getIntensity() > 0.25 * maxvalue:
    print(p.getMZ(), p.getIntensity() )

which produces the following output

140
41

974.4572680576728 6200571.5
974.4589691256419 3215808.75

As we can see, the algorithm has reduced 140 peaks to 41 deisotoped peaks. It also has identified a molecule at 974.45 m/z as the most intense peak in the data (basepeak).

18.3 Visualization

The reason we see two peaks very close together becomes apparent once we look at the data in TOPPView which indicates that the 974.4572680576728 peak is derived from a 2+ peak at m/z 487.73 and the peak at 974.4589691256419 is derived from a 3+ peak at m/z 325.49: the algorithm has identified a single analyte in two charge states and deconvoluted the peaks to their nominal mass of a [M+H]+ ion, which produces two peaks very close together (2+ and 3+ peak):
Looking at the full spectrum and comparing it to the original spectrum, we can see the original (centroided) spectrum on the top and the deisotoped spectrum on the bottom in blue. Note how hovering over a peak in the deisotoped spectrum indicates the charge state:

In the next section, we will look at 2-dimensional deisotoping where instead of a single spectrum, multiple spectra from a LC-MS experiments are analyzed together. There algorithms analyze the full 2-dimensional (m/z and RT) signal and are generally more powerful than the 1-dimensional algorithm discussed here. However, not all data is 2 dimensional and the algorithm discussed here has many application in practice (e.g. single mass spectra, fragment ion
spectra in DDA etc.).
One very common task in mass spectrometry is the detection of 2-dimensional patterns in m/z and time (RT) dimension from a series of MS1 scans. These patterns are called **Features** and they exhibit a chromatographic elution profile in the time dimension and an isotopic pattern in the m/z dimension (see previous section for the 1-dimensional problem). OpenMS has multiple tools that can identify these features in 2-dimensional data, these tools are called **FeatureFinder**. Currently the following FeatureFinders are available in OpenMS:

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

All of the algorithms above are for proteomics data with the exception of FeatureFinderMetabo which works on metabolomics data. 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:

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urlretrieve (gh +"/src/tests/topp/FeatureFinderCentroided_1_input.mzML", "feature_test.mzML")

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("feature_test.mzML", input_map)
input_map.updateRanges()

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

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

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

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:

```python
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 learn 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.
In MS-based proteomics, fragment ion spectra (MS2 spectra) are often interpreted by comparing them against a theoretical set of spectra generated from a FASTA database. OpenMS contains a (simple) implementation of such a “search engine” that compares experimental spectra against theoretical spectra generated from an enzymatic or chemical digest of a proteome (e.g. tryptic digest).

### 20.1 SimpleSearch

In most proteomics applications, a dedicated search engine (such as Comet, Crux, Mascot, MSGFPlus, MSFragger, Myrimatch, OMSSA, SpectraST or XTandem; all of which are supported by pyOpenMS) will be used to search data. Here, we will use the internal `SimpleSearchEngineAlgorithm` from OpenMS used for teaching purposes. This makes it very easy to search an (experimental) mzML file against a fasta database of protein sequences:

```python
from urllib.request import urlretrieve
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urlretrieve (gh +"/src/tests/topp/SimpleSearchEngine_1.mzML", "searchfile.mzML")
urlretrieve (gh +"/src/tests/topp/SimpleSearchEngine_1.fasta", "search.fasta")
protein_ids = []
peptide_ids = []
SimpleSearchEngineAlgorithm().search("searchfile.mzML", "search.fasta", protein_ids,
   peptide_ids)
```

This will print search engine output including the number of peptides and proteins in the database and how many spectra were matched to peptides and proteins:

```
Peptide statistics
unmatched : 0 (0 %)
target/decoy: match to target DB only: 2 (100 %)
```

(continues on next page)
20.2 PSM inspection

We can now investigate the individual hits as we have done before in the Identification tutorial.

```python
for peptide_id in peptide_ids:
    # Peptide identification values
    print(35*'=')
    print("Peptide ID m/z: ", peptide_id.getMZ())
    print("Peptide ID rt: ", peptide_id.getRT())
    print("Peptide scan index: ", peptide_id.getMetaValue("scan_index"))
    print("Peptide scan name: ", peptide_id.getMetaValue("scan_index"))
    print("Peptide ID score type: ", peptide_id.getScoreType())
    # PeptideHits
    for hit in peptide_id.getHits():
        print(" - Peptide hit rank:", hit.getRank())
        print(" - Peptide hit charge:", hit.getCharge())
        print(" - Peptide hit sequence:", hit.getSequence())
        mz = hit.getSequence().getMonoWeight(Residue.ResidueType.Full, hit.getCharge()) / hit.getCharge()
        print(" - Peptide hit monoisotopic m/z:", mz)
        print(" - Peptide ppm error:", abs(mz - peptide_id.getMZ())/mz *10**6 )
        print(" - Peptide hit score:", hit.getScore())
```

We notice that the second peptide spectrum match (PSM) was found for the third spectrum in the file for a precursor at 775.38 m/z for the sequence RPGADSDIGGFGGLFDLQAQGFR.

```python
tsg = TheoreticalSpectrumGenerator()
thspec = MSSpectrum()
p = Param()
p.setValue("add_metainfo", "true")
tsg.setParameters(p)
peptide = AASequence.fromString("RPGADSDIGGFGGLFDLQAQGFR")
tsg.getSpectrum(thspec, peptide, 1, 1)
# Iterate over annotated ions and their masses
for ion, peak in zip(thspec.getStringDataArrays()[0], thspec):
    print(ion, peak.getMZ())

e = MSExperiment()
MzMLFile().load("searchfile.mzML", e)
print("Spectrum native id", e[2].getNativeID())
mz,i = e[2].get_peaks()
peaks = [(mz,i) for mz,i in zip(mz,i) if i > 1500 and mz > 300]
for peak in peaks:
    print(peak[0], "mz", peak[1], "int")
```

Comparing the theoretical spectrum and the experimental spectrum for RPGADSDIGGFGGLFDLQAQGFR we can easily see that the most abundant ions in the spectrum are y8 (877.452 m/z), b10 (926.432), y9 (1024.522 m/z) and b13 (1187.544 m/z).
20.3 Visualization

When loading the `searchfile.mzML` into the OpenMS visualization software TOPPView, we can convince ourselves that the observed spectrum indeed was generated by the peptide `RPGADSDIGGFGGLFDLAQAGFR` by loading the corresponding theoretical spectrum into the viewer using “Tools”->”Generate theoretical spectrum”:

![TOPPView screenshot](image)

From our output above, we notice that the second peptide spectrum match (PSM) at 775.38 m/z for sequence `RPGADSDIGGFGGLFDLAQAGFR` was found with an error tolerance of 2.25 ppm, therefore if we set the precursor mass tolerance to 4 ppm (+/- 2ppm), we expect that we will not find the hit at 775.38 m/z any more:

```python
salgo = SimpleSearchEngineAlgorithm()
p = salgo.getDefaults()
print ( p.items() )
p[b'precursor:mass_tolerance'] = 4.0
salgo.setParameters(p)

protein_ids = []
peptide_ids = []
salgo.search("searchfile.mzML", "search.fasta", protein_ids, peptide_ids)
print("Found", len(peptide_ids), "peptides")
```

As we can see, using a smaller precursor mass tolerance leads the algorithm to find only one hit instead of two. Similarly, if we use the wrong enzyme for the digestion (e.g. `p[b'enzyme'] = "Formic_acid"`), we find no results.
In targeted proteomics, such as SRM / MRM / PRM / DIA applications, groups of chromatograms need to be analyzed frequently. OpenMS provides several powerful tools for analysis of chromatograms. Most of them are part of the OpenSWATH suite of tools and are also discussed in the OpenSwath documentation.

### 21.1 Peak Detection

Here, we will focus on a simple example where 2 peptides are analyzed. We will need 2 input files: the chromatogram files that contains the chromatographic raw data (raw SRM traces or extracted ion chromatograms from PRM/DIA data) as well as the library file used to generated the data which contains information about the targeted peptides:

```python
from urllib.request import urlretrieve
# from urllib import urlretrieve # use this code for Python 2.x
from pyopenms import *
gh = "https://raw.githubusercontent.com/OpenMS/OpenMS/develop"
urlretrieve (gh +"/src/tests/topp/OpenSwathAnalyzer_1_input_chrom.mzML", "chrom.mzML")
urlretrieve (gh +"/src/tests/topp/OpenSwathAnalyzer_1_input.TraML", "transitions.TraML"
....)

chroms = MSExperiment()
library = TargetedExperiment()
MzMLFile().load("chrom.mzML", chroms)
TraMLFile().load("transitions.TraML", library)

# Investigate library
for t in library.getTransitions():
    print ("Transition", t.getNativeID(), "belongs to peptide group", t.
    getPeptideRef())

print ("Input contains", len(library.getTransitions()), "transitions and", len(chroms.
    getChromatograms()), "chromatograms.")
features = FeatureMap()
dummy_traf0 = TransformationDescription()
```

(continues on next page)
dummy_exp = MSExperiment()
MRMFeatureFinderScoring().pickExperiment(chroms, features, library, dummy_trafo,
→ dummy_exp)

for f in features:
    print ("Feature for group", f.getMetaValue("PeptideRef"), "with precursor m/z", f.
    → getMetaValue("PrecursorMZ"))
    print (" Feature found at RT =", f.getRT(), "with library dot product", f.
    → getMetaValue("var_library_dotprod"))

Here we see that for the first group of transitions (tr_gr1), a single peak at retention time 3119 seconds was found. However, for the second group of transitions, two peaks are found at retention times 3119 seconds and at 3055 seconds.

21.2 Visualization

We can confirm the above analysis by visual inspection of the chrom.mzML file produced above in the TOPPView software:

However, our output above contains more information than only retention time:

Feature for group tr_gr1 with precursor m/z 500.0
  Feature found at RT = 3119.09196219877 with library dot product 0.9924204062692046
Feature for group tr_gr2 with precursor m/z 501.0
  Feature found at RT = 3055.58481870532 with library dot product 0.952054383474221
Feature for group tr_gr2 with precursor m/z 501.0
  Feature found at RT = 3119.0630105310684 with library dot product 0.7501676755451506

Based on the output above, we can infer that the peak at 3055 seconds is likely the correct peak for tr_gr2 since it has a high library dot product (0.95) while the peak at 3119 seconds is likely incorrect for tr_gr2 since its dot product is low (0.75). We also see that a peak at 3119 seconds is likely correct for tr_gr1 since it matches well with the expected library intensities and has a high dot product (0.99).
Note: to get an overview over all available scores for a particular MRM feature $f$, you can use

```python
k = []
f.getKeys(k)
print(k)
```

### 21.3 Smoothing

Now you may want to show the chromatograms to your collaborator, but you notice that most software solutions smooth the chromatograms before display. In order to provide smooth chromatograms, you can apply a filter using pyOpenMS:

```python
sg = SavitzkyGolayFilter()
sg.filterExperiment(chroms)
MzMLFile().store(chroms, "chrom.filter.mzML")
```

Which leads to the following smoothed chromatographic traces:
Note: This is an advanced section that dives deep into the mzML format and we will investigate the file format in greater detail. The intricacies of the mzML file format are all handled by pyOpenMS internally and this section is only intended for the interested reader.

Specifically, we will look at mzML stores raw spectral data and how this data is encoded in the XML format. The mzML standard is developed by the HUPO-PSI committee and can be read on the official mzML website. It describes how to store the meta data and the raw data for spectra and chromatograms. In short, the standard uses XML to encode all meta data and stores the raw data using Base64 encoding.

### 22.1 Binary encoding

To proceed, we will download an example file:

```python
from pyopenms import *
from urllib.request import urlretrieve
# from urllib import urlretrieve  # use this code for Python 2.x
urlretrieve("http://proteowizard.sourceforge.net/example_data/tiny.pwiz.1.1.mzML", "test.mzML")
```

Let's investigate the file `test.mzML` and look at line 197:

```python
>>> print(open("test.mzML").readlines()[197].strip())
<binary>
  AAAAAAAAAAAAAAAAAAAAAQAAAAAAAARBAAAAAAAAAGQAAAAAAAAACRAAAAAAAAAAKEAAAAAAAAAAQAAAAAAAADBAAAAA
</binary>
```

We see that line 197 in the `test.mzML` file contains the binary XML tag that contains a long datastring that starts with `AAAAA` and ends with `AAMkA`. This is the raw spectrum data encoded using Base64. We can confirm this by looking at some more context (lines 193 to 199 of the file):
We can now see that the surrounding XML tags describe how to decode the data, namely we see that the data is describes the m/z array and is uncompressed 64 bit data. We can now open the file with pyOpenMS and print the corresponding array which is from the second spectrum in the file:

```python
from pyopenms import *
exp = MSExperiment()
MzMLFile().load("test.mzML", exp)
print( exp.getSpectrum(1).get_peaks()[0] )
# [ 0. 2. 4. 6. 8. 10. 12. 14. 16. 18.]
```

We now see that the data encoded describes 10 m/z data points that are equally spaced in intervals of two, starting from 0 m/z and ending at 18 m/z (note: this is a synthetic dataset).

### 22.2 Base64 encoding

From the mzML standard, we know that the array is base64 encoded and we can now try to decode this data ourselves. We will first use pure Python functions:

```python
encoded_data = b"AAAAAAAAAAAAAAAAAAAAQAAAAAAAABBAAAAAAAAAGEAAAAAAAAAgQAAAAAAAACRAAAAAAAAAKEAAAAAAAAsQAADDBAAAAA" +
                b"AAAAAAAAACRAAAAAAAAAAKEAAAAAAAAsQAADDBAAAAAA"
import base64, struct
raw_data = base64.decodebytes(encoded_data)
out = struct.unpack('<%sd' % (len(raw_data) // 8), raw_data)
# struct.unpack('<%sf' % (len(raw_data) // 4), raw_data) # for 32 bit data
print(out)
# (0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0)
```

The code above uses the `base64` package on line 5 to decode the encoded data to raw binary data. On line 6, we use the `struct` package to transform the raw binary data to 64-bit floating point values. Note that `<%sd` is used for 64 bit data and `<%sf` for 32 bit data.

Alternatively, we could also use pyOpenMS to decode the same data:

```python
from pyopenms import *
out = []
Base64().decode64(encoded_data, Base64.ByteOrder.BYTEORDER_LITTLEENDIAN, out, False)
print( out )
# [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0]
```
This allows us thus to manually decode the data. We can use pyOpenMS to encode and decode 32 and 64 bit values:

```python
encoded_data = b"AAAAAAAAAAAAAAAAAAAAQAAAAAAAABBAAAAAAAAAGEAAAAAAAAAgQ" +
   b"AAAAAAAACRAAAAAAAAAAKEAAAAAAAAAsQAAAAAAAAADBAAAAAAAAAAMKA="
from pyopenms import *
out = []
Base64().decode64(encoded_data, Base64.ByteOrder.BYTEORDER_LITTLEENDIAN, out, False)
print(out)
# [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0]
data = String()
Base64().encode64(out, Base64.ByteOrder.BYTEORDER_LITTLEENDIAN, data, False)
print (data)
# b'AAAAAAAAAAAAAAAAAAAAQAAAAAAAABBAAAAAAAAAGEAAAAAAAAAgQAAAAAAAACRAAAAAAAAAAKEAAAAAAAAAsQAAAAAAAADBAAAAAAAAAAMKA=
Base64().encode64(out, Base64.ByteOrder.BYTEORDER_LITTLEENDIAN, data, True)
print (data)
# b'eJxjYEABDhBKAepLQGkFKK0CpTwgTA6UnoDSRg4A21QDYw=='
data = String()
Base64().encode32(out, Base64.ByteOrder.BYTEORDER_LITTLEENDIAN, data, False)
print (data)
# b'AAAAAAAAAAAAAAAQAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
```

Note how encoding the data with 64 bit precision results in an output string of length 108 characters that is about twice as long compared to encoding the data with 32 bit precision which is of length 56 characters. However, this difference disappears when zlib compression is used and the resulting string is shorter still.

### 22.3 numpress encoding

We can do even better, using the numpress compression. The numpress algorithm uses lossy compression, similar to jpeg compression, which is capable of compressing data even further but at the cost of not being able to recover the original input data exactly:

```python
from pyopenms import *
data = [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0 + 1e-8]
print(data)
# [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.00000001]
res = []
c = NumpressConfig()
c.np_compression = MSNumpressCoder.NumpressCompression.LINEAR
res = String()
MSNumpressCoder().encodeNP(data, res, False, c)
print(res)
# b'Qc//+/AAAAAAAAAA/v//f4iIlIew'
MSNumpressCoder().decodeNP(res, r, False, c)
print(r)
# [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.00000001024455]
```
c.np_compression = MSNumpressCoder.NumpressCompression.PIC
MSNumpressCoder().encodeNP(data, res, False, c)
print(res)
# b'hydHZ4enx+YBYhA='
MSNumpressCoder().decodeNP(res, r, False, c)
print(r)
# [0.0, 2.0, 4.0, 6.0, 8.0, 10.0, 12.0, 14.0, 16.0, 18.0]

Note how the lossy numpress compression leads to even shorter data, with 16 characters for PIC compression and 28 characters for linear compression. This makes the encoding much more efficient than lossless encoding that we have discussed above, however this is at the price of accuracy.

Different numpress compression schemes result in different accuracy, the LINEAR compression scheme introduced an inaccuracy of 10e-10 while the PIC (positive integer compression) can only store positive integers and results in greater loss of accuracy.
CHAPTER 23

pyOpenMS in R

Currently, there are no native wrappers for the OpenMS library in R, however we can use the “reticulate” package in order to get access to the full functionality of pyOpenMS in the R programming language.

23.1 Install the “reticulate” R package

In order to use all pyopenms functionalities in R, we suggest to use the “reticulate” R package. A thorough documentation is available at: https://rstudio.github.io/reticulate/

```r
install.packages("reticulate")
```

Installation of pyopenms is a requirement as well and it is necessary to make sure that R is using the same Python environment.

In case R is having trouble to find the correct Python environment, you can set it by hand as in this example (using miniconda, you will have to adjust the file path to your system to make this work). You will need to do this before loading the “reticulate” library:

```r
Sys.setenv(RETICULATE_PYTHON = "/usr/local/miniconda3/envs/py37/bin/python")
```

Or after loading the “reticulate” library:

```r
library("reticulate")
use_python("/usr/local/miniconda3/envs/py37/bin/python")
```

23.2 Import pyopenms in R

After loading the “reticulate” library you should be able to import pyopenms into R
library(reticulate)
ropenms=import("pyopenms", convert = FALSE)

This should now give you access to all of pyopenms in R. Importantly, the convert option has to be set to FALSE, since type conversions such as 64bit integers will cause a problem.

You should now be able to interact with the OpenMS library and, for example, read and write mzML files:

```r
library(reticulate)
ropenms=import("pyopenms", convert = FALSE)
exp = ropenms$MSExperiment()
ropenms$MzMLFile()$store("testfile.mzML", exp)
```

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

### 23.3 Getting help

Using the “reticulate” R package provides a way to access the pyopenms information about the available functions and methods. We can inspect individual pyOpenMS objects through the `py_help` function:

```r
library(reticulate)
ropenms=import("pyopenms", convert = FALSE)
idXML=ropenms$IdXMLFile
py_help(idXML)
```

Help on class IdXMLFile in module pyopenms.pyopenms_4:

```
class IdXMLFile(__builtin__.object)
 | Methods defined here:
 | __init__(...)
 | Cython signature: void IdXMLFile()
 | load(...)
 | Cython signature: void load(String filename, libc\_vector[ProteinIdentification] & protein_ids, libc\_vector[PeptideIdentification] & peptide_ids)
 | [...]
```

Alternatively, the autocompletion functionality of RStudio can be used:

```
> idXML=ropenms$IdXMLFile()
> idXML$load()
```

In this case, the help function indicates that the `idXML$load()` function requires
In order to read peptide identification data, we can download the idXML example file

```r
download.file("https://github.com/OpenMS/OpenMS/raw/master/share/OpenMS/examples/BSA/BSA1_OMSSA.idXML", "BSA1_OMSSA.idXML")
f="BSA1_OMSSA.idXML"
pepids=r_to_py(list())
protids=r_to_py(list())
idXML$load(f, protids, pepids)
pepids=py_to_r(pepids)
pephits=pepids[[1]]$getHits()
pepseq=pephits[[1]]$getSequence()
print(paste0("Sequence: ", pepseq))
[1] "Sequence: SHC(Carbamidomethyl)IAEVK"
```

Therefore in this case we need to use the `reticulate::r_to_py()` and `reticulate::py_to_r()` functions:

In order to get more information about the wrapped functions, we can also consult the pyOpenMS manual which references to all wrapped functions.

### 23.4 An example use case

#### 23.4.1 Reading an mzML File

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

```r
download.file("https://github.com/OpenMS/OpenMS/raw/master/share/OpenMS/examples/BSA/BSA1.mzML", "BSA1.mzML")
library(reticulate)
ropenms=import("pyopenms", convert = FALSE)
mzML=ropenms$MzMLFile()
exp = ropenms$MSExperiment()
mzML$load("BSA1.mzML", exp)
```

which will load the content of the “BSA1.mzML” file into the `exp` variable of type `MSExperiment`. We can now inspect the properties of this object:
which indicates that the variable `exp` has (among others) the functions `getNrSpectra` and `getNrChromatograms`. We can now try one of these functions:

```r
exp$getNrSpectra()
```

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

### 23.4.2 Visualize spectra

You can easily visualise ms1 level precursor maps:

```r
library(ggplot2)

spectra = py_to_r(exp$getSpectra())

peaks_df=c()
for (i in spectra) {
  if (i$getMSLevel()==1){
    peaks=do.call("cbind", i$get_peaks())
    rt=i$getRT()
    peaks_df=rbind(peaks_df, cbind(peaks, rt))
  }
}
peaks_df=data.frame(peaks_df)
colnames(peaks_df)=c('MZ','Intensity','RT')
peaks_df$Intensity=log10(peaks_df$Intensity)
ggplot(peaks_df, aes(x=RT, y=MZ) ) +
gem_point(size=1, aes(colour = Intensity), alpha=0.25) +
theme_minimal() +
scale_colour_gradient(low = "blue", high = "yellow")
```
Or visualize a particular ms2 spectrum:

```r
library(ggplot2)

spectra = py_to_r(exp$getSpectra())

# Collect all MS2 peak data in a list
peaks_ms2=list()
for (i in spectra) {
  if (i$getMSLevel()==2){
    peaks=do.call("cbind",i$get_peaks())
    peaks_ms2[c(i$getNativeID())]=data.frame(peaks)
  }
}
```

(continues on next page)
Alternatively, we could also have used `apply` to obtain the peak data, which is more idiomatic way of doing things for the R programming language:

```r
ms1 = sapply(spectra, function(x) x$MSLevel() == 1)
peaks = sapply(spectra[ms1], function(x) cbind(do.call("cbind", x$peaks()),x$RT()))
peaks = data.frame( do.call("rbind", peaks) )
```
**23.4.3 Iteration**

Iterating over pyopenms objects is not equal to iterating over R vectors or lists. Note that for many applications, there is a more efficient way to access data (such as `get_peaks` instead of iterating over individual peaks).

Therefore we can not directly apply the usual functions such as `apply()` and have to use `reticulate::iterate()` instead:

```r
spectrum = ropenms$MSSpectrum()
mz = seq(1500, 500, -100)
i = seq(10, 2000, length.out = length(mz))
spectrum$set_peaks(list(mz, i))

iterate(spectrum, function(x) {print(paste0("M/z: " , x$getMZ(), " Intensity: ", x
˓→$getIntensity()))}))
```

```
[1] "M/z :1500.0 Intensity: 10.0"
[1] "M/z :1400.0 Intensity: 209.0"
[1] "M/z :1300.0 Intensity: 408.0"
[1] "M/z :1200.0 Intensity: 607.0"
[1] "M/z :1100.0 Intensity: 806.0"
[1] "M/z :1000.0 Intensity: 1005.0"
[1] "M/z :900.0 Intensity: 1204.0"
[1] "M/z :800.0 Intensity: 1403.0"
[1] "M/z :700.0 Intensity: 1602.0"
[1] "M/z :600.0 Intensity: 1801.0"
[1] "M/z :500.0 Intensity: 2000.0"
```

or we can use a for-loop (note that we use zero-based indices as custom in Python):

```r
for (i in seq(0, py_to_r(spectrum$size())-1)) {
  print(spectrum[i]$getMZ())
  print(spectrum[i]$getIntensity())
}
```
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 documentation 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, 3.6 or 3.7. This is important, else you get a cli that is different than the one used for building the Python executable, and pyOpenMS will crash on import. The OpenMS wiki has detailed information on building pyOpenMS on Windows.

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.

## 24.1 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.
25 Wrapping Workflow and wrapping new Classes

25.1 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.

25.2 How to wrap new methods in existing classes

Let's 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 MSExperiment 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
25.3 How to wrap new classes

25.3.1 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:

```python
from xxx import *
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 `wrap-doc` qualifier

25.3.2 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:

```python
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)
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.

### 25.3.3 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.

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

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 +
        # wrap-ignore
        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):
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.

### 25.3.4 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)

### 25.3.5 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.
 Indices and tables

• genindex
Index

B
Base64 encoding, 92
Binary encoding (mzML), 91

I
install, 5

N
numpress, 93

S
source, 5, 103