OpenMS Tutorial

Version: 2.2.0

Contents

1 Introduction

This tutorial gives an introduction to the OpenMS core data structures and algorithms. It is intended to allow for a quick start in writing your own applications based on the OpenMS framework.

The structure of this tutorial is similar to the modules of the class documentation. First, the basic concepts and data structures of OpenMS are explained. The next chapter is about the kernel data structures. These data structures represent the actual mass spectrometric data: raw data, peaks, spectra and maps. In the following chapters, the more sophisticated data structures and algorithms, e.g. those used for peak picking, feature finding and identification are presented.

All the example programs used in this tutorial, can be found in *doc/code_examples/*.

If you are looking for C++ literature, we recommend the following books:

- **C++:** C++ Primer, Effective C++
- **STL:** Generic Programming and the STL, Effective STL, The C++ Standard Library
- **Qt:** C++ GUI Programming with Qt 4

The following image shows the overall structure of OpenMS:

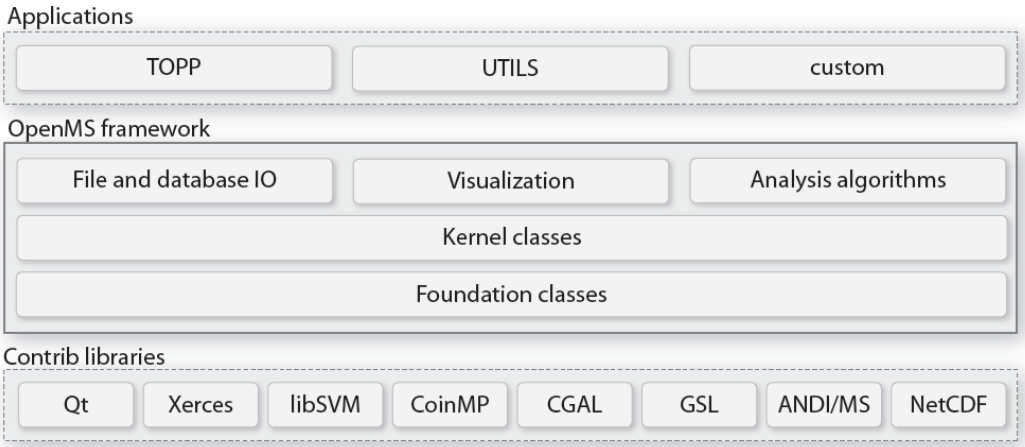

Figure 1: Overall design of OpenMS.

Without looking into the details of OpenMS the situation is very simple. Applications can be implemented using OpenMS, which in turn relies on several external libraries: *Qt* provides visualization and a platform abstraction layer. *Xerces* allows XML file parsing. *libSVM* is used for machine learning tasks. The *Computational Geometry Algorithms Library* (CGAL) provides data structures and algorithms for geometric computation.

OpenMS can itself be subdivided into several layers. At the very bottom are the foundation classes which implement low-level concepts and data structures. They include basic concepts (e.g. factory pattern, exception handling), basic data structures (e.g. string, points, ranges) and system-specific classes (e.g. file system, time). The kernel classes, which capture the actual MS data and metadata, are built upon the foundation classes. Finally, there is a layer of higher-level functionality that relies on the kernel classes. This layer contains file I/O supporting several file formats, data reduction functionality and all other analysis algorithms. The following terms for

MS-related data are used in this tutorial and the OpenMS class documentation:

- **raw data point**
  An unprocessed data point as measured by the instrument.
- **peak**
  Data point that is the result of some kind of peak detection algorithm. Peaks are often referred to as *sticks* or *centroided data* as well.
- **spectrum / scan**
  A mass spectrum containing raw data points (*raw spectrum*) or peaks (*peak spectrum*).

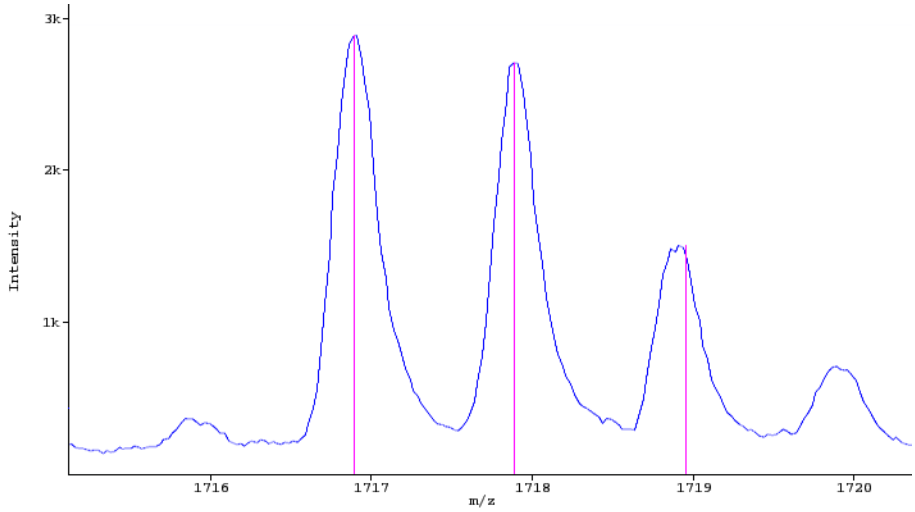

Figure 2: Part of a raw spectrum (blue) with three peaks (red)

- **map**
  A collection of spectra generated by a HPLC-MS experiment. Depending on what kinds of spectra are contained, we use the terms *raw map* or *peak map*. Often a map is also referred to as an *experiment*.
- **feature**
  The signal caused by a chemical entity detected in an HPLC-MS experiment, typically a peptide.

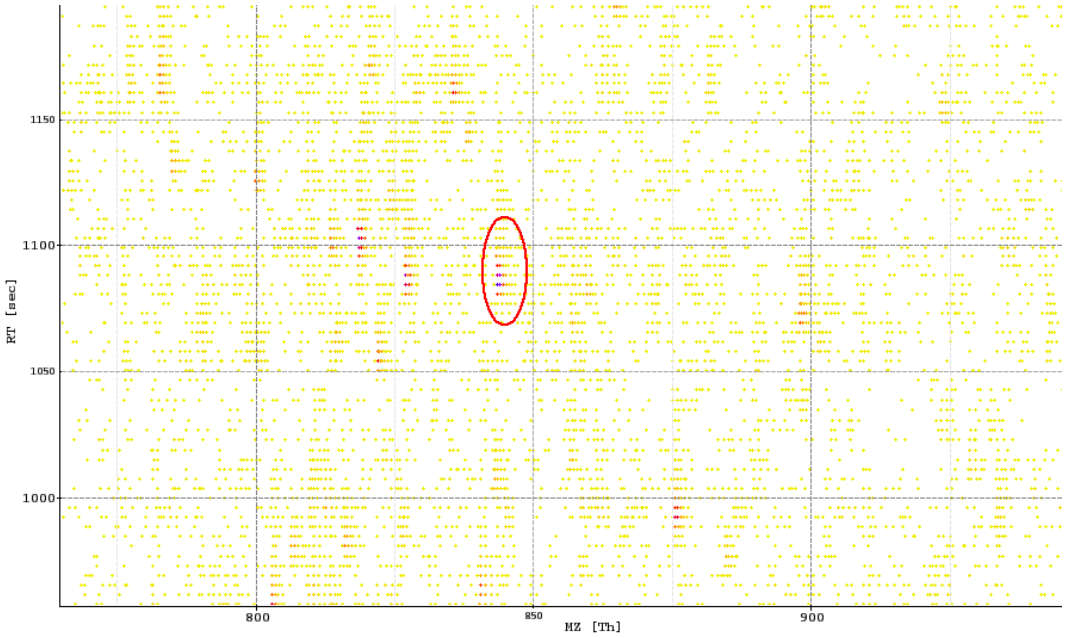

Figure 3: Peak map with a marked feature (red)

2 Basic classes and concepts

This chapter covers some very basic concepts needed to understand OpenMS code. It describes OpenMS primitive types, namespaces, exceptions and important preprocessor macros. The classes described in this section can be found in the *CONCEPT* folder.

2.1 Basic data types

OpenMS has its own names for the C++ primitive types. The integer types of OpenMS are *Int* (int) and *UInt* (unsigned int). For floating point numbers, no such names exist and the primitives float and double should be used.

These and more types are defined in *OpenMS/CONCEPT/Types.h*. The typeAsString() function can be used to find out the actual type of an object, e.g. if typedefs are used.

2.2 The OpenMS namespace

The main classes of OpenMS are implemented in the namespace *OpenMS*. There are several sub-namespaces to the *OpenMS* namespace. The most important ones are:

- *OpenMS::Constants* contains nature constants.
- *OpenMS::Math* contains math functions and classes.
- *OpenMS::Exception* contains the OpenMS exceptions.
- *OpenMS::Internal* contains certain auxiliary classes that are typically used by only one class of the *OpenMS* namespace and not by the user directly.

There are several more namespaces. For a detailed description have a look at the class documentation.

2.3 Exception handling in OpenMS

All exceptions are defined in the namespace *OpenMS::Exception*. The Base class for all OpenMS exceptions is *Base*. This base class provides three members for storing the source file, the line number and the function name where the exception occurred. All derived exceptions provide a constructor that takes at least these arguments. The following code snippet shows the handling of an index overflow:

```
// header
void someMethod(UInt index);

// C file
  void someMethod(UInt index)
  {
    if (index >= size())
    {
      throw Exception::IndexOverflow(__FILE__, __LINE__, __PRETTY_FUNCTION__, index, size()-1);
    }
    // do something
  };
```

Note the first three arguments given to the constructor: ***FILE*** and ***LINE*** are built-in preprocessor macros that hold the file name and the line number. ***PRETTY_FUNCTION*** is replaced by the GNU g++ compiler with the demangled name of the current function (including the class name and argument types). For other compilers we define it as "<unknown>". For an index overflow exception, there are two further arguments: the invalid index and the maximum allowed index.

The file name, line number and function name are very useful in debugging. However, OpenMS also implements its own exception handler which allows to turn each uncaught exception into a segmentation fault. With gcc this mechanism allows developers to trace the source of an exception with a debugger more effectively. To use this feature, set the environment variable *OPENMS_DUMP_CORE*. For Visual Studio you should set a breakpoint in GlobalExceptionHandler::newHandler() in Exception.cpp, otherwise you might loose the stacktrace to pinpoint the initial exception.

2.4 Condition macros

In order to enforce algorithmic invariants, the two preprocessor macros *OPENMS_PRECONDITION* and *OPE↩ NMS_POSTCONDITION* are provided. These macros are enabled only if debug info is enabled and optimization is disabled in *cmake*. Otherwise they are removed by the preprocessor, so they won't cost any performance.

The macros throw Exception::Precondition or Exception::Postcondition respectively if the condition fails. The example from section Exception handling in OpenMS could have been implemented like that:

```
void someMethod(UInt index)
{
  OPENMS_PRECONDITION(index < size(),"Precondition not met!");
  //do something
};
```

This section contains a short introduction to three data structures you will definitely need when programming with OpenMS. The data structures module of the class documentation contains many more classes, which are not mentioned here in detail. The classes described in this section can be found in the *DATASTRUCTURES* folder.

2.5 The OpenMS string implementation

The OpenMS string implementation *String* is based on the STL *std::string*. In order to make the OpenMS string class more convenient, a lot of methods have been implemented in addition to the methods provided by the base class. A selection of the added functionality is given here:

- Checking for a substring (suffix, prefix, substring, char)
- Extracting a substring (suffix, prefix, substring)
- Trimming (left, right, both sides)
- Concatenation of string and other primitive types with *operator+*
- Construction from QString and conversion to QString

2.6 D-dimensional coordinates

Many OpenMS classes, especially the kernel classes, need to store some kind of d-dimensional coordinates. The template class *DPosition* is used for that purpose. The interface of DPosition is pretty straightforward. The operator[] is used to access the coordinate of the different dimensions. The dimensionality is stored in the enum value *DIMENSION*. The following example (Tutorial_DPosition.cpp) shows how to print a DPosition to the standard output stream.

First we need to include the header file for *DPosition* and *iostream*. Then we import all the OpenMS symbols to the scope with the *using* directive.

```
#include <OpenMS/DATASTRUCTURES/DPosition.h>
#include <iostream>

using namespace OpenMS;
```

The first commands in the main method initialize a 2-dimensional *DPosition* :

```
Int main()
{
  DPosition<2> pos;
  pos[0] = 8.15;
  pos[1] = 47.11;
```

Finally we print the content of the DPosition to the standard output stream:

```
  for (Size i = 0; i < DPosition<2>::DIMENSION; ++i)
  {
    std::cout << "Dimension " << i << ": " << pos[i] << std::endl;
  }

  return 0;
} //end of main
```

The output of our first little OpenMS program is the following:

```
Dimension 0: 8.15
Dimension 1: 47.11
```

2.7 D-dimensional ranges

Another important data structure we need to look at in detail is *DRange*. It defines a d-dimensional, half-open interval through its two *DPosition* members. These members are accessed by the *minPosition()* and *maxPosition()* methods and can be set by the *setMin()* and *setMax()* methods.

DRange maintains the invariant that *minPosition()* is geometrically less or equal to *maxPosition()*, i.e. $minPosition()[x] \leq maxPosition()[x]$ for each dimension x. The following example (Tutorial_DRange.cpp) demonstrates this behaviour.

This time, we skip everything before the main method. In the main method, we create a range and assign values to *min* and *max*. Note that the minimum value of the first dimension is larger than the maximum value.

```
Int main()
{
  DRange<2> range;
  range.setMin(DPosition<2>(2.0, 3.0));
  range.setMax(DPosition<2>(1.0, 5.0));
```

Then we print the content of *range* :

```
  for (UInt i = 0; i < DRange<2>::DIMENSION; ++i)
  {
    std::cout << "min " << i << ": " << range.minPosition()[i] << std::endl;
    std::cout << "max " << i << ": " << range.maxPosition()[i] << std::endl;
  }

  return 0;
} //end of main
```

The output is:

```
min 0: 1
max 0: 1
min 1: 3
max 1: 5
```

As you can see, the minimum value of dimension one was adjusted in order to make the maximum of *1* conform with the invariant.

DIntervalBase is the closed interval counterpart (and base class) of *DRange*. Another class derived from *DIntervalBase* is *DBoundingBox*. It also represents a closed interval, but differs in the methods. Please have a look at the class documentation for details.

2.8 Param

Most algorithms of OpenMS and some of the TOPP tools have many parameters. The parameters are stored in instances of *Param*. This class is similar to a Windows INI file. The actual parameters (type, name and value) are stored in sections. Sections can contain parameters and sub-sections, which leads to a tree-like structure. The values are stored in *DataValue*.

Parameter names are given as a string including the sections and subsections in which ':' is used as a delimiter.

The following example (Tutorial_Param.cpp) shows how a file description is given.

```
Int main()
{
  Param param;

  param.setValue("file:name", "test.xml");
  param.setValue("file:size(MB)", 572.3);
  param.setValue("file:data:min_int", 0);
  param.setValue("file:data:max_int", 16459);

  cout << "Name    : " << (String)(param.getValue("file:name")) << endl;
  cout << "Size    : " << (float)(param.getValue("file:size(MB)")) << endl;
  cout << "Min int: " << (UInt)(param.getValue("file:data:min_int")) << endl;
  cout << "Max int: " << (UInt)(param.getValue("file:data:max_int")) << endl;
```

```
  return 0;
} //end of main
```

Especially for peptide/protein identification, a lot of data and data structures for chemical entities are needed. OpenMS offers classes for elements, formulas, peptides, etc. The classes described in this section can be found in the *CHEMISTRY* folder.

2.9 Elements

There is a representation of Elements implemented in OpenMS. The corresponding class is named *Element*. This class stores the relevant information about an element. The handling of the Elements is done by the class ElementDB, which is implemented as a singleton. This means there is only one instance of the class in OpenMS. This is straightforward because the Elements do not change during execution. Data stored in an Element spans its name, symbol, atomic weight, and isotope distribution beside others.

Example (Tutorial_Element.cpp):

```
const ElementDB * db = ElementDB::getInstance();

Element carbon = *db->getElement("Carbon"); // .getResidue("C") would also be ok

cout << carbon.getName() << " "
     << carbon.getSymbol() << " "
     << carbon.getMonoWeight() << " "
     << carbon.getAverageWeight() << endl;
```

Elements can be accessed by the *ElementDB* class. As it is implemented as a singleton, only a pointer of the singleton can be used, via *getInstance()*. The example program writes the following output to the console.

```
Carbon C 12 12.0107
```

2.10 EmpiricalFormula

The Elements described in the section above can be combined to empirical formulas. Application are the exact weights of molecules, like peptides and their isotopic distributions. The class supports a large number of operations like addition and subtraction. A simple example is given in the next few lines of code.

Example (Tutorial_EmpiricalFormula.cpp):

```
EmpiricalFormula methanol("CH3OH"), water("H2O");

EmpiricalFormula sum = methanol + water;

const Element * carbon = ElementDB::getInstance()->getElement("Carbon");

cout << sum << " "
     << sum.getNumberOf(carbon) << " "
     << sum.getAverageWeight() << endl;
```

Two instances of empirical formula are created. They are summed up, and some information about the new formula is printed to the terminal. The next lines show how to create and handle a isotopic distribution of a given formula.

```
IsotopeDistribution iso_dist = sum.getIsotopeDistribution(3);

for (IsotopeDistribution::ConstIterator it = iso_dist.begin(); it != iso_dist.end(); ++it)
{
  cout << it->first << " " << it->second << endl;
}
```

The isotopic distribution can be simply accessed by the *getIsotopeDistribution()* function. The parameter of this function describes how many isotopes should be reported. In our case, 3 are enough, as the following numbers get very small. On larger molecules, or when one want to have the exact distribution, this number can be set much higher. The output of the code snipped might look like this.

```
O2CH6 1 50.0571
50 0.98387
51 0.0120698
52 0.00406
```

2.11 Residue

A 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.

Example (Tutorial_Residue.cpp):

```
const ResidueDB * res_db = ResidueDB::getInstance();

Residue lys = *res_db->getResidue("Lysine"); // .getResidue("K") would also be ok

cout << lys.getName() << " "
     << lys.getThreeLetterCode() << " "
     << lys.getOneLetterCode() << " "
     << lys.getAverageWeight() << endl;
```

This small example show how to create a instance of ResidueDB were all Residues are stored in. The amino acids themselves can be accessed via the getResidue function. ResidueDB reads its amino acid and modification data from share/OpenMS/CHEMISTRY/.

The output of the example would look like this

```
Lysine LYS K 146.188
```

2.12 AASequence

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

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

A very simple example of handling amino acid sequence with AASequence is given in the next few lines.

Example (Tutorial_AASequence.cpp):

```
AASequence seq = AASequence::fromString("DFPIANGER");

AASequence prefix(seq.getPrefix(4));
AASequence suffix(seq.getSuffix(5));

cout << seq << " "
     << prefix << " "
     << suffix << " "
     << seq.getAverageWeight() << endl;
```

Not only the prefix, suffix and subsequence accession is supported, but also most of the features of Empirical↩ Formulas and Residues given above. Additionally, a number of predicates like hasSuffix are supported. The output of the code snippet looks like this.

```
DFPIANGER DFPI ANGER 1018.08
```

2.13 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.

Example (Tutorial_TheoreticalSpectrumGenerator.cpp)

```
TheoreticalSpectrumGenerator tsg;
PeakSpectrum spec1, spec2;
AASequence peptide = AASequence::fromString("DFPIANGER");

// standard behavior is adding b- and y-ions of charge 1
Param p;
p.setValue("add_b_ions", "false", "Add peaks of b-ions to the spectrum");
tsg.setParameters(p);
tsg.getSpectrum(spec1, peptide, 1, 1);

p.setValue("add_b_ions", "true", "Add peaks of a-ions to the spectrum");
tsg.setParameters(p);
tsg.getSpectrum(spec2, peptide, 1, 2);

cout << "Spectrum 1 has " << spec1.size() << " peaks. " << endl;
cout << "Spectrum 2 has " << spec2.size() << " peaks. " << endl;
```

The example shows how to put peaks of a certain type, y-ions in this case, into a spectrum. Spectrum 2 is filled with a complete spectrum of all peaks (a-, b-, y-ions and losses). The TheoreticalSpectrumGenerator has many parameters which have a detailed description located in the class documentation. The output of the program looks like the following two lines.

```
Spectrum 1 has 8 peaks.
Spectrum 2 has 32 peaks.
```

The OpenMS kernel contains the data structures that store the actual MS data, i.e. raw data points, peaks, features, spectra, maps. The classes described in this section can be found in the *KERNEL* folder.

2.14 Raw data point, Peak, Feature, ...

In general, there are three types of data points: raw data points, peaks and picked peaks. Raw data points provide members to store position (mass-to-charge ratio, retention time, ...) and intensity. Peaks are derived from raw data points and add an interface to store meta information. Picked peaks are derived from peaks and have additional members for peak shape information: charge, width, signal-to-noise ratio and many more.

The kernel data points exist in three versions: one-dimensional, two-dimensional and d-dimensional.

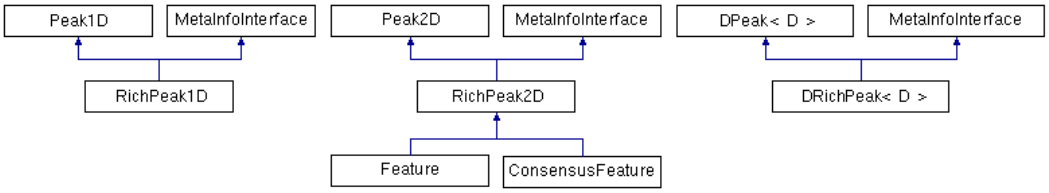

Figure 4: Data structures for MS data points

one-dimensional data points

The one-dimensional data points are most important, the two-dimensional and d-dimensional data points are needed rarely. The base class of the one-dimensional data points is *Peak1D*. It provides members to store the mass-to-charge ratio (*getMZ* and *setMZ*) and the intensity (*getIntensity* and *setIntensity*).
RichPeak1D is derived from *Peak1D* and adds an interface for metadata (see MetaInfo).

two-dimensional data points

The two-dimensional data points are needed when geometry algorithms are applied to the data points. A special case is the *Feature* class, which needs a two-dimensional position (m/z and RT).
The base class of the two-dimensional data points is *Peak2D*. It provides the same interface as *Peak1D* and additional members for the retention time (*getRT* and *setRT*).
RichPeak2D is derived from *Peak2D* and adds an interface for metadata.
Feature is derived from *RichPeak2D* and adds information about the convex hull of the feature, fitting quality and so on.

d-dimensional data points

The d-dimensional data points are needed only in special cases, e.g. in template classes that must operate on any number of dimensions.
The base class of the d-dimensional data points is *DPeak*. The methods to access the position are *getPosition* and *setPosition*.
Note that the one-dimensional and two-dimensional data points also have the methods *getPosition* and *set↩ Position*. They are needed in order to be able to write algorithms that can operate on all data point types. It is, however, recommended not to use these members unless you really write such a generic algorithm.

2.15 Spectra

The most important container for raw data and peaks is *MSSpectrum*. It is a template class that takes the peak type as template argument. The default peak type is *RichPeak1D*. Possible other peak types are classes derived from *Peak1D* or classes providing the same interface.
MSSpectrum is a container for 1-dimensional peak data. It is derived from *SpectrumSettings*, a container for the

meta data of a spectrum. Here, only MS data handling is explained, *SpectrumSettings* is described in section Meta data of a spectrum.

In the following example (Tutorial_MSSpectrum.cpp) 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:

```
Int main()
{
  MSSpectrum<> spectrum;
  Peak1D peak;

  for (float mz = 1500.0; mz >= 500; mz -= 100.0)
  {
    peak.setMZ(mz);
    spectrum.push_back(peak);
  }
```

Then we sort the peaks according to ascending mass-to-charge ratio.

```
  spectrum.sortByPosition();
```

Finally we print the peak positions of those peaks between 800 and 1000 Thomson. For printing all the peaks in the spectrum, we simply would have used the STL-conform methods *begin()* and *end()*.

```
  MSSpectrum<>::Iterator it;
  for (it = spectrum.MZBegin(800.0); it != spectrum.MZEnd(1000.0); ++it)
  {
    cout << it->getMZ() << endl;
  }

  return 0;
} //end of main
```

Typedefs

For convenience, the following type definitions are defined in *OpenMS/KERNEL/StandardTypes.h*.

```
typedef MSSpectrum<RichPeak1D> RichPeakSpectrum;
typedef MSSpectrum<Peak1D> PeakSpectrum;
```

2.16 Maps

Although raw data maps, peak maps and feature maps are conceptually very similar. They are stored in different data types. For raw data and peak maps, the default container is *MSExperiment*, which is an array of *MSSpectrum* instances. Just as *MSSpectrum* it is a template class with the peak type as template parameter.

In contrast to raw data and peak maps, feature maps are no collection of one-dimensional spectra, but an array of two-dimensional *Feature* instances. The main data structure for feature maps is called *FeatureMap*.

Although *MSExperiment* and *FeatureMap* differ in the data they store, they also have things in common. Both store meta data that is valid for the whole map, i.e. sample description and instrument description. This data is stored in the common base class *ExperimentalSettings*.

MSExperiment

The following figure shows the big picture of the kernel data structures. *MSExperiment* is derived from *ExperimentalSettings* (meta data of the experiment) and contains two data vectors, available as *vector<MSSpectrum>* and *vector<MSChromatogram>*. The one-dimensional spectrum *MSSpectrum* is derived from SpectrumSettings (meta data of a spectrum.

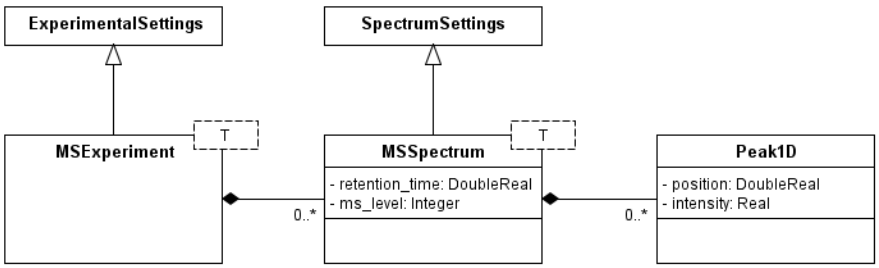

Figure 5: Overview of the main kernel data structures

The following example program (Tutorial_MSExperiment.cpp) creates a *MSExperiment* containing four *MS↩ Spectrum* instances. Then it iterates over an area and prints the peak positions in the area:

First we create the spectra in a for-loop and set the retention time and MS level. Survey scans have a MS level of 1, MS/MS scans would have a MS level of 2, and so on.

```
Int main()
{
  PeakMap exp;

  for (Size i = 0; i < 4; ++i)
  {
    PeakSpectrum spectrum;
    spectrum.setRT(i);
    spectrum.setMSLevel(1);
```

Then we fill each spectrum with several peaks. As all spectra would have the same peaks otherwise, we add the retention time to the mass-to-charge ratio of each peak.

```
    for (float mz = 500.0; mz <= 900; mz += 100.0)
    {
      Peak1D peak;
      peak.setMZ(mz + i);
      spectrum.push_back(peak);
    }
    exp.addSpectrum(spectrum);
  } //end of creation
```

Finally, we iterate over the RT range (2,3) and the m/z range (603,802) and print the peak positions.

```
  for (PeakMap::AreaIterator it = exp.areaBegin(2.0, 3.0, 603.0, 802.0); it != exp.areaEnd(); ++it)
  {
    cout << it.getRT() << " - " << it->getMZ() << endl;
  }
```

The output of this loop is:

```
2 - 702
2 - 802
3 - 603
3 - 703
```

For printing all the peaks in the experiment, we could have used the STL-iterators of the experiment to iterate over the spectra and the STL-iterators of the spectra to iterate over the peaks:

```
  for (PeakMap::Iterator s_it = exp.begin(); s_it != exp.end(); ++s_it)
  {
    for (PeakSpectrum::Iterator p_it = s_it->begin(); p_it != s_it->end(); ++p_it)
    {
```

```
      cout << s_it->getRT() << " - " << p_it->getMZ() << endl;
    }
  }

  return 0;
} //end of main
```

FeatureMap

FeatureMap, the container for features, is simply a *vector<Feature>*. Additionally, it is derived from *Experimental↩ Settings*, to store the meta information. Just like *MSExperiment*, it is a template class. It takes the feature type as template argument.

The following example (Tutorial_FeatureMap.cpp) shows how to insert two features into a map and iterate over the features.

```
Int main()
{
  FeatureMap map;

  Feature feature;
  feature.setRT(15.0);
  feature.setMZ(571.3);
  map.push_back(feature); //append feature 1
  feature.setRT(23.3);
  feature.setMZ(1311.3);
  map.push_back(feature); //append feature 2


  for (FeatureMap::Iterator it = map.begin(); it != map.end(); ++it)
  {
    cout << it->getRT() << " - " << it->getMZ() << endl;
  }

  return 0;
} //end of main
```

RangeManager

All peak and feature containers (*MSSpectrum*, *MSExperiment*, *FeatureMap*) are also derived from *Range↩ Manager*. This class facilitates the handling of MS data ranges. It allows to calculate and store both the position range and the intensity range of the container.

The following example (Tutorial_RangeManager.cpp) shows the functionality of the class *RangeManger* using a *FeatureMap*. First a *FeatureMap* with two features is created, then the ranges are calculated and printed:

```
Int main()
{
  FeatureMap map;

  Feature feature;
  feature.setIntensity(461.3f);
  feature.setRT(15.0);
  feature.setMZ(571.3);
  map.push_back(feature);
  feature.setIntensity(12213.5f);
  feature.setRT(23.3);
  feature.setMZ(1311.3);
  map.push_back(feature);

  //calculate the ranges
  map.updateRanges();

  cout << "Int: " << map.getMinInt() << " - " << map.getMaxInt() << endl;
  cout << "RT:  " << map.getMin()[0] << " - " << map.getMax()[0] << endl;
  cout << "m/z: " << map.getMin()[1] << " - " << map.getMax()[1] << endl;

  return 0;
} //end of main
```

The output of this program is:

```
Int: 461.3 - 12213.5
RT:  15 - 23.3
m/z: 571.3 - 1311.3
```

The meta informations about an HPLC-MS experiment are stored in *ExperimentalSettings* and *Spectrum↩ Settings*. All information that is not covered by these classes can be stored in the type-name-value data structure *MetaInfo*. All classes described in this section can be found in the *METADATA* folder.

2.17 MetaInfo

DataValue is a data structure that can store any numerical or string information. It also supports casting of the stored value back to its original type.

MetaInfo is used to easily store information of any type, that does not fit into the other classes. It implements type-name-value triplets. The main data structure is an associative container that stores *DataValue* instances as values associated to string keys. Internally, the string keys are converted to integer keys for performance reasons i.e. a *map<UInt,DataValue>* is used.

The *MetaInfoRegistry* associates the string keys used in *MetaValue* with the integer values that are used for internal storage. The *MetaInfoRegistry* is a singleton.

If you want a class to have a *MetaInfo* member, simply derive it from *MetaInfoInterface*. This class provides a *MetaInfo* member and the interface to access it.

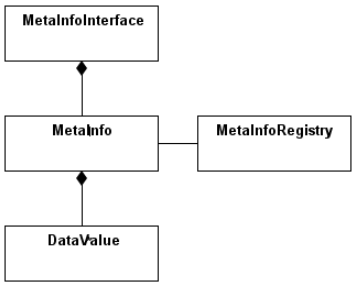

Figure 6: The classes involved in meta information storage

The following example (Tutorial_MetaInfo.cpp) shows how to use *Metadata*. We can simply set values for the string keys, and *setMetaValue* registers these names automatically. In order to access the values, we can either use the registered name or the index of the name. The *getMetaValue* method returns a *DataValue*, which has to be casted to the right type. If you do not know the type, you can use the *DataValue::valueType()* method.

```
Int main()
{
  MetaInfoInterface info;

  //insert meta data
  info.setMetaValue("color", String("#ff0000"));
  info.setMetaValue("id", 112131415);

  //access id by index
  UInt id_index = info.metaRegistry().getIndex("id");
  cout << "id   : " << (UInt)(info.getMetaValue(id_index)) << endl;
  //access color by name
  cout << "color: " << (String)(info.getMetaValue("color")) << endl;

  return 0;
} //end of main
```

2.18 Meta data of a map

This class holds meta information about the experiment that is valid for the whole experiment:

- sample description

- source files
- contact persons
- MS instrument
- HPLC settings
- protein identifications

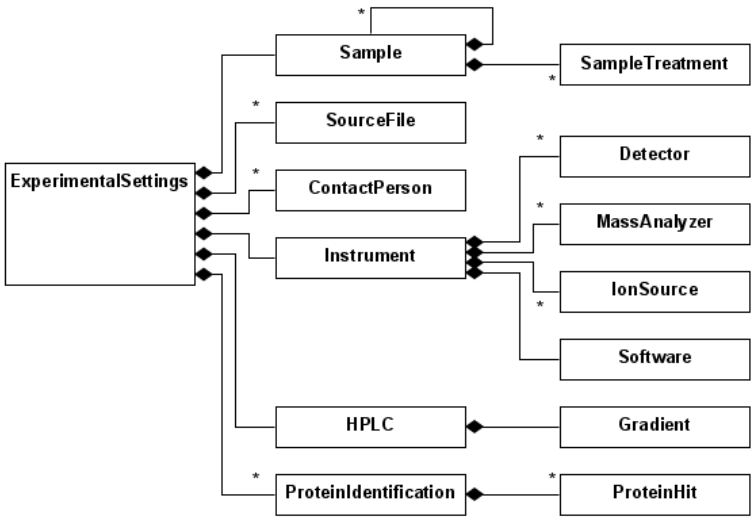

Figure 7: Map meta information

2.19 Meta data of a spectrum

This class contains meta information about settings specific to one spectrum:

- spectrum-specific instrument settings
- source file
- information on the acquisition
- precursor information (e.g. for MS/MS spectra)
- product information (e.g. for SRM spectra)
- processing performed on the data
- peptide identifications

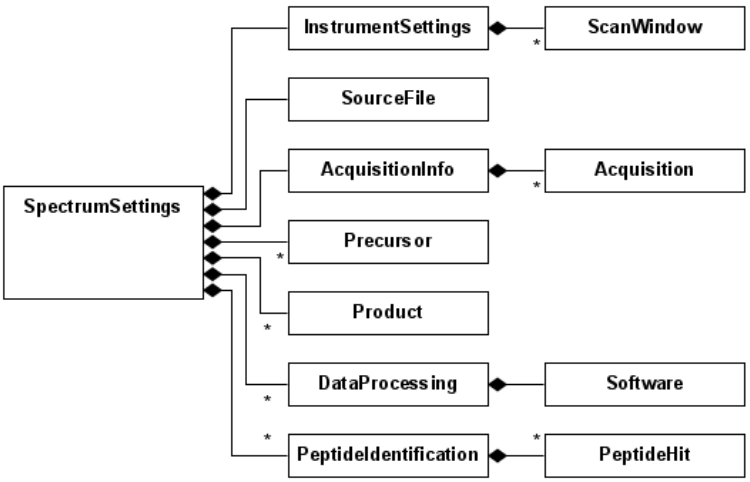

Figure 8: Spectrum meta information

2.20 Meta data associated to peaks

If you want to annotate the peaks or raw data points in your spectra with meta information, there are three different ways to do this with different advantages and disadvantages.

If each peak is annotated with the same type of information (e.g. width of a peak):

- Use the meta data arrays provided by MSSpectrum (recommended)
  - Advantages: Independent of peak type, information automatically stored in mzML files
  - Disadvantages: Information can be accessed through index only
- Derive a new peak type that contains members for the additional information
  - Advantages: Very fast
  - Disadvantages: Information not automatically stored in mzML files, Custom peak type are not supported by all algorithms

If you need to annotate only a small subset of the peaks with meta information:

- Use the MetaInfoInterface of RichPeak1D
  - Advantages: Each peak can be annotated with individual information
  - Disadvantages: Quite slow, Information not automatically stored in mzML files

All classes for file IO can be found in the *FORMAT* folder. For most file formats, you can use the generic interfaces described in the section "File adapter classes" but for some applications (e.g. where CPU or memory usage are of concern), also consult the specific interfaces that OpenMS provides for the mzML data format (see section "File I/O for MzML").

2.21 File adapter classes

The interface of most file adapter classes is very similar. They implement a *load* and a *store* method, that take a file name and the appropriate data structure. Usually these methods expect a complete in-memory representation of an MS run (see below for other ways to access MS data).

The following example (Tutorial_FileIO.cpp) demonstrates the use of *OpenMS::MzMLFile* and *OpenMS::MzXMLFile* to convert one format into another using *OpenMS::MSExperiment* to hold the temporary data:

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  MzXMLFile mzxml;
  MzMLFile mzml;

  // temporary data storage
  PeakMap map;

  // convert MzXML to MzML
  mzxml.load(tutorial_data_path + "/data/Tutorial_FileIO.mzXML", map);
  mzml.store("Tutorial_FileIO.mzML", map);

  return 0;
} //end of main
```

FileHandler

In order to make the handling of different file types easier, the class *FileHandler* can be used. It loads a file into the appropriate data structure independently of the file type. The file type is determined from the file extension or the file contents:

```
MSExperiment in;
FileHandler handler();
handler.loadExperiment("input.mzML",in);
```

2.22 File I/O for MzML

For MzML, several additional interfaces exist which make some data processing tasks easier. Specifically, it is not always feasible to load the complete data of an LC-MS/MS run into memory and for these cases, special interfaces exist. One such interface is the OpenMS::OnDiscMSExperiment which abstracts an mzML file that contains an index. The spectra and chromatogram can be obtained by calling getSpectrum or getChromatogram .

2.22.1 Indexed mzML

The following example (Tutorial_FileIO_mzML.cpp) demonstrates the use of OpenMS::IndexedMzMLFileLoader to obtain a representation of a mass spectrometric experiment in the form of an OnDiscMSExperiment:

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  IndexedMzMLFileLoader imzml;

  // load data from an indexed MzML file
  OnDiscPeakMap map;
  imzml.load(tutorial_data_path + "/data/Tutorial_FileIO_indexed.mzML", map);
```

```
  // Get the first spectrum in memory, do some constant (non-changing) data processing
  MSSpectrum<> s = map.getSpectrum(0);
  std::cout << "There are " << map.getNrSpectra() << " spectra in the input file." << std::endl;
  std::cout << "The first spectrum has " << s.size() << " peaks." << std::endl;

  // store the (unmodified) data in a different file
  imzml.store("Tutorial_FileIO_output.mzML", map);

  return 0;
} //end of main
```

2.22.2 Sequential Reading/Writing of mzML

In addition, the OpenMS::MzMLFile also offers the OpenMS::MzMLFile::transform function which allows to load an MzML file and while loading the spectra and chromatograms are fed into the provided MSDataConsumer (which inherits from the interface IMSDataConsumer). This can allow for memory-efficient implementation of algorithms which only need to access spectra and chromatograms sequentially. Of course several MSDataConsumer implementations can be chained by writing an implementation which hands the results of its computation directly to the next consumer. One common usage scenario uses an OpenMS::PlainMSDataWritingConsumer or an OpenMS::CachedMzMLConsumer as a final consumer which writes the data to disk after usage. In this way, only the memory to keep a single spectrum in memory is used and algorithms that are constant in memory usage with respect to the amount of data to be processed can be implemented.

The following example (Tutorial_FileIO_Consumer.cpp) demonstrates the use of a consumer to obtain all spectra contained in a file sequentially and apply some data processing and then writing them to disk to them without ever loading the whole file into memory:

```
class TICWritingConsumer : public MSDataWritingConsumer
{
  // Inheriting from MSDataWritingConsumer allows to change the data before
  // they are written to disk (to "filename") using the processSpectrum_ and
  // processChromatogram_ functions.
public:
  double TIC;
  int nr_spectra;

  // Create new consumer, set TIC to zero
  TICWritingConsumer(String filename) : MSDataWritingConsumer(filename)
    { TIC = 0.0; nr_spectra = 0;}

  // Add a data processing step for spectra before they are written to disk
  void processSpectrum_(MSDataWritingConsumer::SpectrumType & s)
  {
    for (Size i = 0; i < s.size(); i++) { TIC += s[i].getIntensity(); }
    nr_spectra++;
  }
  // Empty chromatogram data processing
  void processChromatogram_(MSDataWritingConsumer::ChromatogramType& /* c */) {}
};

int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  // Create the consumer, set output file name, transform
  TICWritingConsumer * consumer = new TICWritingConsumer("Tutorial_FileIO_output.mzML");
  MzMLFile().transform(tutorial_data_path + "/data/Tutorial_FileIO_indexed.mzML", consumer);

  std::cout << "There are " << consumer->nr_spectra << " spectra in the input file." << std::endl;
  std::cout << "The total ion current is " << consumer->TIC << std::endl;
  delete consumer;

  return 0;
} //end of main
```

2.22.3 Summary

However, if random access to the data is needed, then the consumer-based approach will not suffice. Here one can either use the OpenMS::OnDiscMSExperiment approach as discussed above or for very fast access to individual spectra (no parsing of base64-encoded needed) one can first cache the data to disk using the OpenMS::CachedmzML class.

2.23 PeakFileOptions

In order to have more control over loading data from files, most adapters can be configured using *PeakFileOptions*. The following options are available:

- only a specific retention time range is loaded
- only a specific mass-to.charge ratio range is loaded
- only a specific intensity range is loaded
- only spectra with a given MS level are loaded
- only meta data of the whole experiment is loaded (*ExperimentalSettings*)

3 Algorithms

OpenMS offers several filters for the reduction of noise and baseline which disturb LC-MS measurements. These filters work spectra-wise and can therefore be applied to a whole raw data map as well as to a single raw spectrum. All filters offer functions for the filtering of raw data containers (e.g. *PeakSpectrum*) "filter" as well as functions for the processing of a collection of raw data containers (e.g. *PeakMap*) "filterExperiment". The functions "filter" and "filterExperiment" can both be invoked with an input container along with an output container or with iterators that define a range on the input container along with an output container. The classes described in this section can be found in the *FILTERING* folder.

3.1 Baseline filters

Baseline reduction can be performed by the *TopHatFilter*. The top-hat filter is a morphological filter which uses the basic morphological operations "erosion" and "dilatation" to remove the baseline in raw data. Because both operations are implemented as described by Van Herk the top-hat filter expects equally spaced raw data points. If your data is not uniform yet, please use the *LinearResampler* to generate equally spaced data.

The *TopHatFilter* removes signal structures in the raw data which are broader than the size of the structuring element.

The following example (Tutorial_MorphologicalFilter.cpp) shows how to instantiate a tophat filter, set the length of the structuring element and remove the base line in a raw LC-MS map.

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  PeakMap exp;

  MzMLFile mzml_file;
  mzml_file.load(tutorial_data_path + "/data/Tutorial_MorphologicalFilter.mzML", exp);

  Param parameters;
  parameters.setValue("struc_elem_length", 1.0);
  parameters.setValue("struc_elem_unit", "Thomson");
  parameters.setValue("method", "tophat");

  MorphologicalFilter mf;
  mf.setParameters(parameters);

  mf.filterExperiment(exp);

  return 0;
} //end of main
```

Note

In order to remove the baseline, the width of the structuring element should be greater than the width of a peak.

3.2 Smoothing filters

We offer two smoothing filters to reduce noise in LC-MS measurements.

3.2.1 Gaussian filter

The class *GaussFilter* is a Gaussian filter. The wider the kernel width, the smoother the signal (the more detail information gets lost).

We show in the following example (Tutorial_GaussFilter.cpp) how to smooth a raw data map. The gaussian kernel width is set to 1 m/z.

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  PeakMap exp;

  MzMLFile mzdata_file;
  mzdata_file.load(tutorial_data_path + "/data/Tutorial_GaussFilter.mzML", exp);

  GaussFilter g;
  Param param;
  param.setValue("gaussian_width", 1.0);
  g.setParameters(param);

  g.filterExperiment(exp);

  return 0;
} //end of main
```

Note

Use a Gaussian filter kernel which has approximately the same width as your mass peaks.

3.2.2 Savitzky Golay filter

The Savitzky Golay filter is implemented in two ways *SavitzkyGolaySVDFilter* and *SavitzkyGolayQRFilter*. Both filters come to the same result but in most cases the *SavitzkyGolaySVDFilter* has a better run time. The Savitzky Golay filter works only on equally spaced data. If your data is not uniform use the *LinearResampler* to generate equally spaced data. The smoothing degree depends on two parameters: the frame size and the order of the polynomial used for smoothing. The frame size corresponds to the number of filter coefficients, so the width of the smoothing interval is given by frame_size∗spacing of the raw data. The bigger the frame size or the smaller the order, the smoother the signal (the more detail information gets lost!).

The following example (Tutorial_SavitzkyGolayFilter.cpp) shows how to use a *SavitzkyGolaySVDFilter* (the *SavitzkyGolayQRFilter* has the same interface) to smooth a single spectrum. The single raw data spectrum is loaded and resampled to uniform data with a spacing of 0.01 /m/z. The frame size of the Savitzky Golay filter is set to 21 data points and the polynomial order is set to 3. Afterwards the filter is applied to the resampled spectrum.

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  PeakSpectrum spectrum;

  DTAFile dta_file;
  dta_file.load(tutorial_data_path + "/data/Tutorial_SavitzkyGolayFilter.dta", spectrum);

  LinearResampler lr;
  Param param_lr;
  param_lr.setValue("spacing", 0.01);
  lr.setParameters(param_lr);
  lr.raster(spectrum);

  SavitzkyGolayFilter sg;
  Param param_sg;
  param_sg.setValue("frame_length", 21);
  param_sg.setValue("polynomial_order", 3);
  sg.setParameters(param_sg);
  sg.filter(spectrum);

  return 0;
} //end of main
```

3.3 Calibration

OpenMS offers methods for external and internal calibration of raw or peak data.

3.3.1 Internal Calibration

The InternalCalibration uses reference masses for calibration. At least two reference masses have to exist in each spectrum, otherwise it is not calibrated. The data to be calibrated can be raw data or already picked data. If we have raw data, a peak picking step is necessary. For the important peak picking parameters, have a look at the Peak picking section.

The following example (Tutorial_InternalCalibration.cpp) shows how to use the InternalCalibration for raw data. First the data and reference masses are loaded.

Then we set the important peak picking parameters and run the internal calibration:

3.3.2 TOF Calibration

The TOFCalibration uses calibrant spectra to convert a spectrum containing time-of-flight values into one with m/z values. For the calibrant spectra, the expected masses need to be known as well as the calibration constants in order to convert the calibrant spectra tof into m/z (determined by the instrument). Using the calibrant spectra's tof and m/z-values, first a quadratic curve fitting is done. The remaining error is estimated by a spline curve fitting. The quadratic function and the splines are used to determine the calibration equation for the conversion of the experimental data.

The following example (Tutorial_TOFCalibration.cpp) shows how to use the TOFCalibration for raw data. First the spectra and reference masses are loaded.

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;

  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  TOFCalibration ec;
  PeakMap exp_raw, calib_exp;
  MzMLFile mzml_file;
  mzml_file.load(tutorial_data_path + "/data/Tutorial_TOFCalibration_peak.mzML", calib_exp);
  mzml_file.load(tutorial_data_path + "/data/Tutorial_TOFCalibration_raw.mzML", exp_raw);

  vector<double> ref_masses;
  TextFile ref_file;
  ref_file.load(tutorial_data_path + "/data/Tutorial_TOFCalibration_masses.txt", true);
  for (TextFile::ConstIterator iter = ref_file.begin(); iter != ref_file.end(); ++iter)
  {
    ref_masses.push_back(String(iter->c_str()).toDouble());
  }
```

Then we set the calibration constants for the calibrant spectra.

```
  std::vector<double> ml1;
  ml1.push_back(418327.924993827);

  std::vector<double> ml2;
  ml2.push_back(253.645187196031);

  std::vector<double> ml3;
  ml3.push_back(-0.0414243465397252);

  ec.setML1s(ml1);
  ec.setML2s(ml2);
  ec.setML3s(ml3);
```

Finally, we set the important peak picking parameters and run the external calibration:

```
  Param param;
  param.setValue("PeakPicker:peak_width", 0.1);
  ec.setParameters(param);
  ec.pickAndCalibrate(calib_exp, exp_raw, ref_masses);

  return 0;
} //end of main
```

Data reduction in LC-MS analysis mostly consists of two steps. In the first step, called "peak picking", important information of the mass spectrometric peaks (e.g. peaks' mass centroid positions, their areas under curve and full-width-at-half-maxima) are extracted from the raw LC-MS data. The second data reduction step, called "feature finding", represents the quantitation of all peptides in a proteomic sample. Therefore, the signals in a LC-MS map caused by all charge and isotopic variants of the peptide are detected and summarized resulting in a list of compounds or features, each characterized by mass, retention time and abundance. The classes described in this section can be found in the *TRANSFORMATIONS* folder.

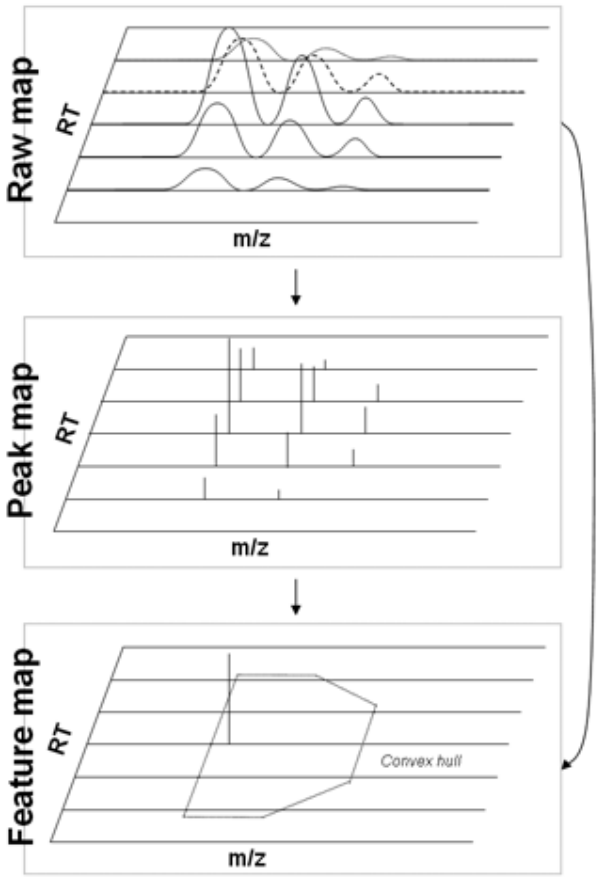

Figure 9: A peptide feature at different stages of data reduction.

3.4 Peak picking

For peak picking, the class *PeakPickerCWT* or *PeakPickerHiRes* is used. Because this class detects and extracts mass spectrometric peaks it is applicable to LC-MS as well as MALDI raw data.

The following example (Tutorial_PeakPickerCWT.cpp) shows how to open a raw map, initialize a PeakPickerCWT object, set the most important parameters (the scale of the wavelet, a peak's minimal height and FWHM), and start the peak picking process.

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  PeakMap exp_raw;
  PeakMap exp_picked;
```

```
  MzMLFile mzml_file;
  mzml_file.load(tutorial_data_path + "/data/Tutorial_PeakPickerCWT.mzML", exp_raw);

  PeakPickerCWT pp;
  Param param;
  param.setValue("peak_width", 0.1);
  pp.setParameters(param);

  pp.pickExperiment(exp_raw, exp_picked);
  exp_picked.updateRanges();

  cout << "\nMinimal fwhm of a mass spectrometric peak: " << (double)param.getValue("peak_width")
       << "\n\nNumber of picked peaks " << exp_picked.getSize() << std::endl;

  return 0;
} //end of main
```

The output of the program is:

```
Scale of the wavelet: 0.2
Minimal FWHM of a mass spectrometric peak: 0.1
Minimal intensity of a mass spectrometric peak 500

Number of picked peaks 14
```

Note

A rough standard value for the peak's scale is the average FWHM of a mass spectrometric peak.

3.5 Feature detection

The FeatureFinders implement different algorithms for the detection and quantitation of peptides from LC-MS maps. In contrast to the previous step (peak picking), we do not only search for pronounced signals (peak) in the LC-MS map but search explicitly for peptides which can be recognized by their isotopic pattern.

OpenMS offers different algorithms for this task. Details of how to apply them are given in the TOPP documentation. Please also refer to our publications on the OpenMS web page. TOPP contains multiple command line programs which allow to execute our algorithms without writing a single line of code.

But you can also write your own FeatureFinder application. This gives you more flexibility and is straightforward to do. A short example (Tutorial_FeatureFinder.cpp) is given below. First we need to instantiate the FeatureFinder, its parameters and the input/output data:

```
  FeatureFinder ff;
  // ... set parameters (e.g. from INI file)
  Param parameters;
  // ... set input data (e.g. from mzML file)
  PeakMap input;
  // ... set output data structure
```

Then we run the FeatureFinder. The first argument is the algorithm name (here 'simple'). Using the second and third parameter, the peak and feature data is handed to the algorithm. The fourth argument sets the parameters used by the algorithm.

```
  FeatureMap output;
  // ... set user-specified seeds, if needed
  FeatureMap seeds;

  ff.run("simple", input, output, parameters, seeds);
```

Now the FeatureMap is filled with the found features. OpenMS offers a number of map alignment algorithms.

The take several peak or feature maps and transform the retention time axis so that peak/feature positions become comparable.

The classes described in this section can be found in the *ANALYSIS/MAPMATCHING* folder.

All map alignment algorithms are derived from the common base class *MapAlignmentAlgorithm* and, thus, share a common interface. That is why only one example (Tutorial_MapAlignment.cpp) is shown here. Other algorithms can be work accordingly.

First, we load two feature maps:

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  FeatureMap reference;
  FeatureMap toAlign;

  FeatureXMLFile xml_file;
  xml_file.load(tutorial_data_path + "/data/Tutorial_MapAlignment_1.featureXML", reference);
  xml_file.load(tutorial_data_path + "/data/Tutorial_MapAlignment_2.featureXML", toAlign);
```

Then, we instantiate the algorithm and align the feature maps:

Finally, the aligned maps are written to files:

As an additional output the algorithms return one *TransformationDescription* per input file. This *Transformation↩ Description* describes the transformation that was applied to the retention times.

Note

In order to align peak maps the method *alignPeakMaps* has to be used.

Based on the features found during the Feature detection, quantitation can be performed. OpenMS offers a number of feature grouping algorithms. The take one or several feature maps and group feature in one map or across maps, depending on the algorithm.

The classes described in this section can be found in the *ANALYSIS/MAPMATCHING* folder.

All feature grouping algorithms are derived from the common base class *FeatureGroupingAlgorithm* and, thus, share a common interface. Currently two algorithms are implemented. One for isotope-labeled experiments with two labels and another for label-free quantitation.

3.6 Feature grouping for label-free quantitation

The first example shows the label-free quantitation (Tutorial_Unlabeled.cpp):

First, we load two feature maps:

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  vector<FeatureMap > maps;
  maps.resize(2);

  FeatureXMLFile feature_file;
  feature_file.load(tutorial_data_path + "/data/Tutorial_Unlabeled_1.featureXML", maps[0]);
  feature_file.load(tutorial_data_path + "/data/Tutorial_Unlabeled_2.featureXML", maps[1]);
```

In order to write the a valid output file, we need to set the input file names and sizes.

```
  ConsensusMap out;
  out.getFileDescriptions()[0].filename = "/data/Tutorial_Unlabeled_1.featureXML";
  out.getFileDescriptions()[0].size = maps[0].size();
  out.getFileDescriptions()[1].filename = "/data/Tutorial_Unlabeled_2.featureXML";
  out.getFileDescriptions()[1].size = maps[1].size();
```

Then, we instantiate the algorithm and group the features:

```
  FeatureGroupingAlgorithmUnlabeled algorithm;
  // ... set parameters
  algorithm.group(maps, out);
```

Finally, we store the grouped features in a consensusXML file.

```
  ConsensusXMLFile consensus_file;
  consensus_file.store("Tutorial_Unlabeled.consensusXML", out);

  return 0;
} //end of main
```

3.7 Feature grouping for isotope-labeled quantitation

The second example shows the isotope-labeled quantitation (Tutorial_Labeled.cpp):

First, we load the feature map:

```
int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  vector<FeatureMap > maps;
  maps.resize(1);

  FeatureXMLFile feature_file;
  feature_file.load(tutorial_data_path + "/data/Tutorial_Labeled.featureXML", maps[0]);
```

The isotope-labeled quantitation finds two types of features in the same map (heavy and light variant). So we add two map descriptions with the same file name to the output and set the labels accordingly:

```
  ConsensusMap out;
  out.getFileDescriptions()[0].filename = "data/Tutorial_Labeled.featureXML";
  out.getFileDescriptions()[0].size = maps[0].size();
  out.getFileDescriptions()[0].label = "light";
  out.getFileDescriptions()[1].filename = "data/Tutorial_Labeled.featureXML";
  out.getFileDescriptions()[1].size = maps[0].size();
  out.getFileDescriptions()[1].label = "heavy";
```

Then, we instantiate the algorithm and group the features:

```
  FeatureGroupingAlgorithmLabeled algorithm;
  // ... set parameters
  algorithm.group(maps, out);
```

Finally, we store the grouped features in a consensusXML file. In order to write a valid file, we need to set the input file names and sizes.

```
  ConsensusXMLFile consensus_file;
  consensus_file.store("Tutorial_Labeled.consensusXML", out);

  return 0;
} //end of main
```

4 Advanced tutorials

Visualization in OpenMS is based on Qt.

4.1 1D view

All types of peak or feature visualization share a common interface. So here only an example how to visualize a single spectrum is given (Tutorial_GUI_Spectrum1D.cpp).

First we need to create a *QApplication* in order to be able to use Qt widgets in out application.

```
Int main(int argc, const char ** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  QApplication app(argc, const_cast<char **>(argv));
```

Then we load a DTA file (the first command line argument of our application).

```
  PeakMap exp;
  exp.resize(1);
  DTAFile().load(tutorial_data_path + "/data/Tutorial_Spectrum1D.dta", exp[0]);
```

Then we create a widget for 1D visualization and hand over the data.

```
  LayerData::ExperimentSharedPtrType exp_sptr(new PeakMap(exp));
  Spectrum1DWidget * widget = new Spectrum1DWidget(Param(), 0);
  widget->canvas()->addLayer(exp_sptr);
  widget->show();
```

Finally we start the application.

```
  return app.exec();
} //end of main
```

4.2 Visual editing of parameters

Param objects are used to set algorithm parameters in OpenMS. In order to be able to visually edit them, the *ParamEditor* class can be used. The following example (Tutorial_GUI_ParamEditor.cpp) show how to use it.

We need to create a QApplication, load the data from a file (e.g. the parameters file of any TOPP tool), create the *ParamEditor* and execute the application:

```
Int main(int argc, const char** argv)
{
  if (argc < 2) return 1;
  // the path to the data should be given on the command line
  String tutorial_data_path(argv[1]);

  QApplication app(argc, const_cast<char**>(argv));

  Param param;
  ParamXMLFile paramFile;

  paramFile.load(tutorial_data_path + "/data/Tutorial_ParamEditor.ini", param);

  ParamEditor* editor = new ParamEditor(0);
  editor->load(param);
  editor->show();

  app.exec();
```

When it is closed, we store the result back to the *Param* object and then to the file.

```
  editor->store();
  paramFile.store("Tutorial_ParamEditor_out.ini", param);

  return 0;
} //end of main
```

In OpenMS, generic hierarchical clustering is available, the example (Tutorial_Clustering.cpp) shows how to build a rudimentary clustering pipeline.

4.3 Inputdata

All types of data can be clustered, as long as a SimilarityComparator for the type is provided. This Comparator has to produce a similarity measurement with the ()-operator in the range of [0,1] for each two elements of this type, so it can be transformed to a distance. Some SimilarityComparators are already implemented, e.g. the base class for the PeakSpectrum-type SimilarityComparator is OpenMS::PeakSpectrumCompareFunctor.

```
class LowLevelComparator
{
public:
  double operator()(const double first, const double second) const
  {
    double x, y;
    x = min(second, first);
    y = max(first, second);
    if ((y - x) > 1)
    {
      throw Exception::InvalidRange(__FILE__, __LINE__, OPENMS_PRETTY_FUNCTION);
    }
    return 1 - (y - x);
  }

}; // end of LowLevelComparator
```

This example of a SimilarityComparator is very basic and takes one-dimensional input of *doubles* in the range of [0,1]. Real input will generally be more complex and so has to be the corresponding SimilarityComparator. Note that similarity in the example is calculated by *1-distance*, whereas generally distance is obtained by getting the similarity and not the other way round.

4.4 Clustering

Clustering is conducted in the *OpenMS::ClusterHierarchical* class that offers an easy way to perform the clustering.

```
Int main()
{
  // data
  vector<double> data; // must be filled
  LowLevelComparator llc;
  CompleteLinkage sl;
  vector<BinaryTreeNode> tree;
  DistanceMatrix<float> dist; // will be filled
  ClusterHierarchical ch;
  ch.setThreshold(0.15);
```

The *ClusterHierarchical* functions will need at least these arguments, setting the threshold is optional (per default set to 1,0). The template-arguments have to be set to the type of clustered data and the type of Compare↩ Functor used. In this example double and LowLevelComparator.

```
  // clustering
  ch.cluster<double, LowLevelComparator>(data, llc, sl, tree, dist);
```

This function will create a hierarchical clustering up to the threshold. See Output.

4.5 Output

If known, at what threshold (see *OpenMS::ClusterHierarchical::cluster*) a reasonable clustering is produced, the setting of the right threshold can potentially speed up the clustering process. After exceeding the threshold, the

resulting tree (std::vector of OpenMS::BinaryTreeNode) is filled with dummy nodes. The tree represents the hierarchy of clusters by storing the stepwise merging process. It can eventually be transformed to a tree-representation in Newick-format and/or be analysed with other methods the OpenMS::ClusterAnalyzer class provides.

```
  ClusterAnalyzer ca;
  std::cout << ca.newickTree(tree) << std::endl;

  return 0;
} //end of main
```

So the output will look something like this (may actually vary since random numbers are used in this example)↩
:

```
( ( ( ( ( 0 , 1 ) , ( 2 , ( 7 , 8 ) ) ) , ( ( 3 , 10 ) , ( 4 , 5 ) ) ) , ( 6 , 9 ) ) , 11 )
```

For closer survey of the clustering process one can also view the whole hierarchy by viewing the tree in Newick-format with a tree viewer such as TreeViewX. A visualization of a particular cluster step (which gives rise to a certain partition of the data clustered) can be created with heatmaps (for example with gnuplot 4.3 heatmaps and the corresponding distance matrix). This tutorial will give you an overview of how to use the peak intensity

prediction (PIP). In general, PIP allows you to predict the peak intensity of a peptide relative to other peptides of the same abundance from its sequence alone. At the same time, this value allows to correct peak intensities for peptide-specific instrument sensitivity in a label-free quantitation application.

This method is still in an early phase: A proof of concept has been conducted and published in [1]. Peak intensities *can* be predicted with significant correlations, but application tests are yet to come.

4.6 Background

The sensitivity of a mass spectrometer depends on the analysed peptides, among other factors. This peptide-specific sensitivity causes peak heights of peptides with the same abundance to be generally different. PIP incorporates a model that maps peptide sequences to peptide-specific sensitivities.

4.7 Machine learning details

The incorporated model has been adapted with a Local Linear Map [2] - a machine learning algorithm that uses both supervised and unsupervised learning in its training, and which is fast and easy to implement. Better results can be achieved with other learning architectures [3], however, these are not implemented in this prototype stage yet.

4.8 About the training data

The model which the PIP module uses has been trained with data from a Bruker Ultraflex MALDI-TOF instrument. Details about these data can be found with [3]. A Pearson's squared correlation of 0.43 in ten-fold cross-validation and of 0.34 across datasets from the same instrument (but with different settings and operating persons) could be achieved. There is no experience yet about the performance across instruments. So we would be pleased if you could share your experience with the model incorporated in PIP applied to other datasets.

At this point, it is not possible to train a model with your own data, but it is a planned feature. It is as of yet unknown how similar peptide-specific sensitivities behave between different MALDI instruments.

4.9 How to use PIP

PIP lets you predict intensities using peptide sequences as input. The output values have been normalized to a mean of 0 and variance 1.

To **test** PIP with data from your instrument, MALDI spectra that contain only peptides of one protein can be used:

1. Normalize your peak intensities with the sum of only the peptide's peaks to make them comparable to other spectra.
2. Logarithmize the resulting values.
3. Center and normalize your peak intensities by variance (of course, multiple spectra should be used to find mean and variance), these value are referred to as *tI* in the following.
4. Predict the peptide's peak intensities (referred to as *pI* in the following)
5. Calculate the correlation between the *tI* and *pI*. If you calculate exp(log(tI) - pI), it should give 1 as a result in this test.

To calculate relative peptide abundance (relative to those of the other peptides in the mixture) from intensities of a peptide mixture using values predicted by PIP, do above steps 2. to 4. Then calculate the peptide level x = exp(log(tI) - pI). **!!!** The quantification with an actual protein mixture has never been tested with this model.

4.10 Example code

There is a usage example for the PeakIntensityPredictor class in `doc/code_examples/Tutorial_PeakIntensity↩ Predictor.cpp`.

Sequences of peptides to be predicted should be stored in a vector of AASequence instances:

```
//Create a vector for the predicted values that is large enough to hold them all
vector<AASequence> peptides;
peptides.push_back(AASequence::fromString("IVGLMPHPEHAVEK"));
peptides.push_back(AASequence::fromString("LADNISNAMQGISEATEPR"));
peptides.push_back(AASequence::fromString("ELDHSDTIEVIVNPEDIDYDAASEQAR"));
peptides.push_back(AASequence::fromString("AVDTVR"));
peptides.push_back(AASequence::fromString("AAWQVK"));
peptides.push_back(AASequence::fromString("FLGTQGR"));
peptides.push_back(AASequence::fromString("NYPSDWSDVDTK"));
peptides.push_back(AASequence::fromString("GSPSFGPESISTETWSAEPYGR"));
peptides.push_back(AASequence::fromString("TELGFDPEAHFAIDDEVIAHTR"));
```

Then create an instance of the model, and predict the peak intensities of the peptides:

```
//Create new predictor model with vector of AASequences
PeakIntensityPredictor model;

//Perform prediction with LLM model
vector<double> predicted = model.predict(peptides);
```

You can output AASequence instances like normal strings:

```
//for each element in peptides print sequence as well as corresponding predicted peak intensity value.
for (Size i = 0; i < peptides.size(); i++)
{
  cout << "Intensity of " << peptides[i] << " is " << predicted[i] << endl;
}
```

4.11 References

[1] :Wiebke Timm: *Peak Intensity Prediction in Mass Spectra using Machine Learning Methods*, PhD Thesis (2008) **[2]** :Helge Ritter: *Learning with Self-Organizing Map, Artificial Neural Networks*, In T. Kohonen et al., eds.: Artificial Neural Networks, Elsevier Science Publishers (1991), 379-384 **[3]** :W. Timm, A. Scherbart, S. Böcker, O. Kohlbacher, T.W. Nattkemper: *Peak Intensity Prediction in MALDI-TOF Mass Spectrometry: A Machine Learning Study to support Quantitative Proteomics*, BMC Bioinformatics (2008)

4.12 Creating a new algorithm

Most of the algorithms in OpenMS share the following base classes:

- *ProgressLogger* is used to report the progress of the algorithm.
- *DefaultParamHandler* is used to make the handling of parameters (and their defaults) easy.
  In most cases, you will not even need accessors for single parameters.

The interfaces of an algorithm depend on the data structures it works on. For an algorithm that works on peak data, a non-template class should be used that provides template methods operating on *MSExperiment* or *MSSpectrum*, no matter which peak type is used. See *PeakPickerCWT* for an example.

For algorithms that do not work on peak data, templates should be avoided.