Report for PyImzMLParser

25.02.15

Summary

The parser is implemented as a single class. Its relevant methods are `getspectrum` and `getionimage`; the relevant attributes are `coordinates` (list of coordinates) and `imzmldict` (dict for meta data).

The XML file is parsed on calling the constructor using the `ElementTree` module's funtcion `parse`. The binary file is read when calling `getspectrum`.

The parser requires the modules `ElementTree` and `numpy`.

Memory mapping

`mmap` is used to map the entire binary file into virtual memory. For 64-bit processors, this should not cause any problems. In case memory mapping fails (e.g. on a 32-bit machine), a conventional file pointer will be used instead. `mmap` allows fixed-size mapping, too, but that would require effortful memory handling and would probably not benefit the performance in any way.

The actual amount of physical memory is irrelevant for mmap, because the file is not actually read into physical memory, but only mapped into virtual memory.

Definitions

The data operations performed by PyImzMLParser can be described by four terms that will be used throughout this report: - **Parse** the .imzML file: Use `ElementTree` to read the .imzML file and store it as a xml tree in memory - **Extract** meta data: Search the xml tree for desired meta data attributes and store these in a dedicated dict - **Read** the binary file: Read bytes from the binary file and store these as a string where each character of the string represents one byte - **Format** the binary data: Use `struct.unpack` to transform the byte-string into a list of numbers representing the m/z array and the itensity array

Parsing and *extracting* happen invisibly during initialization, whereas the binary *reading* and *formatting* happen on each call of `getspectrum`.

Usage

First, we instanciate the parser. This will parse the entire .imzML file and store the xml tree in memory. All following accesses to the meta data will be fast because the xml is already in memory. Also, this will use mmap to map the binary file to memory (if possible).

The use of mmap was not explicitly time profiled, but it improves reading performance by up to 10%. In worst case, performance of mmap is equal to reading without mmap

```
In [1]: from ImzMLParser import ImzMLParser

        p = ImzMLParser('Example.imzML')
```

Meta Data

Now we can get the meta data we're interested in, e.g. the pixel size. The meta data is stored in a regular python dict which we can access as an attribute. Keys are the identifiers from the controlled vocabularies of mzML (http://psidev.cvs.sourceforge.net/viewvc/psidev/psi/psi-ms/mzML/controlledVocabulary/psi-ms.obo) and imzML (http://imzml.org/download/imzml/imagingMS.obo). Currently, the dict only stores a number of selected attributes that are supposed to be used frequently. Those attributes are hardcoded into the method `readimzmlmeta`. This can be extended in the future.

```
In [2]: p.imzmldict["pixel size x"], p.imzmldict["pixel size y"]

Out[2]: (15.0, 15.0)
```

Or the entire dict at once:

```
In [10]: p.imzmldict

Out[10]: {'attenuation': 50.0,
          'focus diameter x': 10.0,
          'focus diameter y': 10.0,
          'matrix solution concentration': 10.0,
          'max count of pixels x': 180,
          'max count of pixels y': 90,
          'max dimension x': 2700,
          'max dimension y': 1350,
          'pixel size x': 15.0,
          'pixel size y': 15.0,
          'pulse duration': 10.0,
          'pulse energy': 10.0,
          'wavelength': 337.0}
```

If one needs to find a xml value that is not included in the `imzmldict`, it is possible to search the xml tree manually. The attribute `p.root` is the XML root element. This is an `Element` instance from the `TreeElement` module. See here () for details.

Spectra

Now we want to get the the entire list of spectra (for later conversion into a different format, for example). Spectra are accessed by their index in the .imzML file. Each call of `getspectrum` will result in two reading accesses in the binary file: One for mzArray, one for intensityArray. The coordinate of an index can be looked up in the attribute `coordinates`. As spectral data images are not neccessarily rectangular, the following generalized approach is recommended:

```
In [4]: for i,(x,y) in enumerate(p.coordinates):
            p.getspectrum(i)
```

For 3-dimensional images, simply add the z-coordinate. Note that this will raise a `ValueError` if there is no third dimension:

In [5]:
```
for i,(x,y,z) in enumerate(p.coordinates):
    p.getspectrum(i)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-a77d2cf6481c> in <module>()
----> 1 for i,(x,y,z) in enumerate(p.coordinates):
      2     p.getspectrum(i)

ValueError: need more than 2 values to unpack
```

In case the dimension is unknown, unpacking the coordinate tuple should be done inside the loop

In [6]:
```
for i,coords in enumerate(p.coordinates):
    if len(coords) == 3:
        p.getspectrum(i)
        x,y,z = coords
        # ... assign to 3-dimensional matrix
    else:
        p.getspectrum(i)
        x,y = coords
        # ... assign to 2-dimensional matrix
```

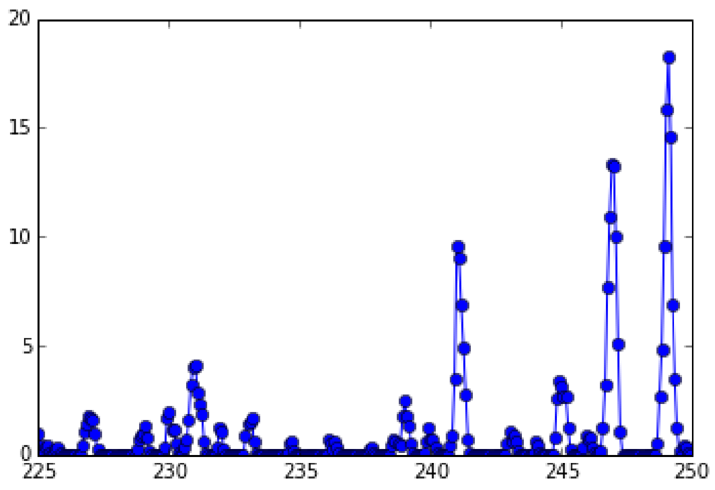
Each spectrum is a tuple of two lists, the m/z array and the intensity array, and can be plotted easily

In [7]:
```
%matplotlib inline
import matplotlib.pyplot as plt

mzA, intA = p.getspectrum(1)

plt.plot(mzA, intA, 'o-')
```

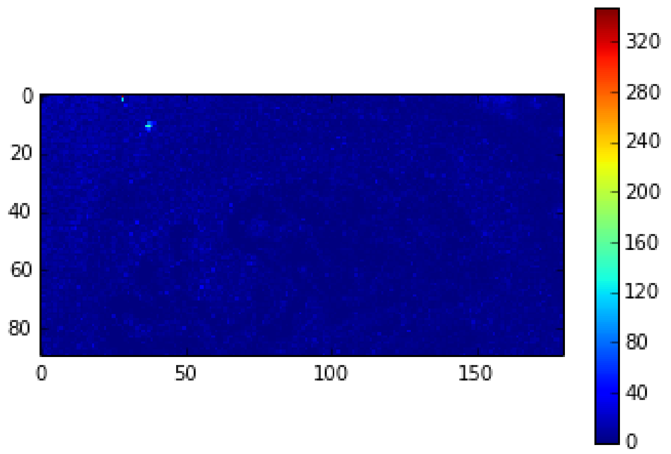
Out[7]:
```
[<matplotlib.lines.Line2D at 0x11baeb510>]
```



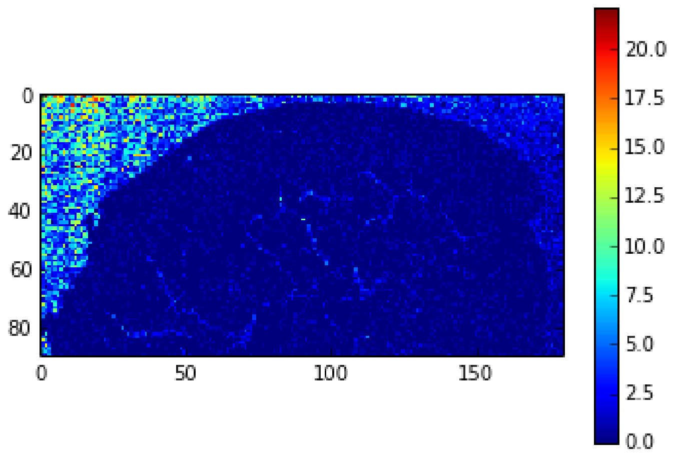
Ion image

Now, we want an image displaying the intensity of an ion throughout the spectrogram. Therefore, we use the `getionimage` method, which returns a numpy matrix that can be displayed as an image by `matplotlib`

```
In [8]: # entire ion image
        im = p.getionimage(500, tol=1000)
        plt.imshow(im).set_interpolation('nearest')
        plt.colorbar()
        plt.show()
```



```
In [9]: # pick a base peak
        mzA, intA = p.getspectrum(1)
        peakMz = mzA[intA.index(max(intA))]
        im = p.getionimage(peakMz)
        plt.imshow(im).set_interpolation('nearest')
        plt.colorbar()
        plt.show()
```



Benchmarks

Data name	Image size	Size XML/Binary	Total Local/NW	Parsing XML	Extracting meta data	Binary reading	Binary formatting	NW Parsing XML	NW Binary Read
S042_Continuous	180x90	25,8/19,4	12,5/-	10,483	1,787	0,092	0,127		
S042_Processed	180x90	25,9/38,9	12,6/-	10,504	1,821	0,166	0,126		
S043_Processed	155x175	43,1/4,6	20,6/-	17,522	3,046	0,034	0,028		
Mousebrain	100x50	8,0/156,3	4,5/7,4	2,930	0,566	0,567	0,451	3,258	2,990
Human	81x133	27,1/688,9	14,9/26,7	9,712	1,456	2,675	1,930	10,24	13,115
HumanBrain	179x140	23,8/212.820	-/6.525,925		1.547		1.642,769	10,208	4.169,873

Time in [s], size in [MB]

Only the last two columns are network-related.

Regular copying: - Network to SSD: ~46 MB/s

Pure reading speed: - SSD: ~270 MB/s - Network: ~48 MB/s

Local timing

Generally, we see that parsing the XML takes a lot longer than extracting the meta data from it, even on SSD. Both parsing and extracting grow linearly with the XML size. Therefore, the time consumption caused by meta data should be quite easy to foresee, given the XML file size.

For the binary file, reading and formatting take about equally long (on SSD). It is noticeable that for the same size, XML processing (reading + extracting) takes a lot longer than binary processing (reading + formatting). However, for data similar to the 'HumanBrain' sample, the binary file is so much larger than the XML file that it requires the very major part of the time.

Network vs SSD

Extracting meta data performs equally well for network and SSD access, as well as formatting binary data because those are operations performed solely on memory. Therefore, those times are only listed once in the table. The XML parsing only takes a little longer on Network than on SSD. This tells us that the major time is consumed by actually building the tree in memory and not by reading the file.

The binary read takes about **5-6 times longer** on network than on SSD.

Comparison to reading the entire file at once

By reading the example binary files into memory as a whole from both SSD and network, I estimated the reading speeds below the benchmark table. Those are about equal to the parser's performance despite reading the file as individual spectra.

Copying the file from Network to SSD before processing does not perform better, because it is slower than simply reading it into memory and would require an additional read from SSD. However, this could be a good idea if the file had to be read multiple times.

Potential for multi threading

The only two processes that could be parallelized are reading and formatting the binary file. This is benefitial when reading many spectra at once (which is the main purpose of this parser). For the 212GB human brain sample, this could improve performance by up to 28%.