Supplemental Text 1: Notable features new to ProteoWizard 3.0

Since the original publication on ProteoWizard[16], several new tools and thousands of library changes have been made. The mzML standard was updated to 1.1 in 2009 and ProteoWizard provided the reference implementation. The new chainsaw tool provides an easy way to digest or validate a FASTA database. The idconvert tool adds peptide identification support to ProteoWizard for the first time; currently, conversion is possible between pepXML and mzIdentML 1.1 and from Mascot DAT. An API for the upcoming TraML standard is available as well. Many spectrum processing filters have been added as well (e.g., chargeStatePredictor, defaultArrayLength, ETDFilter, MS2Denoise, MS2Deisotope, titleMaker, and threshold). In addition, CLI bindings have been added to allow the msData module to be accessed through a variety of .NET supported languages including C#, Visual Basic and IronPython.

Support:

- mzML 1.1 support
- mzIdentML 1.1 support
- pepXML support
- traML draft support
- ABI formats using ClearCore2 SDK
- Bruker using 3.1 CXT and also LC support (i.e. UV chromatograms)
- Thermo MSFileReader 2.2; added extraction of isolation width; EMR spectra, precursor ion spectra, triple-play, precursor intensity

Software Tools

idconvert, chainsaw, msPrefix, msCupid, idCat, peptideVenn, peptideHog, peekaboo, msConvertGUI, peekaboo, slideOut, Skyline, and Topograph

General optimizations:

- significantly optimized I/O code
- major changes to vendor readers (SRM support):

new spectrum_processing filters:

msLevel, precursorRefine, scanTime, sortByScanTime, metadataFixer, titleMaker, threshold, mzWindow, mzPrecursors, defaultArrayLength, MS2Denoise, MS2Deisotope, ETDFilter, chargeStatePredictor, activation, analyzer, polarity

Supplemental Text 2: Mapping mzML to the msData object

The data model of mzML documents (see **Box 1**) is mirrored in ProteoWizard's MSData code (see **Box 2**). The structure of the code supports the production and manipulation of mzML data.

```
 <xs:complexType name="FileDescriptionType">
  <xs:annotation>
   <xs:documentation>Information pertaining to the entire mzML file (i.e. not specific to any part of the data set) is stored here.</xs:documentation>
  </xs:annotation>
  <xs:sequence>
   <xs:element name="fileContent" type="dx:ParamGroupType">
    <xs:annotation>
     <xs:documentation>This summarizes the different types of spectra that can be expected in the file. This is expected to aid processing software in skipping files that do not contain appropriate spectrum types for it. It should also describe the nativeID format used in the file by referring to an appropriate CV term.</xs:documentation>
    </xs:annotation>
   </xs:element>
   <xs:element minOccurs="0" name="sourceFileList" type="dx:SourceFileListType" />
   <xs:element minOccurs="0" maxOccurs="unbounded" name="contact" type="dx:ParamGroupType" />
  </xs:sequence>
 </xs:complexType>
```

Box 1 mzML schema snippet for mzML

```
struct PWIZ_API_DECL SourceFile : public ParamContainer
{
   std::string id;
   std::string name;
   std::string location;

   SourceFile(const std::string _id = "",
              const std::string _name = "",
              const std::string _location = "");


   bool empty() const;
};

struct PWIZ_API_DECL FileDescription
{
   FileContent fileContent;
   std::vector<SourceFilePtr> sourceFilePtrs;
   std::vector<Contact> contacts;
   bool empty() const;
};
```

Box 2 MSData structure representing the mzMLType schema structure.

Most metadata are mapped into controlled vocabularies (CV) (**Box 3**) that ProteoWizard keeps in cvParam objects. These in turn are managed by ParamContainer objects that provide easy methods for working with collections of CV's, such as checking for a particular value in a set or testing for the child of a CV.

```
[Term]
id: MS:1000785
name: moving average smoothing
synonym: "box smoothing" EXACT[]
synonym: "boxcar smoothing" EXACT[]
synonym: "sliding average smoothing" EXACT[]
def: Reduces intensity spikes by averaging each point with two or more
adjacent points. The more adjacent points that used, the stronger the
smoothing effect.
is_a: MS:1000592 ! smoothing
```

The CV is maintained in a central repository by the HUPO-PSI. ProteoWizard parses the CV file at compile time and generates C++ code, which allows convenient, typesafe handling of the CV terms. As noted above, this also gives the programmer access to the inheritance, or “is-a”, structures that are critical in working with tools from arbitrary vendors or labs.

Supplemental Table 1: Supported open formats

Open Formats	Type	Data Type
PSI mzML[17]	XML	raw MS data
PSI mzIdentML[18]	XML	peptide/protein ids
PSI traML (draft) [19]	XML	targeted MS methods
pepXML[20]	XML	peptide ids
ISB mzXML[21]	XML	raw MS data
MZ5 [22]	HDF5	raw MS data
Mascot Generic (MGF)[23]	text	peak lists
Bruker Data Exchange (BTDX)	XML	peak lists
FASTA[24]	text	protein sequences
.ms2, .cms2 [25]	text	tandem spectra

ProteoWizard is able to read files in a variety of open formats that represent primary mass spectrometry data in a variety of ways, as well as related formats that hold protein sequence information or results of MS experiments. ProteoWizard is also able to output to many of these formats.

Supplemental Table 2: Supported vendor formats

Vendor Formats	Windows	Linux (Wine*)
AB SCIEX T2D	✔	X
AB SCIEX WIFF	✔	X
Agilent MassHunter	✔	✔
Bruker BAF/FID/YEP/U2	✔	✔
Thermo Fisher RAW	✔	X
Waters RAW	✔	X
Mascot DAT	✔	✔

ProteoWizard is able to read files in a variety of vendor formats. Though this capability is achieved through vendor-licensed libraries built for Windows in some instances, we have been able to successfully use those libraries in Linux using WINE emulation. Check mark denotes full support. X denotes lack of support.

* http://www.winehq.org/

Supplemental Example 1: Using the msData object to read MS data in multiple languages

Vendor libraries are subject to platform and non-commercial use licensing restrictions; consequently, some features of ProteoWizard may only be available if a vendor library is found and usable at runtime.

```
#include "pwiz_tools/common/FullReaderList.hpp"
#include "pwiz/data/MSDataFile.hpp"

using namespace pwiz::msdata;

int main(int argc, const char* argv[])
{

    // The FullReaderList object enables reading of all supported file types.
    ReaderPtr readers(new FullReaderList);


    // The MSDataFile is an MSData object that handles file I/O. Here
    // we pass the file name we wish to open, along with a list of all
    // standard format readers to load the data, into the constructor.
    MSDataFile msd(argv[1], &readers);

    // Here we get the number of Spectrum objects in a file.
     size_t numSpectra = msd.run.spectrumListPtr->size();
    cout << "Run ID: " << msd.run.id << "; " << numSpectra
          << " spectra" << endl;

    for (size_t i=0; i < numSpectra; ++i)
    {
        // Retrieve the spectrum from the SpectrumList object, with binary data
        const bool getBinaryData = true;

        SpectrumPtr s = msd.run.spectrumListPtr->spectrum(i, getBinaryData);
        // Fetching the data loaded by the call to SpectrumList::spectrum()

        vector<double>& mzArray = s->getMZArray()->data;
        vector<double>& intensityArray = s->getIntensityArray()->data;
        stringstream extrema;
        extrema << mzArray.front() << "," << intensityArray.front() << " "
                << mzArray.back() << "," << intensityArray.back();
        cout << "Spectrum ID: " << s->id << "; "
             << s->defaultArrayLength << " data points; "
              << extrema << endl;
    }
    return 0;
}
```

C#

```
using System;
using System.Linq;
using pwiz.CLI.msdata;

namespace PwizPaper
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = args[1];
            var listAll = new MSDataList();
            ReaderList.FullReaderList.read(path, listAll);
            foreach (var msd in listAll)
            {
                int numSpectra = msd.run.spectrumList.size();
                Console.WriteLine("Run ID: {0}; {1} spectra",
                    msd.run.id, numSpectra);

                for (int i = 0; i < numSpectra; i++)
                {
                    using (var s = msd.run.spectrumList.spectrum(i, true))
                    {
                        var mzArray = s.getMZArray().data;
                        var intensityArray = s.getIntensityArray().data;
                        Console.WriteLine("Spectrum ID: {0}; " +
                            "{1} data points; {2},{3},{4},{5}",
                            s.id,
                            s.defaultArrayLength,
                            mzArray.First(),
                            intensityArray.First(),
                            mzArray.Last(),
                            intensityArray.Last());
                    }
                }
            }
        }
    }
}
```

VB

```
Imports pwiz.CLI.msdata

Namespace PwizPaper
    Class Program
        Private Shared Sub Main(ByVal args As String())
            Dim path As String = args(1)
            Dim listAll = New MSDataList()
            ReaderList.FullReaderList.read(path, listAll)
            For Each msd As MSData In listAll
                Dim numSpectra As Integer = msd.run.spectrumList.size()
                Console.WriteLine("Run ID: {0}; {1} spectra", _
                                  msd.run.id, _
                                  numSpectra)

                For i As Integer = 0 To numSpectra 1
                    Using s = msd.run.spectrumList.spectrum(i, True)
                        Dim mzArray = s.getMZArray().data
                        Dim intensityArray = s.getIntensityArray().data
                        Console.WriteLine("Spectrum ID: {0}; " & _
                                          "{1} data points; {2},{3} {4},{5}", _
                                          s.id, _
                                          s.defaultArrayLength, _
                                          mzArray.First(), _
                                          intensityArray.First(), _
                                          mzArray.Last(), _
                                          intensityArray.Last())
                    End Using
                Next
            Next
        End Sub
    End Class
End Namespace
```

Supplemental Example 2: Using the mzID object to read identification data

```
#include "pwiz/data/identdata/DelimWriter.hpp"
#include "pwiz/data/identdata/DelimReader.hpp"
#include "pwiz/data/identdata/IdentDataFile.hpp"
#include "pwiz/data/identdata/Serializer_mzid.hpp"
#include "pwiz/utility/misc/Filesystem.hpp"
#include "pwiz/Version.hpp"

#include <vector>
#include <iostream>
#include <stdexcept>
#include <fstream>
#include <boost/algorithm/string.hpp>
#include <boost/program_options.hpp>
#include <boost/filesystem.hpp>

using namespace std;
using namespace pwiz::identdata;
using namespace pwiz::util;

// read an mzIdentML/pepXML file
IdentDataFile idd("data.mzid");

// iterate over DBSequences and write accession and sequence
BOOST_FOREACH(const DBSequencePtr& dbs, idd.sequenceCollection.dbSequences)
    cout << dbs->accession << ": " << dbs->seq << endl;

// iterate and write Peptides/Modifications
BOOST_FOREACH(const PeptidePtr& pep, idd.sequenceCollection.peptides)
{
    cout << pep->peptideSequence << ":";
    BOOST_FOREACH(const ModificationPtr& mod, pep.modifications)
        cout << " " << mod->monoisotopicMassDelta << "@" << mod->location;
    BOOST_FOREACH(const SubstitutionModificationPtr& mod, pep.substitutionModifications)
        cout << " " << mod->originalResidue << "->"
             << mod->replacementResidue << "@" << mod->location;
    cout << endl;
}

// iterate and write SpectralIdentificationItems from all spectra from all analyses
BOOST_FOREACH(const SpectrumIdentificationPtr& si,
   idd.analysisCollection.spectrumIdentifications)
BOOST_FOREACH(const SpectrumIdentificationResultPtr& sir,
   si->spectrumIdentificationListPtr->spectrumIdentificationResults)
BOOST_FOREACH(const SpectrumIdentificationItemPtr& sii,
   sir->spectrumIdentificationItems)
   cout << sir->spectrumID << " (" << sii->rank << "): "
        << sii->peptidePtr->peptideSequence << endl;

// iterate and write protein accessions and associated peptides from all
ProteinDetectionHypotheses
BOOST_FOREACH(const ProteinAmbiguityGroupPtr& pag,
idd.analysisCollection.proteinDetection.proteinDetectionListPtr->proteinAmbiguityGroups)
BOOST_FOREACH(const ProteinDetectionHypothesisPtr& pdh, pag->proteinDetectionHypotheses)
{
    cout << pdh->dbSequencePtr->accession << ":";
    BOOST_FOREACH(const PeptideHypothesis& ph, pdh->peptideHypotheses)
        cout << " " << ph.peptideEvidencePtr->peptidePtr->peptideSequence
             << " (start: " << ph.peptideEvidencePtr->start
             << ", spectra: " << ph.spectrumIdentificationItems.size();
    cout << endl;
}
```

Supplemental Example 3: Mass spectrometry I/O with the Bioconductor package mzR and the pwiz backend

The Bioconductor project[26] (www.bioconductor.org) was started a decade ago as a collection of open source packages for the analysis and comprehension of high-throughput biological data, building upon the statistical framework of the R language (www.r-project.org). Since then, it has evolved into a mature project with a core team and a community of many developers. Hundreds of publications, several books, and a series of workshops around the world relate to this project.

The mzR package in Bioconductor provides a common interface to several mass spectrometry data formats. The actual work of reading and parsing the data files is handled by the included C/C++ libraries or "backends." The ProteoWizard library is used as backend for the mzML file format via the Institute of Systems Biology's RAMP parser.

Below is a short example illustrating how to read data from a mass spectrometer. First the required packages are loaded to provide the required functionality and test data. An mzML file is then selected on the hard drive and is opened using the mzR openMSfile function. The latter creates a file handle that will give access to the content of the file:

```
> library(mzR)
> library(msdata)
> filename <system.file("microtofq/MM14.mzML", package = "msdata")
> mzml <openMSfile(filename)
```

The mzml file handle object is now used to directly access information stored in the file (i.e., to obtain the metadata):

```
> runInfo(mzml)
$scanCount
[1] 112

$dStartTime
[1] 270.334

$dEndTime
[1] 307.678

$msLevels
[1] 1
```

The actual data is obtained as a list of spectra or, alternatively, scans can be extracted individually:

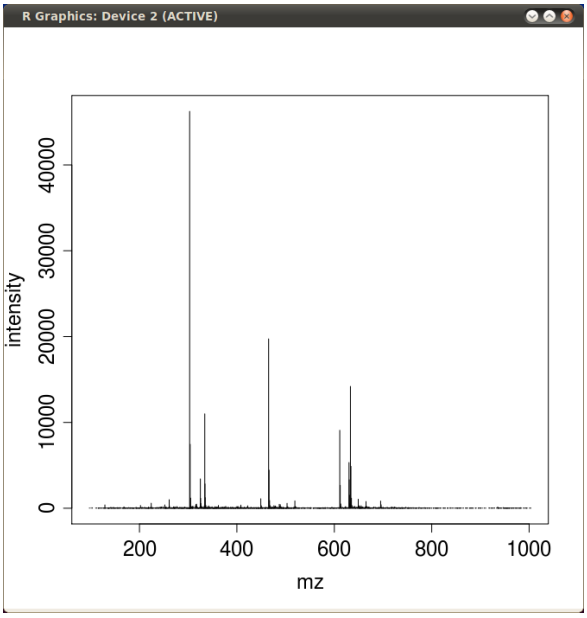
```
> length(peaks(mzml))
[1] 112

> dim(peaks(mzml, scan=14))
[1] 1553    2

> head(peaks(mzml, scan=14))
       mass  intensity
[1,]  97.37         47
[2,] 100.81         56
[3,] 103.44         54
[4,] 110.09         35
[5,] 112.05         41
[6,] 114.94         26
```

The individual spectra can be visualized using standard R plotting functionality:

```
> plot(peaks(mzml, scan=14),
       xlab="mz", ylab="intensity",
       type="h")
```



The mzR package is used by high-level data processing and analysis packages such as MSnbase [27] or XCMS [28, 29] to perform I/O on mass spectrometry data. mzR uses the Bioconductor infrastructure for version control and daily builds. Windows, MacOS and Linux packages for R >= 2.14 and installation instructions are available on http://www.bioconductor.org/packages/release/bioc/html/mzR.html.

Supplemental Example 4: Computing the mass of a peptide

```
// print data for a chemical formula (spaces optional)
#include "pwiz/utility/chemistry/Chemistry.hpp"
pwiz::chemistry::Formula f("C10 H20 O1 Se1");
cout << "Formula: " << f.formula() << "; " << f.monoisotopicMass()
     << "; " << f.molecularWeight()<< endl;


// create a peptide, compute and print its mass
pwiz::proteome::Peptide p("PEPTIDE");
cout << "Peptide: " << p.sequence() << " (" << p.monoisotopicMass()<< endl;

// this can also be done inline.
cout << "Peptide: "
    << pwiz::proteome::Peptide("PEPTIDE").monoisotopicMass()
    << endl;

// create a peptide, modify it, and print its data
#include "pwiz/data/proteome/Peptide.hpp"
pwiz::proteome::Peptide p("ELVIS(P1O4)LIVES(79.9)K", ModificationParsing_Auto);
cout << "Peptide: " << p.sequence() << " (" << p.monoisotopicMass() << ");";
p.modifications()[pwiz::proteome::ModificationMap::NTerminus()].push_back(Modification("O1"));
BOOST_FOREACH_FIELD((int position)(const ModificationList& mods), p.modifications())
BOOST_FOREACH(const Modification& m, mods)
    cout << " " << m.monoisotopicDeltaMass() << "@" << position;

// get fragmentation data for a peptide
bool monoisotopic = true, modified = true;
Fragmentation fn = p.fragmentation(monoisotopic, modified);
for (size_t i=1, length=p.sequence().length(); i <= length; ++i)
    cout << "b" << i << "(+1): " << fn.b(i,1) << "; "
         << "y" << (length-i) << "(+1):" << fn.y(length-i,1) << endl;
```

Supplemental Table 3: Available utilities and analysis classes

Path	Description
pwiz/utility/	**Utility data structures and algorithms (DSAs)**
chemistry/*	Optimized DSAs for chemistry and mass spectrometry, e.g. handling chemical formulas and computing peptide masses
math/*	Some convenient mathematical DSAs
minimxml/*	Stream-based SAXParser and XMLWriter
misc/COMInitializer.*	Easy management of per-thread COM initialization
misc/optimized_lexical_cast.hpp	Optimized lexical_cast from string to numeric types
misc/random_access_compressed_ifstream.*	Stream-based random-access to zlib archives
misc/IntegerSet.hpp	Efficient integer interval containers
pwiz/analysis/spectrum_processing/	**Spectrum processing algorithms**
SpectrumList_ChargeStateCalculator.*	Predicts whether MSn is singly/multiply charged from intensity relative to precursor m/z; or use ETDZ SVM prediction on ETD spectra
SpectrumList_Filter.*	Filters a SpectrumList based on a user-specified predicate
SpectrumList_PeakFilter.*	Filters spectrum peaks based on a user-specified predicate

SpectrumList_Smoother.*	Smooths a spectrum profile signal
SpectrumList_Sorter.*	Sorts a SpectrumList based on a user-specified predicate
SpectrumList_TitleMaker.*	Sets a title to spectra based on a user-specified format string
SpectrumListFactory.*	Parses strings to wrap a SpectrumList with the desired filters
SpectrumList_PeakPicker.*	If possible, calls vendor centroiding on a spectrum, else falls back to a portable but inferior method
SpectrumList_MetadataFixer.*	Overrides the base peak and TIC for all spectra of a SpectrumList
pwiz/analysis/proteome_processing/	**Protein and peptide processing algorithms**
ProteinList_DecoyGenerator.*	Generates on-the-fly decoy proteins for a ProteinList

Supplemental Example 5: Performing an *in silico* digest

For Users

The chainsaw utility allows a user to do *in silico* digests of proteins.

By default, the digest simulates tryptic digestion. Generating the resulting peptides is done by the following command:

> chainsaw database.fasta

The length of the resulting peptides can be controlled through *--minLength* and *–maxLength*.

In order to change the enzyme used in the simulation, the *--proteolyticEnzyme* option is used followed by the enzyme name (trypsin, chromotrypsin, clostripain, cyanogenBromide, pepsin). Alternately, the behavior of an arbitrary enzyme can be specified on the command line using the *–digestionMotif*, *--cleavageAgentRegex, --numMissedCleavages*, and *–specificity* (none, semi, fully) options.

The resulting data will be written to a tab-delimited file with sequence, protein mass, missedCleavages, specificity, nTerminusIsSpecific, cTerminusIsSpecific fields. A typical resulting file begins:

```
sequence        protein mass    missedCleavages specificity     nTerminusIsSpecific
cTerminusIsSpecific
MSQVQVQVQNPSAALSGSQILNK IPI:IPI00000001.2       2426.25875724   0       2       1       1
NQSLLSQPLMSIPSTTSSLPSENAGR      IPI:IPI00000001.2       2714.35450812   0       2       1       1
PIQNSALPSASITSTSAAAESITPTVELNALCMK      IPI:IPI00000001.2       3415.72147182   0       2       1       1
```

When run, an index file will be generated before the analysis begins. This vastly speeds up the analysis. These can be pre-generated by the *--indexOnly* option, which will write it in the same directory as the FASTA file.

> chainsaw --indexOnly database.fasta

This will write its results to a file named *database.fasta.index*.

Alternatively, the proteins themselves can be examined using the *--proteinSummary* argument. Each protein will be output with an index, its protein id, length, molecular weight, and description as it appears in the given FASTA file.

For Developers

The core of the chainsaw code is the go function. Within the go function, the majority of work is done by the ProteomeDataFile object, which reads in the user's FASTA file and provides an vector containing all the information about each protein listed in the FASTA (e.g., its identifiers, description, and sequence).

```
void go(const Config& config)
{
    // This section iterates through the user supplied files.
    vector<string>::const_iterator file_it = config.filenames.begin();
    for( ; file_it != config.filenames.end(); ++file_it)
    {
           // The  ProteomeDataFile object reads in and
        //  parses the FASTA file into easy to use Protein
           // objects. The "true" argument indicates that
        //an index should be generated.
        ProteomeDataFile pd(*file_it, true);

        // If only the indices are being generated,
         // then we bail out here for each file.
        if (config.indexOnly)
            continue;

        if (config.proteinSummary)
            writeSummary(config, pd);
        else
            writeDigestion(config, pd);
    }
}
```

The output is generated by iterating though the elements of ProteomeData::proteinListPtr of Protein objects. At the simplest level, iterating through the Protein objects gives access to contents of the file.

```
void writeSummary(const Config& config, const ProteomeData& pd)
{
    ofstream ofs;
    // This gets a reference to the list proteins wrapped in Protein objects.
    const ProteinList& pl = *pd.proteinListPtr;
    for(size_t index = 0, end=pl.size(); index < end; ++index)
    {
        try {
            // The ProteinPtr is of type boost::shared_ptr<Protein>
             // and can be treated like a normal C++
            // pointer. The Protein object has all the information
             // from the FASTA file as well has
            // helper functions such as molecularWeight()
            ProteinPtr proteinPtr = pl.protein(index, true);

            ofs << index
                << "\t" << proteinPtr->id
                << "\t" << proteinPtr->sequence().length()
                << "\t" << proteinPtr->molecularWeight()
                << "\t" << proteinPtr->description
                << "\n";
        }
        catch (runtime_error& e)
        {
            cerr << "Error summarizing protein "
                    << index << " (" << id << "): " << e.what() << endl;
        }
    }
}
```

To accomplish the *in silico* digest, the Digest object is used. The data needed for creating a Digest object is the Protein object to be digested, the regular expression of the agent, and the configuration object containing the maximum number of missed cleavages and peptide length bounds. Accessing the results of the digestion is done through the begin() and end() methods that behave like standard C++ STL iterators.

```
void writeDigestion(const Config& config, const ProteomeData& pd){
    ofstream ofs;

    // See  writeSummary above.
    const ProteinList& pl = *pd.proteinListPtr;
    for(size_t index = 0, end=pl.size(); index < end; ++index){
        try{
            // digest
            ProteinPtr proteinPtr = pl.protein(index, true);
            id = proteinPtr->id;

            // Constructing a Digestion object is easy at this point.
             // The Protein object is given by the index,
            // and the agent regular expression is chosen in a cascade
             // fashion. If  a cleavageAgentRegex is
            // provided, then it is used; otherwise, if a
             // digestionMotif is provided, it is used. If none of the
            // above are true, then the  proteolyticEnzyme
             // (default trypsin) is used.
            shared_ptr<Digestion> digestion;
            if (!config.cleavageAgentRegex.empty())
                digestion.reset(new Digestion(*proteinPtr,
                  boost::regex(config.cleavageAgentRegex),
                                config.digestionConfig));
            else if (!config.digestionMotif.empty())
                digestion.reset(new Digestion(*proteinPtr,
                                        config.digestionMotif,
                                        config.digestionConfig));
            else
                digestion.reset(new Digestion(*proteinPtr,
                    config.proteolyticEnzyme,
                                        config.digestionConfig));

            // The Digestion iterator returns DigestedPeptide
             // objects that can be queried for the resulting
            // peptides.
            for (Digestion::const_iterator jt = digestion->begin();
                                                  jt != digestion->end(); ++jt)
              ofs << jt->sequence()
                  << "\t" << proteinPtr->id
                  << "\t" << jt->monoisotopicMass(0, false)
                                /* unmodified neutral mass + h2o*/
                    << "\t" << jt->missedCleavages()
                    << "\t" << jt->specificTermini()
                    << "\t" << jt->NTerminusIsSpecific()
                    << "\t" << jt->CTerminusIsSpecific()
                    << "\n";
        }
        catch (runtime_error& e)
        {
            cerr << "Error digesting protein " << index << " (" << id << "): "
                << e.what() << endl;
        }
    }
}
```

Supplemental Table 4: ProteoWizard Toolkit applications

chainsaw	Generates in silico digests from FASTA files.
idCat	Displays the contents of identification results files (e.g., pepXML, protXML, mzIdentML). The output is in a tab delimited format by default.
idConvert	Converts between supported identification formats (e.g., mzIdentML, pepXML).
msAccess	This is a Swiss army knife of tools to display information about a mass spectrometry data file. Metadata, statistics, or a table of run summaries, windows of data, total ion current, and images can be produced and customized. Both content and form of the output can be manipulated by command line parameters. Some commonly used options set delimiters for table data output, m/z, and scan for data selection, and ms level filtering for run summaries.
msCat	Reads diverse MS data formats and outputs 4-column text.
msConvert **msConvert GUI**	Converts between supported MS data formats (e.g., mzML, mzXML, MGF). The input format is automatically detected by the ProteoWizard framework. The output format and data width are user selectable by command line parameters or a configuration file (e.g., mzML, mzXML, mz5, mgf); compression is also selectable (none, zlib, gzip). One of the more powerful features is the ability to filter. Filtering is added through the command line by way of the *--filter* argument. Available filters range from selecting a list of indexes or MS levels to recalculating the precursor of MS2 events (previously **msPrefix**). The entire list of filters can be obtained from the command line by running msConvert without arguments.
msCupid	A framework for AMT matching.
msdDir	Displays summary information about mass spectrometry data sources. The output can be displayed in brief, detailed, or full. Brief output creates a table with the columns (Type, Size, Last Modified, Name), whereas the detailed output adds the number of Spectra for each entry and full gives a tree output of the metadata.
msDiff	Outputs the differences in metadata between two mass-spectrometry files of any supported format. The difference is written in a hierarchical format that highlights the hereditary relationship.
msPicture	Generates a pseudo 2D gel picture from a mass-spectrometry data file. Command line options allow the user to select color scheme, scaling, coarseness, MS2 locations, and peptide identification locations.
msPostfix	Writes a tsv showing the scan by scan information on observed vs. calculated m/z values for MS1 features that have been mapped to nearby MS2 features.

msiStats	Calculates the number of features in a grid of data read from a tab-delimited LCMS feature file in msInspect format. In addition, totals are given for features found in rows as well as columns.
peekaboo	Detects and outputs a list of peaks produced by a ThermoFisher FT or OrbiTrap instrument. A window can be designated by the *--scanBegin*, *--scanEnd*, *--mzLow*, and *--mzHigh* command line arguments.
peptideHog	PeptideHog takes as input a set of pep.xml files from replicate runs of the same sample, and outputs a list of protein names, the number of runs that peptides from this protein were observed in, and the percent of the MS2s that these peptides took up in each run.
peptideProfile	Given a set of pep.xml files, outputs a file listing each unique peptide found in the aggregated MS2 data and the number of times the peptide was observed.
peptideVenn	A simple tool for finding a set of intersecting peptides between two runs. Outputs an xml file listing the matching peptides and all of the information contained in a pair of SpectrumQuery objects for each match.
seeMS	A windows GUI application for viewing spectra.
slideout	Converts from ThermoFisher .sld files to txt.

Supplemental Text 3a: Benchmarking ProteoWizard's Feature Extraction (Peak Picking)

To assess the performance of ProteoWizard's Peakaboo in identifying peaks, we used 9 MS runs from the Standard Protein Mix Database (Klimek et al, 2008), specifically data from Orbitrap MS machines. Each run consisted of raw data plus several thousand peptide identifications, of which on average 1032 were confident identifications (Peptide Prophet score > 0.99). For each confidently identified peptide, we located the nearest peak discovered by Peakaboo by finding the closest m/z value to that of the peptide within a retention time window of 9 minutes. Peakaboo successfully finds most (89%) of the confidently ID'ed peptides with an error of < 40 ppm, and the majority with an error of < 2 ppm (Figure S1). Of the remaining 11% peptides, most (9%) have a deviation of near one mass unit (not shown), likely corresponding to confusion of the monoisotopic peak. Finally, 2.3% of the peptides did not correspond to any nearby peak found by Peakaboo.

For comparison, we ran the same benchmarking on the feature finder in msInspect (Figure S1). The typical m/z error among well-matched peptides is several times larger for msInspect (Figure S2), and the fraction of peptides with one mass unit deviation (12%) is similar to that of Peakaboo. The fraction of peptides with no apparent match (1.1%) is moderately lower. Overall, it appears the performance of the two tools is comparable.

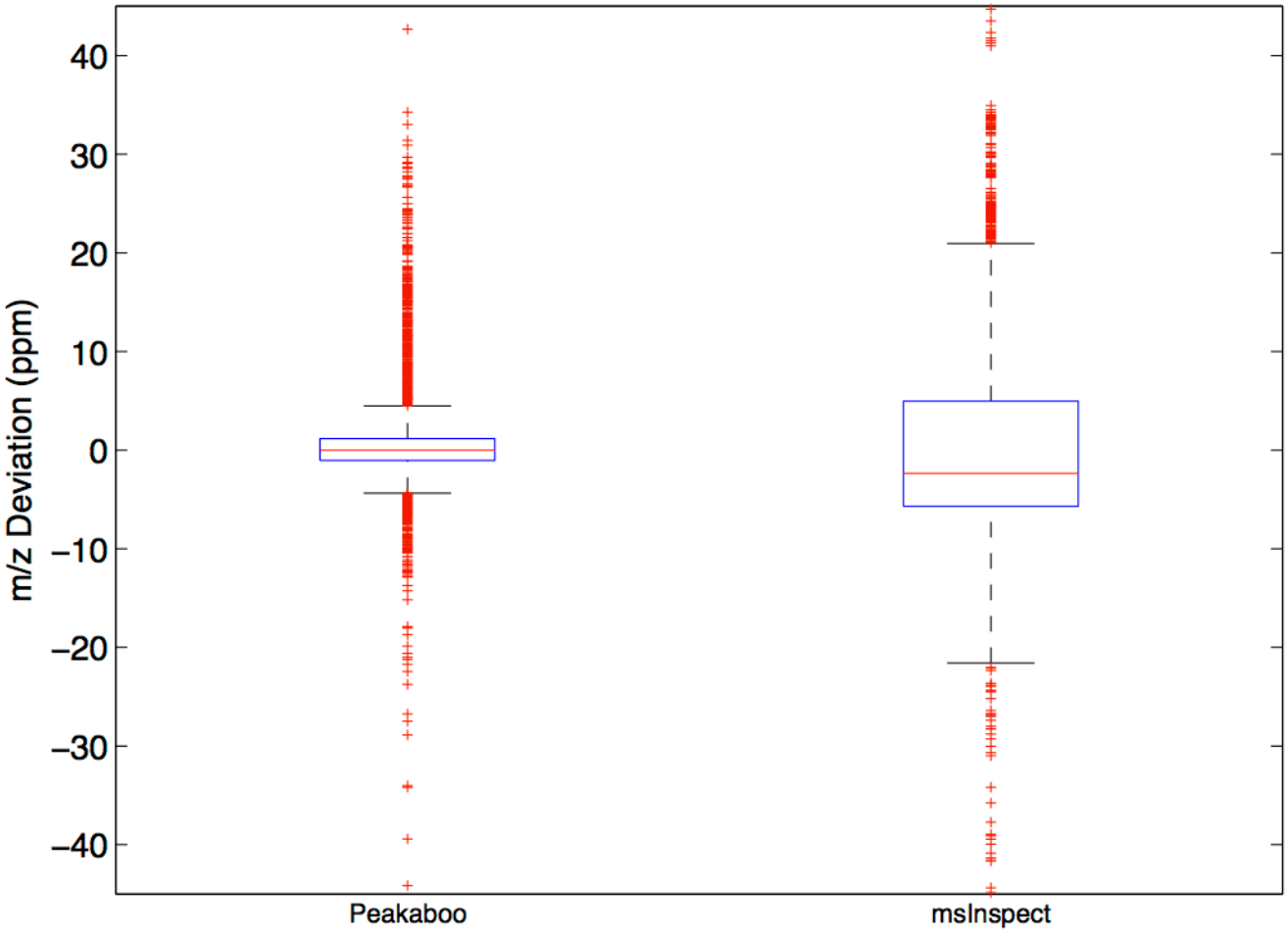


Figure S1. Boxplot of the fractional m/z distance between confidently ID'ed peptides (Peptide Prophet score > 0.99) and the nearest corresponding peak found by Peakaboo (left) and by msInspect (right), for n=9286 peptides from 9 runs from the Standard Protein Mix Database. Whiskers are 1.5 times the interquartile range. Peptides with deviation > 40 ppm not shown.

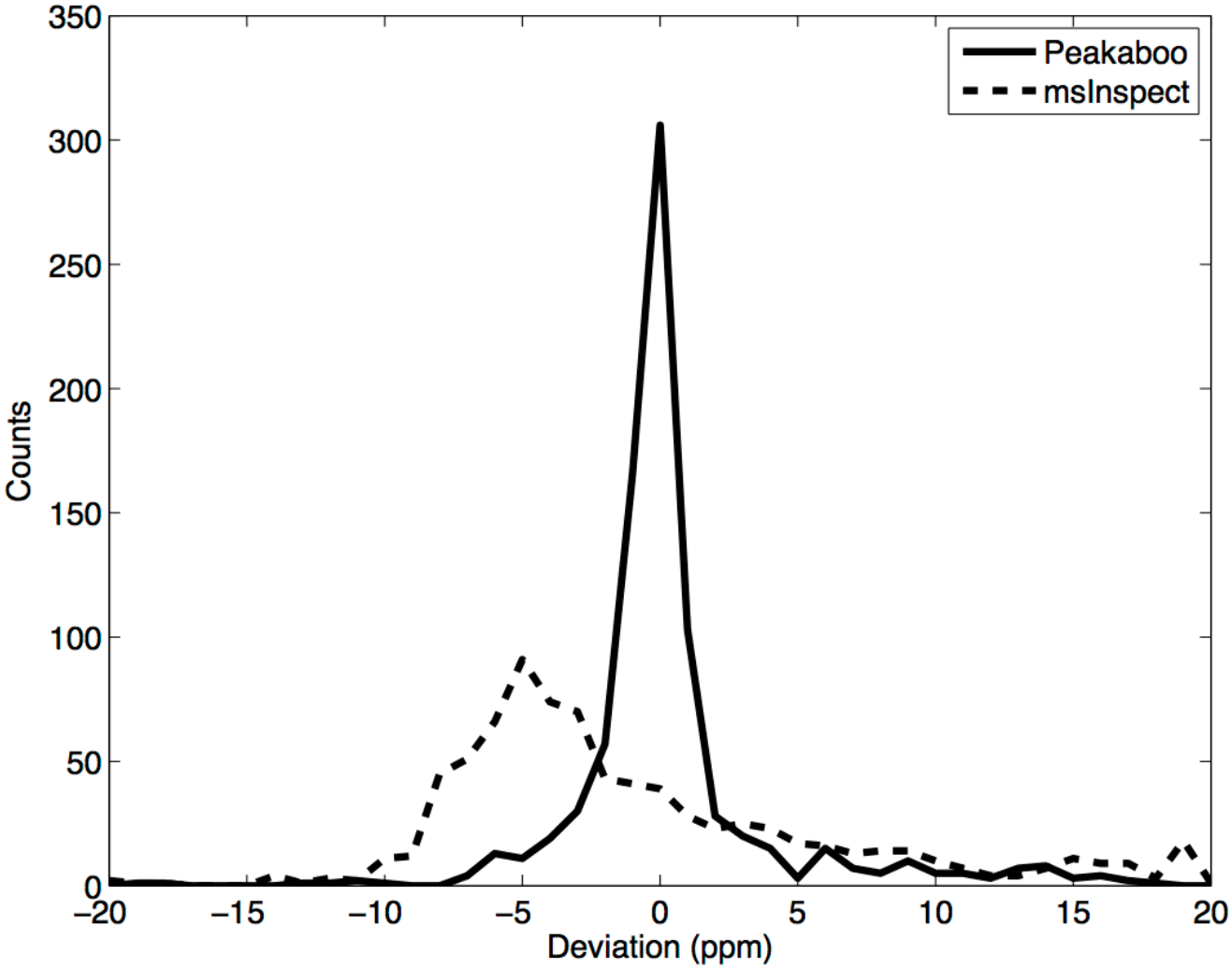


Figure S2. Histogram of m/z deviation for one of the 9 runs (other runs were similar), using n = 984 confidently identified peptides (Peptide Prophet score > 0.99), using Peakaboo (solid line) vs msInspect (dotted line). Peptides with deviation > 40 ppm not shown.

Supplemental Text 3b: Benchmarking ProteoWizard's Centroiding

To assess the performance of ProteoWizard's centroiding features we used an MS run (Mix 3 - B06-11071) from the Standard Protein Mix Database (Klimek et al, 2008), where the MS/MS data was collected in Profile Mode on an LTQ-FT. Data was either not centroided, or centroided with ProteoWizard's native centroider, or centroided with the vendor (Thermo) centroider. Data was then submitted to X! to match MS/MS to peptides followed by Peptide and Protein Prophet.

We then investigated the impact of the different centroiding algorithms for 602 MS/MS scans with high confidence peptide matches (PeptideProphet scores of at least 0.999 and matching peptide assignments across all three peak detection methods: vendor, ProteoWizard (pwiz), and profile mode).

Overall, it appears that the ProteoWizard native centroider performs comparably to the vendor centroider. However, it is more prone to include low-intensity peaks. Regardless of centroiding method, approximately 1300 peptides were identified with PeptideProphet scores > .9.

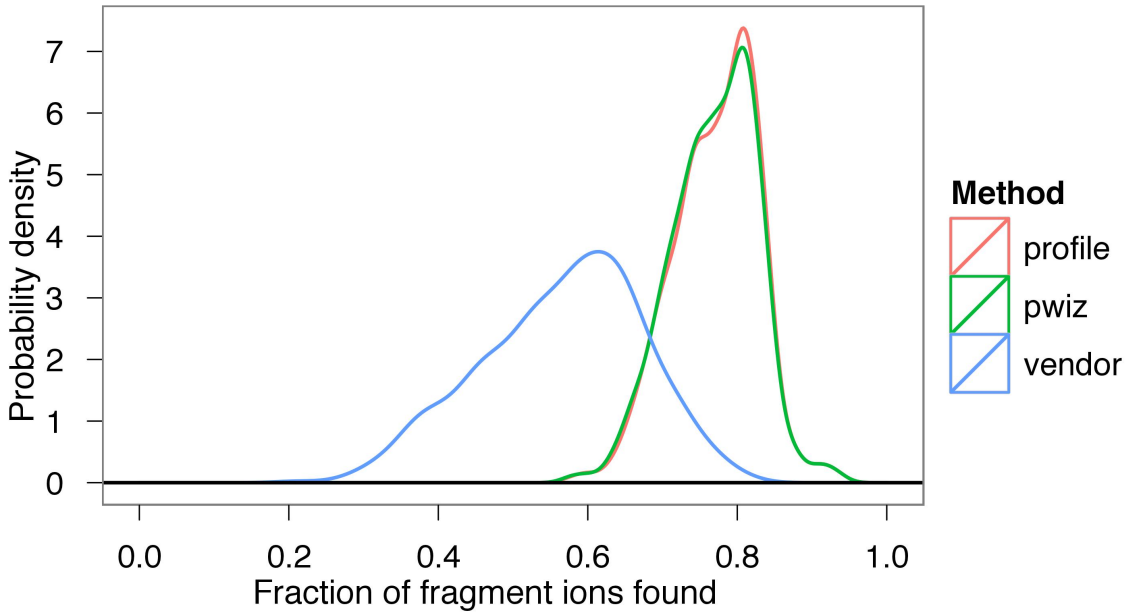


Figure S1. Smoothed histogram for the fraction of matching fragment ions (b/y series charge 1, 2, or 3) in centroided spectra across the 602 scans. The pwiz centroider has a higher fraction of matching fragment ions than the vendor centroider.

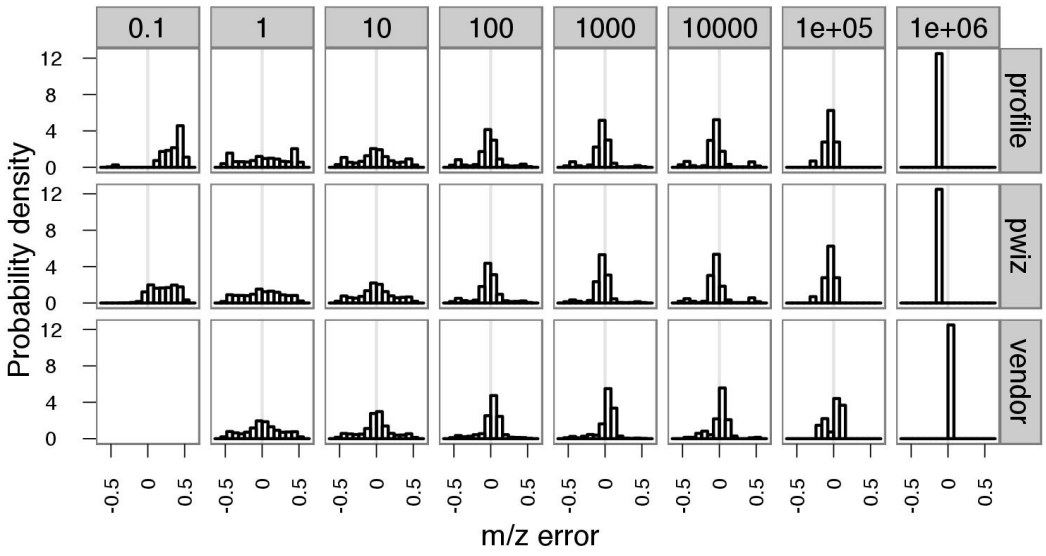


Figure S2. Fragment ion m/z error distributions as a function of method and peak intensity. Vendor centroider appears to threshold more aggressively than pwiz centroider. However, ProteoWizard's low intensity peaks have a higher average error.

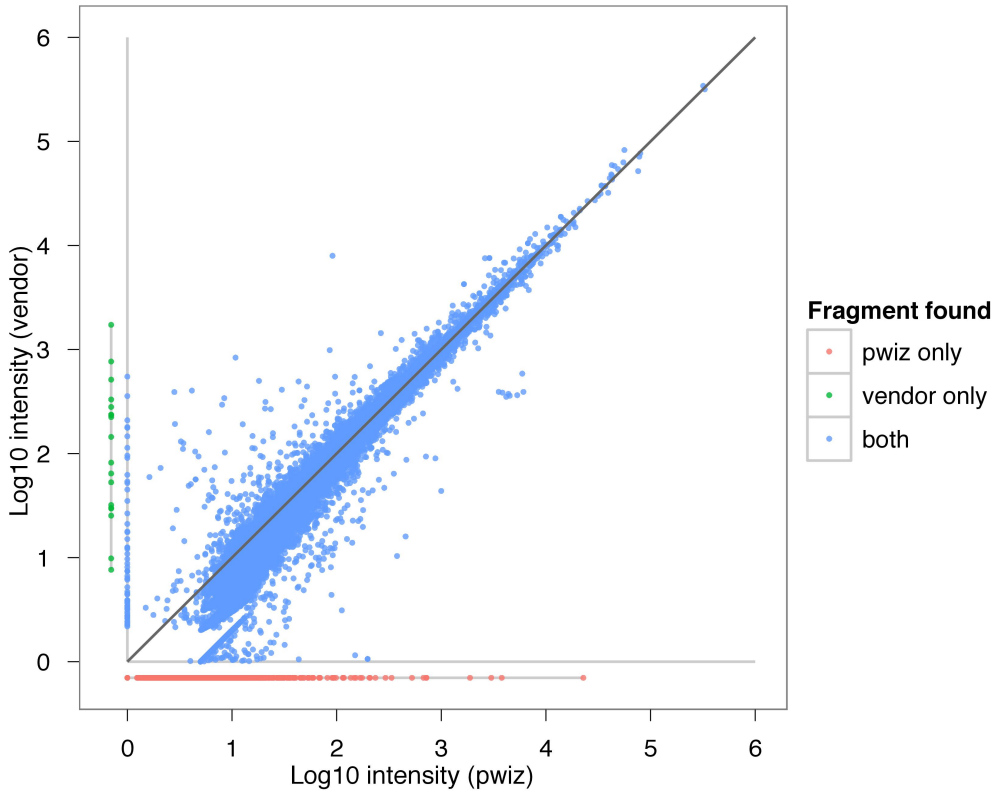


Figure S3. Scatterplot of $\log_{10}$ peak intensities comparing ProteoWizard (x-axis, scaled by 5x; low intensities clipped at 1) to vendor (y-axis) for 24582 matching ions (blue) shows generally broad correlation. 18 peaks were found by vendor only (green). 9635 peaks were found by ProteoWizard only. Notably these peaks span a range of intensities.

Citations:

16. Kessner, D., Chambers, M., Burke, R., Agus, D. & Mallick, P. ProteoWizard: open source software for rapid proteomics tools development. *Bioinformatics* **24**, 2534-2536 (2008).
17. Martens, L. et al. mzML--a community standard for mass spectrometry data. *Mol Cell Proteomics* **10**, R110 000133 (2011).
18. Eisenacher, M. mzIdentML: an open community-built standard format for the results of proteomics spectrum identification algorithms. *Methods Mol Biol* **696**, 161-177 (2011).
19. Helsens, K., Brusniak, M.Y., Deutsch, E., Moritz, R.L. & Martens, L. jTraML: An Open Source Java API for TraML, the PSI Standard for Sharing SRM Transitions. *J Proteome Res* **10**, 5260-5263 (2011).
20. Deutsch, E. 2010).
21. Pedrioli, P.G. et al. A common open representation of mass spectrometry data and its application to proteomics research. *Nature biotechnology* **22**, 1459-1466 (2004).
22. Wilhelm, M., Kirchner, M., Steen, J.A. & Steen, H. mz5: space- and time-efficient storage ofmass spectrometry data sets. *Mol Cell Proteomics* (2011).
23. Creasy, D. 2010).
24. Pearson, W.R. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in enzymology* **183**, 63-98 (1990).
25. McDonald, W.H. et al. MS1, MS2, and SQT-three unified, compact, and easily parsed file formats for the storage of shotgun proteomic spectra and identifications. *Rapid communications in mass spectrometry : RCM* **18**, 2162-2168 (2004).
26. Gentleman, R.C. et al. Bioconductor: open software development for computational biology and bioinformatics. *Genome biology* **5**, R80 (2004).
27. Gatto, L. & Lilley, K.S. MSnbase - an R/Bioconductor package for isobaric tagged mass spectrometry data visualisation, processing and quantitation. *Bioinformatics* (2011).
28. Smith, C.A., Want, E.J., O'Maille, G., Abagyan, R. & Siuzdak, G. XCMS: processing mass spectrometry data for metabolite profiling using nonlinear peak alignment, matching, and identification. *Anal Chem* **78**, 779-787 (2006).
29. Tautenhahn, R., Bottcher, C. & Neumann, S. Highly sensitive feature detection for high resolution LC/MS. *BMC bioinformatics* **9**, 504 (2008).