

## Supplemental data

**Isolation of Mitochondria.** Specimens of rat brain obtained from necropsy were immediately frozen in liquid nitrogen and subsequently stored at  $-80^{\circ}\text{C}$  until used for analysis. Brain specimens (0.5 – 1 g) were homogenized on ice in 10 mL MSB-Ca buffer consisting of 0.21 M mannitol, 0.07 M sucrose, 0.5 M Tris HCl and 3 mM  $\text{CaCl}_2$  (pH 7.5) using a motor driven Teflon coated Dounce homogenizer. Following homogenization, 1/10 volume of 0.1 M  $\text{Na}_2\text{EDTA}$  was added to the solution and it was centrifuged at  $1,500 \times g$  and  $4^{\circ}\text{C}$  for 20 min. Following centrifugation the supernatant was removed and centrifuged at  $20,000 \times g$  and  $4^{\circ}\text{C}$  for 20 min. The resulting pellet was washed three times in MSB-Ca followed by centrifugation. Because this crude pellet was often contaminated with golgi and cytosolic proteins, the pellet was resuspended in 2 mL 50/50 Percoll/MSB-Ca and centrifuged at  $50,000 \times g$  for 1 h. Following centrifugation, enriched mitochondria were isolated at a density of  $\sim 1.035 \text{ g/mL}$ , pelleted, resuspended in Percoll/MSB-Ca and centrifuged through a second Percoll gradient. The resulting enriched mitochondrial pellet was then rinsed three times in PBS and freeze dried. In general, yields of mitochondria enriched through two Percoll gradients approached  $\sim 200$  to  $300 \mu\text{g/g}$  tissue.

**ICAT Labeling.** For proteomic analysis, mitochondrial pellets were resuspended in  $300 \mu\text{l}$  80% 50 mM ammonium bicarbonate/20% acetonitrile and passed through a 26 gauge needle ten times to disrupt mitochondria. Protein concentration was determined using the Pierce BCA method and equal aliquots of each AD and control sample were pooled to generate representative AD and control samples. The pooled samples were divided into triplicate  $100 \mu\text{g}$  samples for ICAT labeling.

ICAT analysis of mitochondrial proteins was carried out using the commercially available cleavable ICAT Reagent Kit (Applied Biosystems, Foster City, CA) as previously described with modification. Briefly, 100  $\mu\text{g}$  samples of AD and control mitochondria were solubilized in 80  $\mu\text{l}$  Tris-SDS denaturing buffer by passing the sample through a 26 gauge needle 10 times and heating in a boiling water bath for 10 min with tris(2-carboxyethyl)phosphine (TCEP) to reduce protein disulfide bonds. After cooling to room temperature, control mitochondrial proteins were reacted with light ( $^{12}\text{C}_9$ -labeled) ICAT reagent, whereas AD proteins were reacted with heavy ( $^{13}\text{C}_9$ -labeled) ICAT reagent for 2 h at 37° C. Labeled protein samples were combined and digested with sequencing grade trypsin (Promega Madison, WI) (1:40 protease:protein) at 37°C for 16 h. The resulting peptide mixture was passed through a cation-exchange cartridge to remove TCEP, SDS, and unreacted ICAT reagents. ICAT labeled peptides in the eluent were isolated on an avidin cartridge and eluted with 30% acetonitrile/70% aqueous 0.4% trifluoroacetic acid (TFA). The isolated labeled peptides were evaporated to dryness, resuspended in TFA and incubated for two hours at 37° C to cleave the biotin portion of the ICAT tags. Solutions containing ICAT-labeled peptides were evaporated to dryness, reconstituted in 10  $\mu\text{l}$  5% acetonitrile/95% aqueous formic acid (0.1%) prior to 2D-LC/MS/MS.

**Chromatography.** Peptides were separated using a laboratory constructed 2D HPLC capillary column (350  $\mu\text{m}$  i.d.) containing 5 cm of strong cation exchange resin (Partisil 10 m, Alltech, Deerfield, IL) packed on top of 15 cm of reversed phase  $\text{C}_{18}$  resin (Macrosphere 300 5 m, Alltech, Deerfield, IL). Peptide separations used ammonium acetate steps from 0 to 300 mM to elute peptides from the cation exchange phase. Each

salt step was subjected to a complete reversed phase gradient from 5% acetonitrile (ACN)/95% aqueous 0.1% formic acid to 70% ACN/30% aqueous 0.1% formic acid over 110 min. Salt and organic gradients were generated using an LC Packings Ultimate HPLC pump (Dionex, Sunnyvale, CA) at a solvent flow of 4 $\mu$ L/min. LC/MS/MS spectra were acquired on a ThermoFisher Deca quadrupole ion trap mass spectrometer (ThermoFisher, San Jose, CA). Tandem mass spectra were acquired in a data dependent mode. Three spectra were averaged to generate the data dependent full scan spectrum with the most intense ion subjected to tandem mass spectrometry with five spectra averaged to produce the MS/MS spectrum. Masses subjected to MS/MS were excluded from further mass spectrometry for 2 min. The acquired tandem mass spectra were searched against a fasta database containing the rodent proteins using the CPAS<sup>17</sup> implementation of the Sequest Cluster (ThermoFisher) and the TPP. Peptide identifications matching rodent protein sequences with PeptideProphet probabilities of < 0.05 were omitted from the final output list.

## XPRESS Results

**Table 1.** Standard deviations and sample sizes for each Peptide Prophet score versus Correlation score classification. PeptideProphet scores were classified as "Low Probability" if they were below 0.90, "Medium Probability" for scores between 0.90 and 0.99, and "High Probability" for scores above 0.99. Correlation scores classified as "Low Correlation" if they were below 0.8, "Medium Correlation" if they were between 0.8 and 0.95, and "High Correlation" if they were greater than 0.95. The numbers in parentheses are the number of peptides in each cross-classification.

	LowCorr	MedCorr	HighCorr
LowProb	1.49 (128)	0.71 (77)	0.23 (40)
MedPro	0.52 (24)	0.29 (55)	0.16 (43)
HighProb	0.36 (58)	0.27 (157)	0.17 (167)

**Table 2.** PeptideProphet proteins with peptides filtered from the abundance calculation. Prob- ProteinProphet probability score; Tot Pep – total number of peptides associated with the protein; Uniq Pep – number of unique peptides; Ratio Pep – total number of peptides used in the abundance ratio calculation (after ProteinProphet filtering: removed adjusted PeptideProphet prob. < .05 and correlation scores < 0.5); L2H Mean – light to heavy mean calculated without filtering; F-L2H - light to heavy mean calculated with filtering; StdDev – standard deviation calculated without filtering; F-StdDev – standard deviation calculated with filtering.

	Prob	Tot Pep	Uniq Pep	Ratio Pep	L2H Mean	F-L2H	StdDev	F-StdDev
1	1.00	20	7	19	1.00	0.99	0.11	0.11
2	1.00	15	5	13	0.78	0.84	0.12	0.11
3	1.00	15	4	14	0.97	0.97	0.09	0.09
4	0.95	11	4	0	0.71	0.00	0.00	0.00
5	1.00	14	3	13	1.20	1.18	0.11	0.10
6	1.00	6	2	5	1.13	1.05	0.11	0.07
7	1.00	4	2	3	1.16	1.03	0.14	0.11
8	0.94	3	2	2	1.30	0.89	0.30	0.01
9	1.00	14	1	13	1.14	1.10	0.12	0.11
10	1.00	5	1	4	0.83	0.83	0.06	0.06
11	1.00	4	1	2	1.57	0.91	0.38	0.03
12	1.00	3	1	2	0.96	1.10	0.80	0.01
13	1.00	2	1	1	1.10	0.65	0.25	0.00
14	0.24	2	1	1	10.69	23.05	3.57	0.00
15	0.98	1	1	0	0.65	0.00	0.00	0.00

16	0.40	1	1	0	0.93	0.00	0.00	0.00
17	0.37	1	1	0	0.95	0.00	0.00	0.00
18	0.35	1	1	0	0.44	0.00	0.00	0.00
19	0.30	1	1	0	9.44	0.00	0.00	0.00
20	0.29	1	1	0	0.63	0.00	0.00	0.00

Figure 1 Xpress results for a peptide from the pepXML file. The confidence score is new attribute added to contain the correlation coefficient of the heavy and light peaks.

```
<analysis_result analysis="xpress">
  <xpressratio_result
    light_firstscan="354"
    light_lastscan="383"
    light_mass="1705.800"
    heavy_firstscan="354"
    heavy_lastscan="383"
    heavy_mass="1714.800"
    confidence_score="0.906"
    mass_tol="1.000" ratio="0.96:1"
    heavy2light_ratio="1.04:1"
    light_area="2.45e+007"
    heavy_area="2.56e+007"
    decimal_ratio="0.96"/>
</analysis_result>
```

## Source code

XPressPeptideParser /linreg.cpp

```
#include <math.h>
#include <float.h>
#include "linreg.h"

LinearRegression::LinearRegression(Point2D *p, int size)
{
    a = b = sumX = sumY = sumXY = 0.0;
    sumXsquared = sumYsquared = 0.0;
    n = 0;

    if (size > 0 && p != NULL)
        for (int i = 0; i < size; i++)
            addPoint(p[i]);
}
```

```

}

LinearRegression::LinearRegression
(double *x, double *y, int size)
{
    a = b = sumX = sumY = sumXY = 0.0;
    sumXsquared = sumYsquared = 0.0;
    n = 0;

    if (size > 0 && x != NULL && y != NULL)
        for (int i = 0; i < size; i++)
            addXY(x[i], y[i]);
}

LinearRegression::LinearRegression
(double *x, double *y, int iStart, int iEnd)
{
    a = b = sumX = sumY = sumXY = 0.0;
    sumXsquared = sumYsquared = 0.0;
    n = 0;

    if (iEnd - iStart > 0 && x != NULL && y != NULL)
        for (int i = iStart; i <= iEnd ; i++)
            addXY(x[i], y[i]);
}

void LinearRegression::addXY(const double& x, const double& y)
{
    n++;
    sumX += x;
    sumY += y;
    sumXsquared += x * x;
    sumYsquared += y * y;
    sumXY += x * y;
    Calculate();
}

void LinearRegression::Calculate()
{
    coefD = 0;
    coefC = 0;
    stdError = 0;
    if (haveData())
    {
        if (fabs( double(n) * sumXsquared - sumX * sumX)
            > DBL_EPSILON)
        {
            b = ( double(n) * sumXY - sumY * sumX) /
                ( double(n) * sumXsquared - sumX * sumX);
            a = (sumY - b * sumX) / double(n);

            double sx = b * (sumXY - sumX * sumY / double(n));
            double sy2 = sumYsquared - sumY * sumY / double(n);
            double sy = sy2 - sx;

            coefD = sx / sy2;
            coefC = sqrt(coefD);
        }
    }
}

```

```

        stdError = sqrt(sy / double(n - 2));
    }
    else
    {
        a = b = coefD = coefC = stdError = 0.0;
    }
}

ostream& operator<<(ostream& out, LinearRegression& lr)
{
    if (lr.haveData())
        out << "f(x) = " << lr.getA()
            << " + ( " << lr.getB()
            << " * x )";
    return out;
}

```

### XPressPeptideParser /linreg.h

```

/* linreg.h */
#ifndef _LINREG_H_
#define _LINREG_H_
#include <iostream>

using std::ostream;

// a class encapsulating a point in Cartesian coordinates
class Point2D
{
public:
    Point2D(double X = 0.0, double Y = 0.0) : x(X), y(Y)
    { }

    void setPoint(double X, double Y) { x = X; y = Y; }
    void setX(double X) { x = X; }
    void setY(double Y) { y = Y; }

    double getX() const { return x; }
    double getY() const { return y; }

private:
    double x, y;
};

// a linear regression analysis class
class LinearRegression
{
    friend ostream& operator<<(ostream&, LinearRegression&);

public:
    // Constructor using an array of Point2D objects
    // This is also the default constructor
    LinearRegression(Point2D *p = 0, int size = 0);
}

```

```

        LinearRegression(double *x, double *y, int size = 0);

        LinearRegression(double *x, double *y, int iStart, int
iEnd);

virtual void addXY(const double& x, const double& y);
virtual void addPoint(const Point2D& p)
{ addXY(p.getX(), p.getY()); }

// Must have at least 3 points to calculate
// standard error of estimate.
//Do we have enough data?
int haveData() const { return (n > 2 ? 1 : 0); }
int items() const { return n; }

virtual double getA() const { return a; }
virtual double getB() const { return b; }

double getCoefDeterm() const { return coefD; }
double getCoefCorrel() const { return coefC; }
double getStdErrorEst() const { return stdError; }
virtual double estimateY(double x) const
{ return (a + b * x); }

protected:
int n; // number of data points input
double sumX, sumY; // sums of x and y
double sumXsquared, // sum of x squares
sumYsquared; // sum y squares
double sumXY; // sum of x*y

double a, b; // coefficients of f(x) = a + b*x
double coefD, // coefficient of determination
coefC, // coefficient of correlation
stdError; // standard error of estimate

void Calculate(); // calculate coefficients
};

#endif // end of linreg.h

```

### XPressPeptideParser / XPressPeptideParser.cxx

```

/*
Program      : XPressPeptideParser
Author       : J.Eng and Andrew Keller <akeller@systemsbiology.org>
Date        : 11.27.02

```

Additional work for mzData handling Copyright (C) Brian Pratt  
Insilicos LLC 2005

Primary data object holding all mixture distributions for each  
precursor ion charge

Copyright (C) 2003 Andrew Keller

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Andrew Keller  
Insitute for Systems Biology  
1441 North 34th St.  
Seattle, WA 98103 USA  
akeller@systemsbiology.org

```
*/
#include "XPressPeptideParser.h"
#include "TPPVersion.h" // contains version number, name,
revision
#include "TagListComparator.h" // regression test stuff - bpratt
Insilicos LLC, Nov 2005
#include <string>

XPressPeptideParser::XPressPeptideParser(const char *xmlfile, const
InputStruct & options, const char *testMode):Parser("xpress")
{
    pInput_ = options;

    fp_ = NULL;

    pCache_ = getScanCache(250);

#ifdef USE_STD_MODS
    modinfo_ = NULL;
#endif

    testMode_ = testMode ? strdup(testMode) : NULL; // regression
test stuff - bpratt Insilicos LLC, Nov 2005

    init(xmlfile);
}

XPressPeptideParser::~XPressPeptideParser()
{
    if (xmlfile_ != NULL)
    {
        delete[]xmlfile_;
    }
}
```

```

    }
    free(testMode_);
    freeScanCache(pCache_);
}

void XPressPeptideParser::parse(const char *xmlfile)
{
    readAllModMasses(xmlfile);
    char *engine = NULL;
    char *enzyme = NULL;
    char *massspec = NULL;
    Array < Tag * >*tags = NULL;
    Tag *tag = NULL;

    char *data = NULL;

    double MIN_PROB = modelOpts_.minprob_; //0.0; //0.05; // for
now

    char text3[100];

    //
    // regression test stuff - bpratt Insilicos LLC, Nov 2005
    //
    Array < Tag * >test_tags;
    Boolean bLearnTest, bRunTest;
    char *testFileName = NULL;
    checkRegressionTestArgs(testMode_, bLearnTest, bRunTest);
    if (bLearnTest || bRunTest)
    {
        std::string options;
        testFileName = constructTagListFilename(xmlfile, // input file
        testMode_, // program args
        "XPressPeptideParser", //
program name
        bLearnTest ? LEARNTEST : RUNTEST);
        // user info output
    }

    #define RECORD(tag) {(tag)->write(fout);if (bLearnTest||bRunTest)
{test_tags.insertAtEnd(new Tag(*(tag)));} }

    Tag *timestamp_start = new Tag("analysis_timestamp", True,
False);
    timestamp_start->setAttributeValue("analysis", getName());
    timestamp_start->setAttributeValue("time", time_);
    timestamp_start->setAttributeValue("id", "1");
    Tag *timestamp_stop = new Tag("analysis_timestamp", False, True);

    Tag *timestamp = new Tag("xpressratio_timestamp", True, True);
    sprintf(text3, "%d", pInput_.bXpressLight1);
    timestamp->setAttributeValue("xpress_light", text3);

    Tag *analysis_start = new Tag("analysis_summary", True, False);
    analysis_start->setAttributeValue("analysis", getName());
    analysis_start->setAttributeValue("time", time_);

```

```

Tag  *analysis_stop = new Tag("analysis_summary", False, True);

Tag  *summary = getSummaryTag(pInput_);

Tag  *result_start = new Tag("analysis_result", True, False);
result_start->setAttributeValue("analysis", getName());
Tag  *result_stop = new Tag("analysis_result", False, True);

TagFilter *xpress_filter = new TagFilter("analysis_result");
xpress_filter->enterRequiredAttributeVal("analysis", getName());
TagFilter *xpress_summ_filter = new
TagFilter("analysis_timestamp");
xpress_summ_filter->enterRequiredAttributeVal("analysis",
getName());
TagFilter *xpress_summ = new TagFilter("analysis_summary");
xpress_summ->enterRequiredAttributeVal("analysis", getName());

char suffix[] = ".tmp";
char *outfile = new char[strlen(xmlfile) + strlen(suffix) + 1];
strcpy(outfile, xmlfile);
strcat(outfile, suffix);

ofstream fout(outfile);
if (!fout)
{
    cerr << "cannot write output to file " << outfile << endl;
    exit(1);
}

Boolean first = False;
Tag  *xpress_ratio = NULL;
int  ratio_tags_written = 0;

Boolean collected = False;

#ifdef USE_STD_MODS
    monoisotopic_ = False;          // unless proven otherwise
    Boolean mod_on = False;
    Array < Tag * >*modifications = NULL;
#endif
Boolean top_hit = False;

ifstream fin(xmlfile);
if (!fin)
{
    cerr << "error opening " << xmlfile << endl;
    exit(1);
}
char *nextline = new char[line_width_];
while (fin.getline(nextline, line_width_))
{
    //cout << "next: " << nextline << endl;

    data = strstr(nextline, "<");
    while (data != NULL)
    {
        tag = new Tag(data);
    }
}

```

```

        //tag->write(cout);
        collected = False;

        setFilter(tag);

        if ((!xpress_filter->filter(tag) && !xpress_summ_filter-
>filter(tag) && !xpress_summ->filter(tag))
        {

            if (tag->isStart() && !strcmp(tag->getName(),
"msms_pipeline_analysis"))
            {
                RECORD(tag);
                RECORD(analysis_start);
                RECORD(summary);
                RECORD(analysis_stop);
                delete analysis_start;
                analysis_start = NULL;
                delete summary;
                summary = NULL;
                delete analysis_stop;
                analysis_stop = NULL;

            }
            else if (tag->isStart() && !strcmp(tag->getName(),
"msms_run_summary"))
            {
                rampConstructInputPath(mzXMLfile_, sizeof(mzXMLfile_),
                pInput_.szMzXMLDir, tag-
>getAttributeValue("base_name"));
                if ((fp_ = rampOpenFile(mzXMLfile_)) == NULL)
                {
                    printf("Error - cannot open %s\n", mzXMLfile_);
                    exit(1);
                }
                // set index here.....
                index_ = readIndex(fp_, getIndexOffset(fp_),
&(pInput_.iAnalysisLastScan));
                if (pCache_ != NULL) {
                    clearScanCache(pCache_); // this was missing, ouch - bpratt
Nov 10 2006
                }

                pInput_.iAnalysisFirstScan = 1;

                first = True;
                RECORD(tag);
            }
            // msms_summ
            else if (tag->isEnd() && !strcmp(tag->getName(),
"search_summary"))
            {
                RECORD(tag);
                RECORD(timestamp_start);
                RECORD(timestamp);
                RECORD(timestamp_stop);

            }
        }

```

```

        else if (tag->isStart() && !strcmp(tag->getName(),
"search_summary"))
        {
            monoisotopic_ = !strcmp(tag-
>getAttributeValue("precursor_mass_type"), "monoistotopic");
            RECORD(tag);
        }
        // have a modification worth recording here
        else if (tag->isStart()
            && !strcmp(tag->getName(), "terminal_modification")
            && strchr(pInput_.szXpressResidues, tag-
>getAttributeValue("terminus")[0]) != NULL)
        {

            RECORD(tag);

        }
        // term modification
        else if (tag->isStart()
            && !strcmp(tag->getName(), "aminoacid_modification")
            && strchr(pInput_.szXpressResidues, tag-
>getAttributeValue("aminoacid")[0]) != NULL)
        {

            RECORD(tag);

        }
        // aa modification

        else if (filter_)
        {

            if (tag->isStart() && !strcmp("spectrum_query", tag-
>getName()))
            {
                pInput_.iFirstScan = atoi(tag-
>getAttributeValue("start_scan"));
                pInput_.iLastScan = atoi(tag-
>getAttributeValue("end_scan"));
                pInput_.iChargeState = atoi(tag-
>getAttributeValue("assumed_charge"));
            }
            else if (tag->isStart() && !strcmp("search_hit", tag-
>getName()) && !strcmp("1", tag->getAttributeValue("hit_rank")))
            {
                strcpy(pInput_.szPeptide, tag-
>getAttributeValue("peptide"));
                pInput_.dPeptideMass = (double) (atof (tag->
getAttributeValue("calc_neutral_pep_mass")) + 1.00794);
                top_hit = True;
            }
            else if (tag->isStart() && !strcmp("search_hit", tag-
>getName()) && strcmp("1", tag->getAttributeValue("hit_rank")))
            {
                top_hit = False;
            }
        }
#ifdef USE_STD_MODS

```

```

        else if (top_hit && tag->isStart() &&
!strcmp("modification_info", tag->getName()))
        {
            if (modifications == NULL)
                modifications = new Array < Tag * >;
            modifications->insertAtEnd(tag);
            mod_on = !tag->isEnd();
            if (!mod_on) { // tag already closed, process it now
                modinfo_ = new ModificationInfo(modifications);
            }
        }
        else if (top_hit && mod_on && tag->isEnd() &&
!strcmp("modification_info", tag->getName()))
        {
            modifications->insertAtEnd(tag);
            modinfo_ = new ModificationInfo(modifications);
            mod_on = False;
        }
        else if (top_hit && mod_on)
        {
            modifications->insertAtEnd(tag);
        }
#endif

        if (tags == NULL)
            tags = new Array < Tag * >;
        tags->insertAtEnd(tag);
        collected = True;
    }
    else
    {
        if (tag->isEnd() && !strcmp(tag->getName(),
"msms_run_summary"))
        {
            if (fp_ != NULL)
                rampCloseFile(fp_);
            if (index_ != NULL)
                delete index_;
        }

        if (tag != NULL)
        {
            RECORD(tag);
        }
    }

    if (filter_memory_)
    {
        // process
        if (tags != NULL)
        {
            if (pInput_.iChargeState >= 0)
                xpress_ratio = getRatio();

            for (int k = 0; k < tags->length(); k++)
            {
                if ((*tags)[k] != NULL)
                {

```

```

        if (!xpress_filter->filter((*tags)[k]))
        {
            // here check for correct time to write
xpress tag
            if ((*tags)[k]->isEnd() &&
!strcmp((*tags)[k]->getName(), "search_hit") && xpress_ratio != NULL)
            {
                ratio_tags_written++;
                const int FEEDBACK = 20;
                if (ratio_tags_written % FEEDBACK == 0)
                    cout << ".";
                if (ratio_tags_written % (FEEDBACK * 50)
== 0)
                    cout << " " << ratio_tags_written /
1000 << "k\n";

                RECORD(result_start);
                RECORD(xpress_ratio);
                RECORD(result_stop);
                delete xpress_ratio;
                xpress_ratio = NULL;
            }
            RECORD((*tags)[k]);
        }
        delete (*tags)[k];
    }
    // next tag
    delete tags;
    tags = NULL;
#ifdef USE_STD_MODS
    if (modifications != NULL)
    {
        delete modifications;
        modifications = NULL;
    }
#endif
}

pInput_.iChargeState = -1;    // reset

#ifdef USE_STD_MODS
    if (modinfo_ != NULL)
        delete modinfo_;
    modinfo_ = NULL;
#endif

    //xpress_ratio = NULL;
}
// if not filtered
else
{
}
// filtered
if (!collected && tag != NULL)
    delete tag;
data = strstr(data + 1, "<");
}
// next tag

```

```

    } // next line
    fin.close();
    fout.close();

    if (!overwrite(xmlfile, outfile, "</msms_pipeline_analysis>"))
    {
        cerr << "error: no xpress data written to file " << xmlfile <<
endl;
    }

    if (bLearnTest || bRunTest)
    {
        //
        // regression test stuff - bpratt Insilicos LLC, Nov 2005
        //

        if (bLearnTest)
        {
            writeTagListFile(testFileName, test_tags);
        }
        else
        {
            if (TagListComparator(test_tags, testFileName).compare())
            {
                cerr << "regression test FAILED in XPressPeptideParser!!!"
<< endl;
                exit(1);
            }
        }

        delete [] testFileName;
        for (int k = test_tags.length(); k--;)
        {
            delete test_tags[k];
        }
    }

    delete [] nextline;

    delete timestamp_start;
    delete timestamp_stop;
    delete timestamp;
    if (analysis_start != NULL)
        delete analysis_start;
    if (analysis_stop != NULL)
        delete analysis_stop;
    if (summary != NULL)
        delete summary;
    delete result_start;
    delete result_stop;
    delete xpress_filter;
    delete xpress_summ_filter;
    delete xpress_summ;
}

void XPressPeptideParser::readAllModMasses(const char *xmlfile)
{

```

```

char *engine = NULL;
char *enzyme = NULL;
char *massspec = NULL;
Array < Tag * >*tags = NULL;
Tag *tag = NULL;

char *data = NULL;

double MIN_PROB = modelOpts_.minprob_; //0.0; //0.05; // for
now

char text3[100];

//
// regression test stuff - bpratt Insilicos LLC, Nov 2005
//
Array < Tag * >test_tags;
Boolean bLearnTest, bRunTest;
char *testFileName = NULL;
checkRegressionTestArgs(testMode_, bLearnTest, bRunTest);
if (bLearnTest || bRunTest)
{
    std::string options;
    testFileName = constructTagListFilename(xmlfile, // input file
                                           testMode_, // program args
                                           "XpressPeptideParser", //
program name
                                           bLearnTest ? LEARNTEST : RUNTEST);
// user info output
}

Tag *timestamp_start = new Tag("analysis_timestamp", True,
False);
timestamp_start->setAttributeValue("analysis", getName());
timestamp_start->setAttributeValue("time", time_);
timestamp_start->setAttributeValue("id", "1");
Tag *timestamp_stop = new Tag("analysis_timestamp", False, True);

Tag *timestamp = new Tag("xpressratio_timestamp", True, True);
sprintf(text3, "%d", pInput_.bXpressLight1);
timestamp->setAttributeValue("xpress_light", text3);

Tag *analysis_start = new Tag("analysis_summary", True, False);
analysis_start->setAttributeValue("analysis", getName());
analysis_start->setAttributeValue("time", time_);
Tag *analysis_stop = new Tag("analysis_summary", False, True);

Tag *summary = getSummaryTag(pInput_);

Tag *result_start = new Tag("analysis_result", True, False);
result_start->setAttributeValue("analysis", getName());
Tag *result_stop = new Tag("analysis_result", False, True);

TagFilter *xpress_filter = new TagFilter("analysis_result");
xpress_filter->enterRequiredAttributeVal("analysis", getName());

```

```

    TagFilter *xpress_summ_filter = new
TagFilter("analysis_timestamp");
    xpress_summ_filter->enterRequiredAttributeVal("analysis",
getName());
    TagFilter *xpress_summ = new TagFilter("analysis_summary");
    xpress_summ->enterRequiredAttributeVal("analysis", getName());

    char suffix[] = ".tmp";
    char *outfile = new char[strlen(xmlfile) + strlen(suffix) + 1];
    strcpy(outfile, xmlfile);
    strcat(outfile, suffix);

    Boolean first = False;
    Tag *xpress_ratio = NULL;
    int ratio_tags_written = 0;

    Boolean collected = False;

#ifdef USE_STD_MODS
    monoisotopic_ = False;          // unless proven otherwise
    Boolean mod_on = False;
    Array < Tag * >*modifications = NULL;
#endif
    Boolean top_hit = False;

    ifstream fin(xmlfile);
    if (!fin)
    {
        cerr << "error opening " << xmlfile << endl;
        exit(1);
    }
    char *nextline = new char[line_width_];
    while (fin.getline(nextline, line_width_))
    {
        //cout << "next: " << nextline << endl;

        data = strstr(nextline, "<");
        while (data != NULL)
        {
            tag = new Tag(data);

            //tag->write(cout);
            collected = False;

            setFilter(tag);

            if ((!xpress_filter->filter(tag) && !xpress_summ_filter-
>filter(tag) && !xpress_summ->filter(tag))
                {
                    if (tag->isStart() && !strcmp(tag->getName(),
"msms_pipeline_analysis"))
                    {
                        delete analysis_start;
                        analysis_start = NULL;
                        delete summary;

```

```

summary = NULL;
delete analysis_stop;
analysis_stop = NULL;

#ifdef USE_STD_MODS
for (int k = 0; k < 26; k++)
{
    light_label_masses_[k] = 0.0;
    heavy_label_masses_[k] = 0.0;
}
heavy_nterm_mass_ = 0.0;
light_nterm_mass_ = 0.0;
heavy_cterm_mass_ = 0.0;
light_cterm_mass_ = 0.0;
#endif

}
else if (tag->isStart() && !strcmp(tag->getName(),
"msms_run_summary"))
{

    pInput_.iAnalysisFirstScan = 1;

    first = True;
} // msms_summ
else if (tag->isEnd() && !strcmp(tag->getName(),
"search_summary"))
{

#ifdef USE_STD_MODS
// check label tags here
for (int k = 0; k < (int) strlen(pInput_.szXpressResidues);
k++)
{
    if (pInput_.szXpressResidues[k] == 'n')
    {
        if (heavy_nterm_mass_ > 0.0 && light_nterm_mass_ ==
0.0)
        {
            if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'n') != 0)
                light_nterm_mass_ = heavy_nterm_mass_ -
pInput_.dXpressMassDiff1;
            else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'n') != 0)
                light_nterm_mass_ = heavy_nterm_mass_ -
pInput_.dXpressMassDiff2;
            else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'n') != 0)
                light_nterm_mass_ = heavy_nterm_mass_ -
pInput_.dXpressMassDiff3;
            else
                light_nterm_mass_ = ResidueMass::getMass('n',
monoisotopic_);
        }
        else if (heavy_nterm_mass_ == 0.0 &&
light_nterm_mass_ > 0.0)

```

```

        {
            //must switch them around
            if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'n') != 0) {
                heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff1;
            }
            else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'n') != 0) {
                heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff2;
            }
            else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'n') != 0) {
                heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff3;
            }
            else if (pInput_.dXpressMassDiff1 < 0.0 &&
strchr(pInput_.szXpressResidues1, 'n') != 0) {
                heavy_nterm_mass_ = light_nterm_mass_;
                light_nterm_mass_ = heavy_nterm_mass_ +
pInput_.dXpressMassDiff1;
            }
            else if (pInput_.dXpressMassDiff2 < 0.0 &&
strchr(pInput_.szXpressResidues2, 'n') != 0) {
                heavy_nterm_mass_ = light_nterm_mass_;
                light_nterm_mass_ = heavy_nterm_mass_ +
pInput_.dXpressMassDiff2;
            }
            else if (pInput_.dXpressMassDiff3 < 0.0 &&
strchr(pInput_.szXpressResidues3, 'n') != 0) {
                heavy_nterm_mass_ = light_nterm_mass_;
                light_nterm_mass_ = heavy_nterm_mass_ +
pInput_.dXpressMassDiff3;
            }
            else {
                heavy_nterm_mass_ = light_nterm_mass_;
                light_nterm_mass_ = ResidueMass::getMass('n',
monoisotopic_);
            }
        }
        }
        else if (heavy_nterm_mass_ == 0.0 &&
light_nterm_mass_ == 0.0)
        {
            if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'n') != 0)
            {
                light_nterm_mass_ = ResidueMass::getMass('n',
monoisotopic_);
                heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff1;
            }
            else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'n') != 0)
            {
                light_nterm_mass_ = ResidueMass::getMass('n',
monoisotopic_);

```

```

        heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff2;
    }
    else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'n') != 0)
    {
        light_nterm_mass_ = ResidueMass::getMass('n',
monoisotopic_);
        heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff3;
    }
    else
    {
        // error
        cout << "nterm: " << light_nterm_mass_ << " vs "
<< heavy_nterm_mass_ << endl;
        exit(1);
    }
}
// must override if user defined
else
{
    if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'n') != 0)
        heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff1;
    else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'n') != 0)
        heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff2;
    else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'n') != 0)
        heavy_nterm_mass_ = light_nterm_mass_ +
pInput_.dXpressMassDiff3;
}
} // n case
else if (pInput_.szXpressResidues[k] == 'c')
{
    if (heavy_cterm_mass_ > 0.0 && light_cterm_mass_ ==
0.0)
    {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'c') != 0)
            light_cterm_mass_ = heavy_cterm_mass_ -
pInput_.dXpressMassDiff1;
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'c') != 0)
            light_cterm_mass_ = heavy_cterm_mass_ -
pInput_.dXpressMassDiff2;
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'c') != 0)
            light_cterm_mass_ = heavy_cterm_mass_ -
pInput_.dXpressMassDiff3;
        else
            light_cterm_mass_ = ResidueMass::getMass('c',
monoisotopic_);

```

```

    }
    else if (heavy_ctype_mass_ == 0.0 &&
light_ctype_mass_ > 0.0)
    {
        //must switch them around
        //must switch them around
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'c') != 0) {
            heavy_ctype_mass_ = light_ctype_mass_ +
pInput_.dXpressMassDiff1;
        }
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'c') != 0) {
            heavy_ctype_mass_ = light_ctype_mass_ +
pInput_.dXpressMassDiff2;
        }
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'c') != 0) {
            heavy_ctype_mass_ = light_ctype_mass_ +
pInput_.dXpressMassDiff3;
        }
        else if (pInput_.dXpressMassDiff1 < 0.0 &&
strchr(pInput_.szXpressResidues1, 'c') != 0) {
            heavy_ctype_mass_ = light_ctype_mass_;
            light_ctype_mass_ = heavy_ctype_mass_ +
pInput_.dXpressMassDiff1;
        }
        else if (pInput_.dXpressMassDiff2 < 0.0 &&
strchr(pInput_.szXpressResidues2, 'c') != 0) {
            heavy_ctype_mass_ = light_ctype_mass_;
            light_ctype_mass_ = heavy_ctype_mass_ +
pInput_.dXpressMassDiff2;
        }
        else if (pInput_.dXpressMassDiff3 < 0.0 &&
strchr(pInput_.szXpressResidues3, 'c') != 0) {
            heavy_ctype_mass_ = light_ctype_mass_;
            light_ctype_mass_ = heavy_ctype_mass_ +
pInput_.dXpressMassDiff3;
        }
        else {
            heavy_ctype_mass_ = light_ctype_mass_;
            light_ctype_mass_ = ResidueMass::getMass('c',
monoisotopic_);
        }
    }
    else if (heavy_ctype_mass_ == 0.0 &&
light_ctype_mass_ == 0.0)
    {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'c') != 0)
        {
            light_ctype_mass_ = ResidueMass::getMass('c',
monoisotopic_);
            heavy_ctype_mass_ = light_ctype_mass_ +
pInput_.dXpressMassDiff1;
        }
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'c') != 0)

```

```

        {
            light_ctype_mass_ = ResidueMass::getMass('c',
monoisotopic_);
            heavy_ctype_mass_ = light_ctype_mass_ +
pInput_.dXpressMassDiff2;
        }
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'c') != 0)
        {
            light_ctype_mass_ = ResidueMass::getMass('c',
monoisotopic_);
            heavy_ctype_mass_ = light_ctype_mass_ +
pInput_.dXpressMassDiff3;
        }
        else
        {
            // error
            cout << "ctype: " << light_ctype_mass_ << " vs "
<< heavy_ctype_mass_ << endl;
            exit(1);
        }
    }
    // must override if user defined
    else
    {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'c') != 0)
            heavy_ctype_mass_ = light_nctype_mass_ +
pInput_.dXpressMassDiff1;
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'c') != 0)
            heavy_ctype_mass_ = light_nctype_mass_ +
pInput_.dXpressMassDiff2;
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'c') != 0)
            heavy_ctype_mass_ = light_nctype_mass_ +
pInput_.dXpressMassDiff3;
    }

    } // c case
    else
    {
        char next_res = pInput_.szXpressResidues[k];
        if (heavy_label_masses_[next_res - 'A'] > 0.0 &&
light_label_masses_[next_res - 'A'] == 0.0)
        {
            if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, next_res) != 0)
                light_label_masses_[next_res - 'A'] =
heavy_label_masses_[next_res - 'A'] - pInput_.dXpressMassDiff1;
            else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, next_res) != 0)
                light_label_masses_[next_res - 'A'] =
heavy_label_masses_[next_res - 'A'] - pInput_.dXpressMassDiff2;
            else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, next_res) != 0)

```

```

        light_label_masses_[next_res - 'A'] =
heavy_label_masses_[next_res - 'A'] - pInput_.dXpressMassDiff3;
    else
        light_label_masses_[next_res - 'A'] =
ResidueMass::getMass(next_res, monoisotopic_);
    }
    else if (heavy_label_masses_[next_res - 'A'] == 0.0
&& light_label_masses_[next_res - 'A'] > 0)
    {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, next_res) != 0) {
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff1;
        }
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, next_res) != 0) {
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff2;
        }
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, next_res) != 0) {
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff3;
        }
        else if (pInput_.dXpressMassDiff1 < 0.0 &&
strchr(pInput_.szXpressResidues1, next_res) != 0) {
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'];
            light_label_masses_[next_res - 'A'] =
heavy_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff1;
        }
        else if (pInput_.dXpressMassDiff2 < 0.0 &&
strchr(pInput_.szXpressResidues2, next_res) != 0) {
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'];
            light_label_masses_[next_res - 'A'] =
heavy_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff2;
        }
        else if (pInput_.dXpressMassDiff3 < 0.0 &&
strchr(pInput_.szXpressResidues3, next_res) != 0) {
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'];
            light_label_masses_[next_res - 'A'] =
heavy_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff3;
        }
        else {
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'];
            light_label_masses_[next_res - 'A'] =
ResidueMass::getMass(next_res, monoisotopic_);
        }
    }
    else if (heavy_label_masses_[next_res - 'A'] == 0.0
&& light_label_masses_[next_res - 'A'] == 0)
    {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, next_res) != 0)

```

```

        {
            light_label_masses_[next_res - 'A'] =
ResidueMass::getMass(next_res, monoisotopic_);
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff1;
        }
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, next_res) != 0)
        {
            light_label_masses_[next_res - 'A'] =
ResidueMass::getMass(next_res, monoisotopic_);
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff2;
        }
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues3, next_res) != 0)
        {
            light_label_masses_[next_res - 'A'] =
ResidueMass::getMass(next_res, monoisotopic_);
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff3;
        }
        else
        {
            // error
            cout << "label " << next_res << ": " <<
                light_label_masses_[next_res - 'A'] << " vs "
<<
                heavy_label_masses_[next_res - 'A'] << endl;
            exit(1);
        }
    }
    // must override if user defined
    else if (heavy_label_masses_[next_res - 'A'] <= 0.0)
    {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, next_res) != 0)
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff1;
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, next_res) != 0)
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff2;
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, next_res) != 0)
            heavy_label_masses_[next_res - 'A'] =
light_label_masses_[next_res - 'A'] + pInput_.dXpressMassDiff3;
    }

    //          cout << "setting light to " <<
light_label_masses_[next_res-'A'] << " and heavy to " <<
heavy_label_masses_[next_res-'A'] << " for " << next_res << endl;
    }
    // aa case

    }
    // next label

#endif
}
#ifdef USE_STD_MODS

```

```

        else if (tag->isStart() && !strcmp(tag->getName(),
"search_summary"))
        {
            monoisotopic_ = !strcmp(tag-
>getAttributeValue("precursor_mass_type"), "monoistotopic");
        }
        // have a modification worth recording here
        else if (tag->isStart()
            && !strcmp(tag->getName(), "terminal_modification")
            && strchr(pInput_.szXpressResidues, tag-
>getAttributeValue("terminus")[0]) != NULL)
        {
            char next_res = tag->getAttributeValue("terminus")[0];
            double nextmass = atof(tag->getAttributeValue("mass"));
            if (next_res == 'n')
            {
                if (!strcmp(tag->getAttributeValue("variable"), "Y"))
                {
                    // must be heavy
                    if (heavy_nterm_mass_ > 0.0)
                    {
                        cout << "WARNING: Found more than one variable mod on
N-terminus." << endl;
                        if (nextmass > heavy_nterm_mass_) {
                            light_nterm_mass_ = heavy_nterm_mass_;
                            heavy_nterm_mass_ = nextmass;
                        }
                        else if (nextmass < light_nterm_mass_ ||
light_nterm_mass_
<= 0.0) {
                            light_nterm_mass_ = nextmass;
                        }
                        // error
                        //exit(1);
                    }
                    heavy_nterm_mass_ = nextmass;
                }
                // variable
            }
            else
            {
                // place in light for now
                light_nterm_mass_ = nextmass;
            }
        }
        // n terminal
        else if (next_res == 'c')
        {
            if (!strcmp(tag->getAttributeValue("variable"), "Y"))
            {
                // must be heavy
                if (heavy_cterm_mass_ > 0.0)
                {
                    cout << "WARNING: Found more than one variable mod on
N-terminus." << endl;
                    if (nextmass > heavy_cterm_mass_) {
                        light_cterm_mass_ = heavy_cterm_mass_;
                        heavy_cterm_mass_ = nextmass;
                    }
                    else if (nextmass < light_cterm_mass_ ||
light_cterm_mass_
<= 0.0) {
                        light_cterm_mass_ = nextmass;
                    }
                }
            }
        }
    }
}

```

```

    }
    // error
    //exit(1);
    }
    heavy_ctype_mass_ = nextmass;
} // variable
else
{
    // place in light for now
    light_ctype_mass_ = nextmass;
}

} // c

} // term modification
else if (tag->isStart()
    && !strcmp(tag->getName(), "aminoacid_modification")
    && strchr(pInput_.szXpressResidues, tag-
>getAttributeValue("aminoacid")[0]) != NULL)
{
    char next_res = tag->getAttributeValue("aminoacid")[0];
    double nextmass = atof(tag->getAttributeValue("mass"));
    if (!strcmp(tag->getAttributeValue("variable"), "Y"))
    {
        // must be heavy
        if (heavy_label_masses_[next_res - 'A'] > 0.0)
        {
            cout << "WARNING: Found more than one variable mod on
\'" << next_res << "\'." << endl;
            if (nextmass > heavy_label_masses_[next_res - 'A']) {
                light_label_masses_[next_res - 'A'] =
heavy_label_masses_[next_res - 'A'];
                heavy_label_masses_[next_res - 'A'] = nextmass;
            }
            else if (nextmass < light_label_masses_[next_res -
'A'] || light_label_masses_[next_res - 'A'] <= 0.0) {
                light_label_masses_[next_res - 'A'] = nextmass;
            }
            // error
            //exit(1);
        }
        heavy_label_masses_[next_res - 'A'] = nextmass;
    } // variable
else
{
    // place in light for now
    light_label_masses_[next_res - 'A'] = nextmass;
}

} // aa modification

#endif

#ifdef USE_STD_MODS
    if (modifications != NULL)
    {
        delete modifications;
    }
}

```

```

        modifications = NULL;
    }
#endif

    pInput_.iChargeState = -1;        // reset

#ifdef USE_STD_MODS
    if (modinfo_ != NULL)
        delete modinfo_;
    modinfo_ = NULL;
#endif

    //xpress_ratio = NULL;

}
// if not filtered

if (tag != NULL)
    delete tag;
data = strstr(data + 1, "<");
} // next tag

} // next line
fin.close();

if (bLearnTest || bRunTest)
{
    //
    // regression test stuff - bpratt Insilicos LLC, Nov 2005
    //

    if (bLearnTest)
    {
        writeTagListFile(testFileName, test_tags);
    }
    else
    {
        if (TagListComparator(test_tags, testFileName).compare())
        {
            cerr << "regression test FAILED in XPressPeptideParser!!!"
<< endl;
            exit(1);
        }
    }

    delete [] testFileName;
    for (int k = test_tags.length(); k--;)
    {
        delete test_tags[k];
    }
}

delete [] nextline;

delete timestamp_start;
delete timestamp_stop;
delete timestamp;

```

```

    if (analysis_start != NULL)
        delete analysis_start;
    if (analysis_stop != NULL)
        delete analysis_stop;
    if (summary != NULL)
        delete summary;
    delete result_start;
    delete result_stop;
    delete xpress_filter;
    delete xpress_summ_filter;
    delete xpress_summ;
}

void XpressPeptideParser::setFilter( Tag * tag)
{
    if (tag == NULL)
        return;

    if (filter_memory_)
    {
        filter_memory_ = False;
        filter_ = False;
    }

    if (!strcmp(tag->getName(), "spectrum_query"))
    {
        if (tag->isStart())
        {
            //tag->print();
            filter_ = True;
        }
        else
            filter_memory_ = True;
    }
}

Tag *XpressPeptideParser::getSummaryTag( const InputStruct &opts)
{
    Tag *output = new Tag("xpressratio_summary", True, True);
    char version[200];
    snprintf(version, sizeof(version), "%s (%s)", PROGRAM_VERSION,
szTPPVersionInfo);
    output->setAttributeValue("version", version);
    output->setAttributeValue("author", PROGRAM_AUTHOR);

    char text[100];
    if (opts.bUseSameScanRange)
        output->setAttributeValue("same_scan_range", "Y");
    else
        output->setAttributeValue("same_scan_range", "N");
    output->setAttributeValue("labeled_residues",
opts.szXpressResidues); /*jke */

    sprintf(text, "%d", opts.bXpressLight1);
    output->setAttributeValue("xpress_light", text);
}

```

```

if (opts.dXpressMassDiff3 > 0.0)
{
    sprintf(text, "%s,%0.2f %s,%0.2f %s,%0.2f",
            opts.szXpressResidues1, opts.dXpressMassDiff1,
            opts.szXpressResidues2, opts.dXpressMassDiff2,
            opts.szXpressResidues3, opts.dXpressMassDiff3); /*jke */
}
else if (opts.dXpressMassDiff2 > 0.0)
{
    sprintf(text, "%s,%0.2f %s,%0.2f",
            opts.szXpressResidues1, opts.dXpressMassDiff1,
            opts.szXpressResidues2, opts.dXpressMassDiff2); /*jke */
}
else
{
    sprintf(text, "%s,%0.2f", opts.szXpressResidues1,
opts.dXpressMassDiff1); /*jke */
}
output->setAttributeValue("massdiff", text);
sprintf(text, "%0.2f", opts.dMassTol);
output->setAttributeValue("masstol", text);

return output;
}

#ifdef USE_STD_MODS
Tag *XPressPeptideParser::getRatio()
{
    char szBuf[SIZE_BUF];
    // check modifications only to see if heavy or light or none
    Boolean light = False;
    Boolean heavy = False;
    Boolean unmod = False; // could be counted as light

    // calculate if correct, and mass diff between heavy and light
    double massdiff = 0.0;

    // MUST CHECK FOR N AND C TERMINAL MODS HERE FIRST.....
    if (modinfo_ != NULL)
    {
        if (strchr(pInput_.szXpressResidues, 'n') != NULL)
        {
            if (modinfo_->getNtermModMass() > 0.0)
            {
                double nextmass = modinfo_->getNtermModMass();
                if (nextmass - light_nterm_mass_ <= MOD_ERROR &&
light_nterm_mass_ - nextmass <= MOD_ERROR)
                    light = True;
                else if (nextmass - heavy_nterm_mass_ <= MOD_ERROR &&
heavy_nterm_mass_ - nextmass <= MOD_ERROR)
                    heavy = True;
            } // have modified terminus
        }
        else
        {

```

```

        if (fabs(light_nterm_mass_ - ResidueMass::getMass('n',
monoisotopic_)) < 0.001)
            light = True;
        else
            unmod = True;    // illegal
    }

    if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'n') != 0)
        massdiff += pInput_.dXpressMassDiff1;
    else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'n') != 0)
        massdiff += pInput_.dXpressMassDiff2;
    else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'n') != 0)
        massdiff += pInput_.dXpressMassDiff3;
    else
        massdiff += heavy_nterm_mass_ - light_nterm_mass_;

} // n
if (strchr(pInput_.szXpressResidues, 'c') != NULL)
{
    if (modinfo_->getCtermModMass() > 0.0)
    {
        double nextmass = modinfo_->getCtermModMass();
        if (nextmass - light_cterm_mass_ <= MOD_ERROR &&
light_cterm_mass_ - nextmass <= MOD_ERROR)
            light = True;
        else if (nextmass - heavy_cterm_mass_ <= MOD_ERROR &&
heavy_cterm_mass_ - nextmass <= MOD_ERROR)
            heavy = True;
    } // have modified terminus
    else
    {
        if (fabs(light_cterm_mass_ - ResidueMass::getMass('c',
monoisotopic_)) < 0.001)
            light = True;
        else
            unmod = True;    // illegal
    }
    if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'c') != 0)
        massdiff += pInput_.dXpressMassDiff1;
    else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'c') != 0)
        massdiff += pInput_.dXpressMassDiff2;
    else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'c') != 0)
        massdiff += pInput_.dXpressMassDiff3;
    else
        massdiff += heavy_cterm_mass_ - light_cterm_mass_;

} // c
} // some mods to look at

for (int k = 0; k < (int) strlen(pInput_.szPeptide); k++)
{

```

```

        if (strchr(pInput_.szXpressResidues, pInput_.szPeptide[k]) !=
NULL)
    {
        // have a labeled aa
        double nextmass = modinfo_ == NULL ? 0.0 : modinfo_
>getModifiedResidueMass(k);
        if (nextmass > 0.0)
        {
            // really is modified, find out if
light or heavy

            if (nextmass - light_label_masses_[pInput_.szPeptide[k] -
'A'] <=
                MOD_ERROR && light_label_masses_[pInput_.szPeptide[k] -
'A'] - nextmass <= MOD_ERROR)
                light = True;
            else if (nextmass -
heavy_label_masses_[pInput_.szPeptide[k] - 'A'] <= MOD_ERROR
                && heavy_label_masses_[pInput_.szPeptide[k] - 'A']
- nextmass <= MOD_ERROR)
                heavy = True;
        }
        // have modified aa
    else
    {
        light = True;
/*
        if (light_label_masses_[pInput_.szPeptide[k] - 'A'] ==
ResidueMass::getMass(pInput_.szPeptide[k],
monoisotopic_))
            light = True;
        else
            unmod = True;    // illegal
*/
    }
    if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, pInput_.szPeptide[k]) != 0)
        massdiff += pInput_.dXpressMassDiff1;
    else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, pInput_.szPeptide[k]) != 0)
        massdiff += pInput_.dXpressMassDiff2;
    else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, pInput_.szPeptide[k]) != 0)
        massdiff += pInput_.dXpressMassDiff3;
    else
        massdiff += heavy_label_masses_[pInput_.szPeptide[k] - 'A']
- light_label_masses_[pInput_.szPeptide[k] - 'A'];

    }
    // modified res
}
//printf("peptide=%s, unmod=%d, mdiff=%0.2f, light=%d, heavy=%d,
heavycterm=%f, lightcterm=%f\n", pInput_.szPeptide, unmod, massdiff,
light, heavy, heavy_cterm_mass_, light_cterm_mass_);
//DDS: Check if the light mod is on the terminals
if (!heavy && !light) {
    massdiff=0;
    if (strchr(pInput_.szXpressResidues, 'c') != NULL) {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'c') != 0)
            massdiff += pInput_.dXpressMassDiff1;

```

```

        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'c') != 0)
            massdiff += pInput_.dXpressMassDiff2;
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'c') != 0)
            massdiff += pInput_.dXpressMassDiff3;
        else
            massdiff += heavy_cterm_mass_ - light_cterm_mass_;
//    massdiff += heavy_cterm_mass_ - light_cterm_mass_;
        light = True;
        unmod = False;
    }
    if (strchr(pInput_.szXpressResidues, 'n') != NULL) {
        if (pInput_.dXpressMassDiff1 > 0.0 &&
strchr(pInput_.szXpressResidues1, 'n') != 0)
            massdiff += pInput_.dXpressMassDiff1;
        else if (pInput_.dXpressMassDiff2 > 0.0 &&
strchr(pInput_.szXpressResidues2, 'n') != 0)
            massdiff += pInput_.dXpressMassDiff2;
        else if (pInput_.dXpressMassDiff3 > 0.0 &&
strchr(pInput_.szXpressResidues3, 'n') != 0)
            massdiff += pInput_.dXpressMassDiff3;
        else
            massdiff += heavy_nterm_mass_ - light_nterm_mass_;
//    massdiff += heavy_nterm_mass_ - light_nterm_mass_;
        light = True;
        unmod = False;
    }
}

if (pInput_.iMetabolicLabeling
    || (!unmod && (light || heavy) && (!light || !heavy) && massdiff
> 0.0))
{
    // valid quant data to process

    if (pInput_.iMetabolicLabeling == 1)
    {
        pXpress_.dLightPeptideMass = pInput_.dPeptideMass;
        pXpress_.dHeavyPeptideMass = pInput_.dPeptideMass +
NITROGEN_COUNT(pInput_.szPeptide);
    }
    else if (pInput_.iMetabolicLabeling == 2)
    {
        pXpress_.dHeavyPeptideMass = pInput_.dPeptideMass;
        pXpress_.dLightPeptideMass = pInput_.dPeptideMass -
NITROGEN_COUNT(pInput_.szPeptide);
    }
    else
    {
        pXpress_.dLightPeptideMass = light ? pInput_.dPeptideMass :
pInput_.dPeptideMass - massdiff;
        pXpress_.dHeavyPeptideMass = heavy ? pInput_.dPeptideMass :
pInput_.dPeptideMass + massdiff;
    }

    // continue to process now....
    this->XPRESS_ANALYSIS(light);
}

```

```

pXpress_.iChargeState = pInput_.iChargeState;
pXpress_.bXpressLight1 = pInput_.bXpressLight1;
pXpress_.iMetabolicLabeling = pInput_.iMetabolicLabeling;
pXpress_.dMassTol = pInput_.dMassTol;

Tag *output = new Tag("xpressratio_result", True, True);
sprintf(szBuf, "%d", pXpress_.iLightFirstScan);
output->setAttributeValue("light_firstscan", szBuf);
sprintf(szBuf, "%d", pXpress_.iLightLastScan);
output->setAttributeValue("light_lastscan", szBuf);
sprintf(szBuf, "%0.3f", pXpress_.dLightPeptideMass);
output->setAttributeValue("light_mass", szBuf);
sprintf(szBuf, "%d", pXpress_.iHeavyFirstScan);
output->setAttributeValue("heavy_firstscan", szBuf);
sprintf(szBuf, "%d", pXpress_.iHeavyLastScan);
output->setAttributeValue("heavy_lastscan", szBuf);
sprintf(szBuf, "%0.3f", pXpress_.dHeavyPeptideMass);
output->setAttributeValue("heavy_mass", szBuf);
//WDN: Correlation coefficient stuff
sprintf(szBuf, "%0.3f", pXpress_.dCorrCoef);
output->setAttributeValue("confidence_score", szBuf);
//WDN: End correlation coefficient stuff

sprintf(szBuf, "%0.3f", pXpress_.dMassTol);
output->setAttributeValue("mass_tol", szBuf);
sprintf(szBuf, "%s", pXpress_.szQuan);
output->setAttributeValue("ratio", szBuf);

// now the flipped guy....
flipRatio(szBuf, szBuf);
output->setAttributeValue("heavy2light_ratio", szBuf);

sprintf(szBuf, "%0.2e", pXpress_.dLightArea);
output->setAttributeValue("light_area", szBuf);
sprintf(szBuf, "%0.2e", pXpress_.dHeavyArea);
output->setAttributeValue("heavy_area", szBuf);

if (pXpress_.dLightArea == 0.0)
{
    if (pXpress_.dHeavyArea > 0.0)
        sprintf(szBuf, "%0.1f", 999.0);
    else
        sprintf(szBuf, "%0.1f", -1.0);
}
else
{
    if (pXpress_.dHeavyArea != 0.0)
        sprintf(szBuf, "%0.2f", pXpress_.dLightArea /
pXpress_.dHeavyArea);
    else
        sprintf(szBuf, "%0.2f", 999.0);
}
output->setAttributeValue("decimal_ratio", szBuf);

//output->write(cout);

```

```

        return output;
    }

    return NULL;
}

#endif
#ifndef USE_STD_MODS

Tag *XpressPeptideParser::getRatio()
{
    int bHasXpress;
    int iLen;
    int ii;
    char szBuf[SIZE_BUF];

    /*
     * check to see if it has a labeled residue
     */
    bHasXpress = FALSE;
    iLen = (int) strlen(pInput_.szPeptide);
    for (ii = 0; ii < iLen; ii++)
    {
        if (strchr(pInput_.szXpressResidues1, pInput_.szPeptide[ii])
            || strchr(pInput_.szXpressResidues2, pInput_.szPeptide[ii])
            || strchr(pInput_.szXpressResidues3, pInput_.szPeptide[ii]))
        {
            bHasXpress = TRUE;
            break;
        }
    }

    /*
     * check if peptide contains Cys, is light or
     * heavy, and valid (not both light & heavy)
     */
    if (bHasXpress)
    {
        int iNumModRes1, iNumModRes2, iNumModRes3, bLightPeptide = TRUE,
bError;

        bError = FALSE;

        if ((ii + 1) == iLen || isalpha(pInput_.szPeptide[ii + 1]))
            bLightPeptide = TRUE;
        else
            bLightPeptide = FALSE;

        iNumModRes1 = 0;
        iNumModRes2 = 0;
        iNumModRes3 = 0;

        for (ii = 0; ii < iLen; ii++)

```

```

{
  if (strchr(pInput_.szXpressResidues1, pInput_.szPeptide[ii]))
  {
    iNumModRes1++;

    /*
     * Make sure not both heavy/light residue in peptide
     */
    if (bLightPeptide)
    {
      if (ii + 1 < iLen && !isalpha(pInput_.szPeptide[ii +
1]))
      {
        bError = TRUE;
        break;
      }
    }
    else
    {
      if (ii + 1 == iLen || isalpha(pInput_.szPeptide[ii +
1]))
      {
        bError = TRUE;
        break;
      }
    }
  }
  else if (strchr(pInput_.szXpressResidues2,
pInput_.szPeptide[ii]))
  {
    iNumModRes2++;

    /*
     * Make sure not both heavy/light residue in peptide
     */
    if (bLightPeptide)
    {
      if (ii + 1 < iLen && !isalpha(pInput_.szPeptide[ii +
1]))
      {
        bError = TRUE;
        break;
      }
    }
    else
    {
      if (ii + 1 == iLen || isalpha(pInput_.szPeptide[ii +
1]))
      {
        bError = TRUE;
        break;
      }
    }
  }
  else if (strchr(pInput_.szXpressResidues3,
pInput_.szPeptide[ii]))
  {

```

```

        iNumModRes3++;

        /*
         * Make sure not both heavy/light residue in peptide
         */
        if (bLightPeptide)
        {
            if (ii + 1 < iLen && !isalpha(pInput_.szPeptide[ii +
1]))
            {
                bError = TRUE;
                break;
            }
        }
        else
        {
            if (ii + 1 == iLen || isalpha(pInput_.szPeptide[ii +
1]))
            {
                bError = TRUE;
                break;
            }
        }
    }

    /*
     * Peptide is a valid, isotopically labelled peptide
     * with only heavy or light Cys residues
     */
    if (!bError)
    {
        char szTmp[SIZE_BUF];
        int iLightFirstScan, iLightLastScan, iHeavyFirstScan,
iHeavyLastScan;
        double dLightPeptideMass, dHeavyPeptideMass;

        /*
         * Calculate the corresponding heavy/light MH+ peptide mass
         */
        if (bLightPeptide)
        {
            dLightPeptideMass = pInput_.dPeptideMass;
            dHeavyPeptideMass =
iNumModRes1 +
            pInput_.dPeptideMass + pInput_.dXpressMassDiff1 *
            pInput_.dXpressMassDiff2 * iNumModRes2 +
            pInput_.dXpressMassDiff3 * iNumModRes3;
        }
        else
        {
            dHeavyPeptideMass = pInput_.dPeptideMass;
            dLightPeptideMass =
iNumModRes1 -
            pInput_.dPeptideMass - pInput_.dXpressMassDiff1 *
            pInput_.dXpressMassDiff2 * iNumModRes2 -
            pInput_.dXpressMassDiff3 * iNumModRes3;
        }
    }
}

```

```

}

pXpress_.dLightPeptideMass = dLightPeptideMass;
pXpress_.dHeavyPeptideMass = dHeavyPeptideMass;

XPRESS_ANALYSIS(bLightPeptide);

//strcpy(pXpress_.szXMLFile, pInput_.szXMLFile);
//strcpy(pXpress_.szOutFile, pInput_.szOutFile);
pXpress_.iChargeState = pInput_.iChargeState;
pXpress_.bXpressLight1 = pInput_.bXpressLight1;
pXpress_.dMassTol = pInput_.dMassTol;

/*
 * print the XPRESS link
 */

Tag *output = new Tag("xpressratio_result", True, True);
sprintf(szBuf, "%d", pXpress_.iLightFirstScan);
output->setAttributeValue("light_firstscan", szBuf);
sprintf(szBuf, "%d", pXpress_.iLightLastScan);
output->setAttributeValue("light_lastscan", szBuf);
sprintf(szBuf, "%0.3f", pXpress_.dLightPeptideMass);
output->setAttributeValue("light_mass", szBuf);
sprintf(szBuf, "%d", pXpress_.iHeavyFirstScan);
output->setAttributeValue("heavy_firstscan", szBuf);
sprintf(szBuf, "%d", pXpress_.iHeavyLastScan);
output->setAttributeValue("heavy_lastscan", szBuf);
sprintf(szBuf, "%0.3f", pXpress_.dHeavyPeptideMass);
output->setAttributeValue("heavy_mass", szBuf);
//sprintf(szBuf, "%d", pXpress_.iChargeState);
//output->setAttributeValue("charge", szBuf);

//if(pXpress_.bXpressLight1)
//  sprintf(szBuf, "%d", 1);
//else
//  sprintf(szBuf, "%d", 0);
//output->setAttributeValue("xpresslight", szBuf);
sprintf(szBuf, "%0.3f", pXpress_.dMassTol);
output->setAttributeValue("mass_tol", szBuf);
sprintf(szBuf, "%s", pXpress_.szQuan);
output->setAttributeValue("ratio", szBuf);

// now the flipped guy....
flipRatio(szBuf, szBuf);
output->setAttributeValue("heavy2light_ratio", szBuf);

sprintf(szBuf, "%0.2e", pXpress_.dLightArea);
output->setAttributeValue("light_area", szBuf);
sprintf(szBuf, "%0.2e", pXpress_.dHeavyArea);
output->setAttributeValue("heavy_area", szBuf);

if (pXpress_.dLightArea == 0.0)
{
    if (pXpress_.dHeavyArea > 0.0)
        sprintf(szBuf, "%0.1f", 999.0);
}

```

```

        else
            sprintf(szBuf, "%0.1f", -1.0);
    }
    else
    {
        //double max_decimal = 999.0;
        //double next_ratio =
pXpress_.dLightArea/pXpress_.dHeavyArea;
        //if(next_ratio > max_decimal)
        // next_ratio = max_decimal;

        //sprintf(szBuf, "%0.2f", next_ratio);
        if (pXpress_.dHeavyArea != 0.0)
            sprintf(szBuf, "%0.2f", pXpress_.dLightArea /
pXpress_.dHeavyArea);
        else
            sprintf(szBuf, "%0.2f", 999.0);
    }
    output->setAttributeValue("decimal_ratio", szBuf);

    //sprintf(szBuf, "%d", index);
    //output->setAttributeValue("index", szBuf);
    //cout << "returning..." << endl;
    output->write(cout);

    return output;
}
} // if have cys
return NULL;
} /*ADD_XPRESS */

#endif

/*
 * Return mass difference based on number of nitrogens in peptide
 * sequence for metabolic labeling
 */
double XPRESSPeptideParser::NITROGEN_COUNT(char *szPeptide)
{
    int i;
    int iMassDiff;
    int iLen;
    double dDiff = 0.99703477;

    // 14N = 14.003074020, 15N = 15.00010897, diff = 0.99703477
    // http://www.webelements.com/webelements/elements/text/N/isot.html

    iMassDiff = 0;
    iLen = (int) strlen(szPeptide);

    for (i = 0; i < iLen; i++)
    {
        if (szPeptide[i] == 'A')
            iMassDiff += 1;
        else if (szPeptide[i] == 'R')
            iMassDiff += 4;
    }
}

```

```

    else if (szPeptide[i] == 'N')
        iMassDiff += 2;
    else if (szPeptide[i] == 'D')
        iMassDiff += 1;
    else if (szPeptide[i] == 'C')
        iMassDiff += 1;
    else if (szPeptide[i] == 'E')
        iMassDiff += 1;
    else if (szPeptide[i] == 'Q')
        iMassDiff += 2;
    else if (szPeptide[i] == 'G')
        iMassDiff += 1;
    else if (szPeptide[i] == 'H')
        iMassDiff += 3;
    else if (szPeptide[i] == 'I')
        iMassDiff += 1;
    else if (szPeptide[i] == 'L')
        iMassDiff += 1;
    else if (szPeptide[i] == 'K')
        iMassDiff += 2;
    else if (szPeptide[i] == 'M')
        iMassDiff += 1;
    else if (szPeptide[i] == 'F')
        iMassDiff += 1;
    else if (szPeptide[i] == 'P')
        iMassDiff += 1;
    else if (szPeptide[i] == 'S')
        iMassDiff += 1;
    else if (szPeptide[i] == 'T')
        iMassDiff += 1;
    else if (szPeptide[i] == 'W')
        iMassDiff += 2;
    else if (szPeptide[i] == 'Y')
        iMassDiff += 1;
    else if (szPeptide[i] == 'V')
        iMassDiff += 1;
}

return (iMassDiff * dDiff);
}

/*
 * Reads mzXML files and get quantitation numbers
 */
void XPressPeptideParser::XPRESS_ANALYSIS(int bLightPeptide)
{
    int ii, ctScan, iLightStartScan, iLightEndScan, iHeavyStartScan,
    iHeavyEndScan, iStart, iEnd;

    //Array<double>* dLightMS, dLightFilteredMS, dHeavyMS,
    dHeavyFilteredMS;

    double dLightMass, dHeavyMass, H = 1.00794, dMassTol = 1.0,
    dLightQuanValue, dHeavyQuanValue;
    //          dLightMS[MAX_MS_SCAN],          /* Stores values of all MS
    scans for +1 mass */

```

```

//      dLightFilteredMS[MAX_MS_SCAN], /* Stored smoothed version
of above */
//      dHeavyMS[MAX_MS_SCAN],          /* Stores values of all MS
scans for +1 mass */
//      dHeavyFilteredMS[MAX_MS_SCAN], /* Stored smoothed version
of above */

//Array<double>* dLightMS = new Array<double>(MAX_MS_SCAN);
//Array<double>* dLightFilteredMS = new Array<double>(MAX_MS_SCAN);
//Array<double>* dHeavyMS = new Array<double>(MAX_MS_SCAN);
//Array<double>* dHeavyFilteredMS = new Array<double>(MAX_MS_SCAN);

if (pInput_.iChargeState < 1)
{
    printf(" Error, charge state = %d\n\n", pInput_.iChargeState);
    exit(EXIT_FAILURE);
}
else
{
    dLightMass = (H * (pInput_.iChargeState - 1) +
pXpress_.dLightPeptideMass) / pInput_.iChargeState;
    dHeavyMass = (H * (pInput_.iChargeState - 1) +
pXpress_.dHeavyPeptideMass) / pInput_.iChargeState;
}

/*
 * Clear all values
 */
for (int i = 0; i <= MAX_MS_SCAN; i++)
{
    dLightMS_[i] = 0;
    dHeavyMS_[i] = 0;
}
*/
memset(dLightMS_, 0, sizeof(dLightMS_));
memset(dHeavyMS_, 0, sizeof(dHeavyMS_));

iStart = pInput_.iFirstScan - 100;
iEnd = pInput_.iLastScan + 100;

if (iStart < pInput_.iAnalysisFirstScan)
    iStart = pInput_.iAnalysisFirstScan;
if (iEnd > pInput_.iAnalysisLastScan)
    iEnd = pInput_.iAnalysisLastScan;

//printf("iStart=%d, iEnd=%d, %d-%d\n", iStart, iEnd,
pInput_.iAnalysisFirstScan, pInput_.iAnalysisLastScan);
/*
 * Read all MS scan values
 */

//printf("LightMass %f HeavyMass %f Tol %f\n", dLightMass,
dHeavyMass, pInput_.dMassTol);
//getchar();

// Expensive loop, cache values.

```

```

double dLightMassLower = dLightMass - pInput_.dMassTol;
double dLightMassUpper = dLightMass + pInput_.dMassTol;
double dHeavyMassLower = dHeavyMass - pInput_.dMassTol;
double dHeavyMassUpper = dHeavyMass + pInput_.dMassTol;

RAMPREAL *pPeaks;
RAMPREAL fMass;
RAMPREAL fInten;
int n;

for (ctScan = iStart; ctScan <= iEnd; ctScan++)
{
    struct ScanHeaderStruct* pHeader = readHeaderCached(pCache_,
ctScan, fp_, index_[ctScan]);
    if (pHeader->msLevel != 1)
        continue;

    /*
     * Open a scan
     */
    pPeaks = readPeaksCached(pCache_, ctScan, fp_, index_[ctScan]);
    if (pPeaks == NULL)
        continue;

    // Binary search to the heaviest peak <= dLightMassLower.
    int l = 0, r = pHeader->peaksCount;
    while (l < r)
    {
        int m = l + (r - l) / 2;
        fMass = pPeaks[m*2];
        if (fMass == dLightMassLower)
        {
            l = r = m;
            break;
        }
        else if (fMass > dLightMassLower)
            r = m - 1;
        else
            l = m + 1;
    }
    n = l * 2;

    // Step forward to lightest peak >= dLightMassLower, if
necessary.
    fMass = pPeaks[n];
    if (fMass != -1 && fMass < dLightMassLower)
        n += 2;

    // Scan peak intensities between dLightMassLower and
dLightMassUpper.
    while (pPeaks[n] != -1)
    {
        fMass = pPeaks[n];
        n++;

        if (fMass > dHeavyMassUpper)
            break;
    }
}

```

```

fInten = pPeaks[n];
n++;

if (fMass <= dLightMassUpper)
{
    if (fInten > dLightMS_[ctScan])
    {
        dLightMS_[ctScan] = (double) fInten;
        //printf("light  %d  %f\n", ctScan, fInten);
    }
}
if (fMass >= dHeavyMassLower)
{
    if (fInten > dHeavyMS_[ctScan])
    {
        dHeavyMS_[ctScan] = (double) fInten;
        //printf("heavy  %d  %f\n", ctScan, fInten);
    }
}
}
}
/*for */

/*
 * Now that we have an MS profile of each of the two
 * light/heavy masses, we can calculate their intensities.
 * First, need to determine the start & end scan of the
 * entire peptide peak by smoothing MS profile.
 */
FILTER_MS(dLightMS_, dLightFilteredMS_, iStart, iEnd);
FILTER_MS(dHeavyMS_, dHeavyFilteredMS_, iStart, iEnd);

/*
 * Starting from the start and end scans read from .out
 * files, need to see the real start/end scan of eluting
 * peptide by looking at smoothed/filtered MS profile.
 */

/*
 * Get peptide start & end scans
 */
iLightStartScan = pInput_.iFirstScan;
iLightEndScan = pInput_.iLastScan;

/*
 * Backtrack to last MS scan for start
 */
for (ctScan = iLightStartScan; ctScan >= pInput_.iAnalysisFirstScan;
ctScan--)
{
    struct ScanHeaderStruct scanHeader;

    scanHeader.msLevel = readMsLevelCached(pCache_, ctScan, fp_,
index_[ctScan]);

    if (scanHeader.msLevel == 1)
        break;
}

```

```

}

if (abs(iLightStartScan - ctScan) > 10)
{
    printf(" Error - peptide %s at start scan %d with previous MS
scan at %d\n",
        pInput_.szPeptide, iLightStartScan, ctScan);
    printf(" ... the gap is too large.\n\n");
    exit(EXIT_FAILURE);
}
else
    iLightStartScan = ctScan;

/*
 * Backtrack to last MS scan for end
 */
for (ctScan = iLightEndScan; ctScan >= pInput_.iAnalysisFirstScan;
ctScan--)
{
    struct ScanHeaderStruct scanHeader;

    scanHeader.msLevel = readMsLevelCached(pCache_, ctScan, fp_,
index_[ctScan]);

    if (scanHeader.msLevel == 1)
        break;
}

if (abs(iLightEndScan - ctScan) > 15)
{
    printf(" Error - peptide %s at end scan %d with previous MS scan
at %d\n",
        pInput_.szPeptide, iLightStartScan, ctScan);
    printf(" ... the gap is too large (>15).\n\n");
    exit(EXIT_FAILURE);
}
else
    iLightEndScan = ctScan;

if (iLightStartScan < pInput_.iAnalysisFirstScan)
    iLightStartScan = pInput_.iAnalysisFirstScan;
if (iLightEndScan > pInput_.iAnalysisLastScan)
    iLightEndScan = pInput_.iAnalysisLastScan;

iHeavyStartScan = iLightStartScan;
iHeavyEndScan = iLightEndScan;

//printf("full %d-%d    light %d-%d    heavy %d-%d\n",
pInput_.iAnalysisLastScan, pInput_.iAnalysisFirstScan, iLightStartScan,
iLightEndScan, iHeavyStartScan, iHeavyEndScan);
if (!(pInput_.bUseSameScanRange)) /* don't use same scan range */
{
    FIND_ENDPOINTS(dLightFilteredMS_, &iLightStartScan,
&iLightEndScan,
                pInput_.iAnalysisFirstScan,
pInput_.iAnalysisLastScan);
}

```

```

        FIND_ENDPOINTS(dHeavyFilteredMS_, &iHeavyStartScan,
&iHeavyEndScan,
                    pInput_.iAnalysisFirstScan,
pInput_.iAnalysisLastScan);
    /*
    * Make sure not at zero point since using
    * filtered data to find endpoints
    */
    if (dLightMS_[iLightStartScan] != 0.0)
        iLightStartScan++;
    if (dLightMS_[iLightEndScan] == 0.0)
        iLightEndScan--;
    if (dHeavyMS_[iHeavyStartScan] != 0.0)
        iHeavyStartScan++;
    if (dHeavyMS_[iHeavyEndScan] == 0.0)
        iHeavyEndScan--;
}
else if (pInput_.bUseFixedScanRange == 1) /* just take +-
.iFixedScanRange scans from ID scan */
{
    if (bLightPeptide)
    {
        iLightStartScan -= pInput_.iFixedScanRange;
        if (iLightStartScan < pInput_.iAnalysisFirstScan)
            iLightStartScan = pInput_.iAnalysisFirstScan;
        iLightEndScan += pInput_.iFixedScanRange;
        if (iLightEndScan > pInput_.iAnalysisLastScan)
            iLightEndScan = pInput_.iAnalysisLastScan;

        iHeavyStartScan = iLightStartScan;
        iHeavyEndScan = iLightEndScan;
    }
    else
    {
        iHeavyStartScan -= pInput_.iFixedScanRange;
        if (iHeavyStartScan < pInput_.iAnalysisFirstScan)
            iHeavyStartScan = pInput_.iAnalysisFirstScan;
        iHeavyEndScan += pInput_.iFixedScanRange;
        if (iHeavyEndScan > pInput_.iAnalysisLastScan)
            iHeavyEndScan = pInput_.iAnalysisLastScan;

        iLightStartScan = iHeavyStartScan;
        iLightEndScan = iHeavyEndScan;
    }
}
else if (pInput_.bUseFixedScanRange == 2) /* just take +-
.iFixedScanRange scans from apex */
{
    if (bLightPeptide)
    {
        FIND_ENDPOINTS_FIX(dLightFilteredMS_, &iLightStartScan,
&iLightEndScan, pInput_.iAnalysisFirstScan,
pInput_.iAnalysisLastScan);

        iHeavyStartScan = iLightStartScan;
        iHeavyEndScan = iLightEndScan;
    }
}

```

```

else
{
    FIND_ENDPOINTS_FIX(dHeavyFilteredMS_, &iHeavyStartScan,
                      &iHeavyEndScan, pInput_.iAnalysisFirstScan,
                      pInput_.iAnalysisLastScan);

    iLightStartScan = iHeavyStartScan;
    iLightEndScan = iHeavyEndScan;
}
}
else if (bLightPeptide) /* light identified, use same scan
range */
{
    FIND_ENDPOINTS(dLightFilteredMS_, &iLightStartScan,
&iLightEndScan,
                  pInput_.iAnalysisFirstScan,
pInput_.iAnalysisLastScan);

    if (dLightMS_[iLightStartScan] != 0.0)
        iLightStartScan++;
    if (dLightMS_[iLightEndScan] == 0.0)
        iLightEndScan--;

    iHeavyStartScan = iLightStartScan;
    iHeavyEndScan = iLightEndScan;
}
else /* heavy identified, use same scan
range */
{
    FIND_ENDPOINTS(dHeavyFilteredMS_, &iHeavyStartScan,
&iHeavyEndScan,
                  pInput_.iAnalysisFirstScan,
pInput_.iAnalysisLastScan);

    if (dHeavyMS_[iHeavyStartScan] != 0.0)
        iHeavyStartScan++;
    if (dHeavyMS_[iHeavyEndScan] == 0.0)
        iHeavyEndScan--;

    iLightStartScan = iHeavyStartScan;
    iLightEndScan = iHeavyEndScan;
}
//WDN: Correlation coefficient stuff
double corrCoef;
LinearRegression *lr = new LinearRegression(dLightMS_, dHeavyMS_,
iLightStartScan, iLightEndScan);
corrCoef = lr->getCoefCorrel();
pXpress_.dCorrCoef = corrCoef;
delete lr;
//WDN: End correlation coefficient stuff

dLightQuanValue = 0.0;
dHeavyQuanValue = 0.0;

for (ii = iLightStartScan; ii <= iLightEndScan; ii++)
    dLightQuanValue += dLightMS_[ii];
for (ii = iHeavyStartScan; ii <= iHeavyEndScan; ii++)

```

```

    dHeavyQuanValue += dHeavyMS_[ii];

    if (pInput_.bXpressLight1 == 1)
    {
        if (dLightQuanValue == 0.0)
            sprintf(pXpress_.szQuan, "1:INF");
        else
            sprintf(pXpress_.szQuan, "1:%0.2f", dHeavyQuanValue /
dLightQuanValue);
    }
    else if (pInput_.bXpressLight1 == 2)
    {
        if (dHeavyQuanValue == 0.0)
            sprintf(pXpress_.szQuan, "INF:1");
        else
            sprintf(pXpress_.szQuan, "%0.2f:1", dLightQuanValue /
dHeavyQuanValue);
    }
    else
    {
        if (dLightQuanValue == 0.0 && dHeavyQuanValue == 0.0)
            sprintf(pXpress_.szQuan, "?:?");
        else if (dLightQuanValue > dHeavyQuanValue)
            sprintf(pXpress_.szQuan, "1:%0.2f", dHeavyQuanValue /
dLightQuanValue);
        else
            sprintf(pXpress_.szQuan, "%0.2f:1", dLightQuanValue /
dHeavyQuanValue);
    }

    pXpress_.dLightArea = dLightQuanValue;
    pXpress_.dHeavyArea = dHeavyQuanValue;

    pXpress_.iLightFirstScan = iLightStartScan;
    pXpress_.iLightLastScan = iLightEndScan;
    pXpress_.iHeavyFirstScan = iHeavyStartScan;
    pXpress_.iHeavyLastScan = iHeavyEndScan;
    //delete dLightMS;
    //delete dLightFilteredMS;
    //delete dHeavyMS;
    //delete dHeavyFilteredMS;
}

/*XPRESS_ANALYSIS */

void XPressPeptideParser::flipRatio(char *ratio, char *flipped)
{
    // find the : and 1
    double left, right;
    if (strstr(ratio, "?:?") != NULL)
        return;
    sscanf(ratio, "%lf:%lf", &left, &right);
    if (left == 1.0)
        ;
    else if (left == 0.0)
        left = 999;
}

```

```

else if (left >= 999.)
    left = 0.0;
else
    left = 1.0 / left;
if (right == 1.0)
    ;
else if (right >= 999.)
    right = 0.0;
else if (right == 0.0)
    right = 999;
else
    right = 1.0 / right;

if (left == 1.0)
    sprintf(flipped, "1:%0.2f", right);
else
    sprintf(flipped, "%0.2f:1", left);
}

void XPressPeptideParser::FIND_ENDPOINTS(
    double *pdFiltered,
    int *iPepStartScan,
    int *iPepEndScan,
    int iAnalysisFirstScan,
    int iAnalysisLastScan)
{
    int i, bLeftFalling, bRightFalling;

    /*
     * Find start and end scan of peak in the MS profile
     */

    /*
     * check to see if pWhichFilterd is rising or falling
     * as iPepStartScan is decreased.
     */
    i = 1;
    while (pdFiltered[*iPepStartScan] == pdFiltered[*iPepStartScan - i]
&&
        *iPepStartScan - i > 0)
    {
        i++;
    }
    if (pdFiltered[*iPepStartScan] > pdFiltered[*iPepStartScan - i])
        bLeftFalling = TRUE;
    else
        bLeftFalling = FALSE;

    i = 1;
    while (pdFiltered[*iPepEndScan] == pdFiltered[*iPepEndScan + i] &&
        *iPepStartScan + i < iAnalysisLastScan)
    {
        i++;
    }
    if (pdFiltered[*iPepEndScan] > pdFiltered[*iPepEndScan + i])
        bRightFalling = TRUE;
    else

```

```

    bRightFalling = FALSE;

    if (bLeftFalling == FALSE && bRightFalling == FALSE)
    {
        /*
        * error in valley of 2 peaks - just sum up from start to end
scan
        */
    }
    else if (bLeftFalling == TRUE && bRightFalling == TRUE)
    {
        /*
        * at a peak ... continue down left side to find end
        */
        while (*iPepStartScan - 1 > iAnalysisFirstScan
            && pdFiltered[*iPepStartScan] >= pdFiltered[*iPepStartScan
- 1])
        {
            (*iPepStartScan)--;
        }

        /*
        * Backtrack if the end point is at a flat plateau
        */
        while (pdFiltered[*iPepStartScan] == pdFiltered[*iPepStartScan +
1])
            (*iPepStartScan)++;

        /*
        * at a peak ... continue down right side to find end
        */
        while (*iPepEndScan + 1 < iAnalysisLastScan
            && pdFiltered[*iPepEndScan] >= pdFiltered[*iPepEndScan +
1])
        {
            (*iPepEndScan)++;
        }

        /*
        * Backtrack if the end point is at a flat plateau
        */
        while (pdFiltered[*iPepEndScan] == pdFiltered[*iPepEndScan - 1])
            (*iPepEndScan)--;
    }
    else if (bLeftFalling == TRUE && bRightFalling == FALSE)
    {
        /*
        * Walk down to the left
        */
        while (*iPepStartScan > 0
            && pdFiltered[*iPepStartScan] >= pdFiltered[*iPepStartScan
- 1])
        {
            (*iPepStartScan)--;
        }

        /*

```

```

    * Backtrack if the end point is at a flat plateau
    */
1]) while (pdFiltered[*iPepStartScan] == pdFiltered[*iPepStartScan +
    (*iPepStartScan)++);

    /*
    * Need to walk over right hump and down other side of peak
    */
1]) while (*iPepEndScan + 1 < iAnalysisLastScan
    && pdFiltered[*iPepEndScan] <= pdFiltered[*iPepEndScan +
    {
    (*iPepEndScan)++;
    }
    while (*iPepEndScan + 1 < iAnalysisLastScan
    && pdFiltered[*iPepEndScan] >= pdFiltered[*iPepEndScan +
1]) {
    (*iPepEndScan)++;
    }

    /*
    * Backtrack if the end point is at a flat plateau
    */
    while (pdFiltered[*iPepEndScan] == pdFiltered[*iPepEndScan - 1])
        (*iPepEndScan)--;
}
else if (bLeftFalling == FALSE && bRightFalling == TRUE)
{
    /*
    * Need to walk over left hump and down other side of peak
    */
    while (*iPepStartScan > 0
        && pdFiltered[*iPepStartScan] <= pdFiltered[*iPepStartScan
- 1])
    {
    (*iPepStartScan)--;
    }
    while (*iPepStartScan > 0
        && pdFiltered[*iPepStartScan] >= pdFiltered[*iPepStartScan
- 1])
    {
    (*iPepStartScan)--;
    }

    /*
    * Backtrack if the end point is at a flat plateau
    */
1]) while (pdFiltered[*iPepStartScan] == pdFiltered[*iPepStartScan +
    (*iPepStartScan)++;

    /*
    * Walk down to the right
    */
    while (*iPepEndScan + 1 < iAnalysisLastScan

```

```

        && pdFiltered[*iPepEndScan] >= pdFiltered[*iPepEndScan +
1])
    {
        (*iPepEndScan)++;
    }

    /*
     * Backtrack if the end point is at a flat plateau
     */
    while (pdFiltered[*iPepEndScan] == pdFiltered[*iPepEndScan - 1])
        (*iPepEndScan)--;
}

if (*iPepStartScan < iAnalysisFirstScan)
    *iPepStartScan = iAnalysisFirstScan;
if (*iPepStartScan > iAnalysisLastScan)
    *iPepStartScan = iAnalysisLastScan;

if (*iPepEndScan < iAnalysisFirstScan)
    *iPepEndScan = iAnalysisFirstScan;
if (*iPepEndScan > iAnalysisLastScan)
    *iPepEndScan = iAnalysisLastScan;

if (*iPepStartScan >= *iPepEndScan)
    *iPepStartScan = *iPepEndScan - 1;
}

/*FIND_ENDPOINTS */

/*
 * Return fixed scan # from apex
 */
void XPressPeptideParser::FIND_ENDPOINTS_FIX(
    double *pdFiltered,
    int *iPepStartScan,
    int *iPepEndScan,
    int iAnalysisFirstScan,
    int iAnalysisLastScan)
{
    int i, bLeftFalling, bRightFalling;

    /*
     * Find peak apex and go return fixed amount (5) scans each way
     */

    /*
     * check to see if pWhichFilterd is rising or falling
     * as iPepStartScan is decreased.
     */
    i = 1;
    while (pdFiltered[*iPepStartScan] == pdFiltered[*iPepStartScan - i]
        && *iPepStartScan - i > 0)
        i++;
    if (pdFiltered[*iPepStartScan] > pdFiltered[*iPepStartScan - i])
        bLeftFalling = TRUE;
}

```

```

else
    bLeftFalling = FALSE;

i = 1;
while (pdFiltered[*iPepEndScan] == pdFiltered[*iPepEndScan + i]
    && *iPepStartScan + i < iAnalysisLastScan)
    i++;
if (pdFiltered[*iPepEndScan] > pdFiltered[*iPepEndScan + i])
    bRightFalling = TRUE;
else
    bRightFalling = FALSE;

    //printf("bLeftFalling=%d, bRightFalling=%d\n", bLeftFalling,
bRightFalling);

    if (bLeftFalling == FALSE && bRightFalling == FALSE)
    {
        /*
        * error in valley of 2 peaks - just sum up from start to end
scan
        */
    }
    else if (bLeftFalling == TRUE && bRightFalling == TRUE)
    {
        /*
        * at a peak ... continue down left side to find end
        */
        while (*iPepStartScan - 1 > iAnalysisFirstScan
            && pdFiltered[*iPepStartScan] >= pdFiltered[*iPepStartScan
- 1])
            (*iPepStartScan)--;

        *iPepEndScan = *iPepStartScan + 5;
        *iPepStartScan -= 5;
    }
    else if (bLeftFalling == TRUE && bRightFalling == FALSE)
    {
        /*
        * Need to walk over right hump and down other side of peak
        */
        while (*iPepEndScan + 1 < iAnalysisLastScan
            && pdFiltered[*iPepEndScan] <= pdFiltered[*iPepEndScan +
1])
            (*iPepEndScan)++;

        *iPepStartScan = *iPepEndScan - 5;
        *iPepEndScan += 5;
    }
    else if (bLeftFalling == FALSE && bRightFalling == TRUE)
    {
        /*
        * Need to walk over left hump and down other side of peak
        */
        while (*iPepStartScan > 0
            && pdFiltered[*iPepStartScan] <= pdFiltered[*iPepStartScan
- 1])

```

```

        (*iPepStartScan)--;

        *iPepEndScan = *iPepStartScan + 5;
        *iPepStartScan -= 5;
    }

    if (*iPepStartScan < iAnalysisFirstScan)
        *iPepStartScan = iAnalysisFirstScan;
    if (*iPepStartScan > iAnalysisLastScan)
        *iPepStartScan = iAnalysisLastScan;

    if (*iPepEndScan < iAnalysisFirstScan)
        *iPepEndScan = iAnalysisFirstScan;
    if (*iPepEndScan > iAnalysisLastScan)
        *iPepEndScan = iAnalysisLastScan;

    if (*iPepStartScan >= *iPepEndScan)
        *iPepStartScan = *iPepEndScan - 1;
}

/*FIND_ENDPOINTS_FIX */

/*
 * Use my standard filtering routine
 */
void XPressPeptideParser::FILTER_MS(double *dOrigMS, double
*dFilteredMS, int iStart, int iEnd)
{
    int maxScan = pInput_.iAnalysisLastScan;
    int i, iArraySize = maxScan * sizeof(double);
    //double dTmpFilter[maxScan];

/*
    Defines 5th order butterworth filter w/ cut-off frequency
    of 0.25 where 1.0 corresponde to half the sample rate.

    5th order, 0.075
    double a[FILTER_SIZE]={1.0000, -4.2380, 7.2344, -6.2125, 2.5821, -
0.4655},
        b[FILTER_SIZE]={0.0158, 0.0792, 0.1585, 0.1585, 0.0792,
0.0158};

    5th order, 0.10
    double a[FILTER_SIZE]={1.0000, -3.9845, 6.4349, -5.2536, 2.1651, -
0.3599},
        b[FILTER_SIZE]={0.0000598, 0.0002990, 0.0005980, 0.0005980,
0.0002990, 0.0000598};

    5th order, 0.15
    double a[FILTER_SIZE]={1.0000, -3.4789, 5.0098, -3.6995, 1.3942, -
0.2138},
        b[FILTER_SIZE]={0.0004, 0.0018, 0.0037, 0.0037, 0.0018,
0.0004};

    5th order, 0.20

```

```

    double a[FILTER_SIZE]={1.0000, -2.9754, 3.8060, -2.5453, 0.8811, -
0.1254},
        b[FILTER_SIZE]={0.0013, 0.0064, 0.0128, 0.0128, 0.0064,
0.0013};

    5th order, 0.25
    double a[FILTER_SIZE]={1.0, -2.4744, 2.8110, -1.7038, 0.5444, -
0.0723},
        b[FILTER_SIZE]={0.0033, 0.0164, 0.0328, 0.0328, 0.0164,
0.0033};
*/
    double a[FILTER_SIZE] = { 1.0000, -3.9845, 6.4349, -5.2536, 2.1651,
-0.3599 },
        b[FILTER_SIZE] = { 0.0000598, 0.0002990, 0.0005980,
0.0005980, 0.0002990, 0.0000598};

memset(dFilteredMS, 0, sizeof(double)*MAX_MS_SCAN);
memcpy(dTmpFilter_, dOrigMS, sizeof(double)*MAX_MS_SCAN);

// Add padding on either end to make sure the range we care about is
// filtered correctly.
iStart = iStart - 50;
if (iStart < 0)
    iStart = 0;
iEnd = iEnd + 50;
if (iEnd > maxScan)
    iEnd = maxScan;

/*
 * Pass MS profile through IIR low pass filter:
 *  $y(n) = b(1)*x(n) + b(2)*x(n-1) + \dots + b(nb+1)*x(n-nb)$ 
 *  $- a(2)*y(n-1) - \dots - a(na+1)*y(n-na)$ 
 */
for (i = iStart; i < iEnd; i++)
{
    int ii;
    int end = FILTER_SIZE;
    if (end > i + 1)
        end = i + 1;

    dFilteredMS[i] = b[0] * dTmpFilter_[i];

    for (ii = 1; ii < end; ii++)
    {
        dFilteredMS[i] += b[ii] * dTmpFilter_[i - ii];
        dFilteredMS[i] -= a[ii] * dFilteredMS[i - ii];
    }
}

/*
 * Filtered sequence is reversed and re-filtered resulting
 * in zero-phase distortion and double the filter order.
 */
for (i = iStart; i < iEnd; i++)
    dTmpFilter_[i] = dFilteredMS[iEnd - 1 - (i - iStart)];

```

```

memset(dFilteredMS, 0, sizeof(double)*MAX_MS_SCAN);
for (i = iStart; i < iEnd; i++)
{
    int ii;
    int end = FILTER_SIZE;
    if (end > i + 1)
        end = i + 1;

    dFilteredMS[i] = b[0] * dTmpFilter_[i];
    for (ii = 1; ii < end; ii++)
    {
        dFilteredMS[i] += b[ii] * dTmpFilter_[i - ii];
        dFilteredMS[i] -= a[ii] * dFilteredMS[i - ii];
    }
}

/*
 * Filtered sequence is reversed again
 */
for (i = iStart; i < iEnd; i++)
    dTmpFilter_[i] = dFilteredMS[iEnd - 1 - (i - iStart)];

memcpy(dFilteredMS, dTmpFilter_, sizeof(double)*MAX_MS_SCAN);
}
/*FILTER_MS */

```

## XPressPeptideParser / XPressPeptideParser.h

```

#ifndef XPRESS_PEP_PARSER_H
#define XPRESS_PEP_PARSER_H

/*

Program      : XPressPeptideParser
Author       : J.Eng and Andrew Keller <akeller@systemsbiology.org>
Date        : 11.27.02

```

added mzData support - Brian Pratt Insilicos LLC 2005

Primary data object holding all mixture distributions for each precursor ion charge

Copyright (C) 2003 Andrew Keller

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,

but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Andrew Keller  
Insitute for Systems Biology  
1441 North 34th St.  
Seattle, WA 98103 USA  
akeller@systemsbiology.org

\*/

```
#include <stdio.h>
#include <math.h>
#include <time.h>
```

```
#include "../Parser/Parser.h"
#include "TagFilter.h"
#include "Option.h"
//#include "MixtureModel.h"
#include "ramp.h"
#include "base64.h"
#include "constants.h"
#include "ResidueMass.h"
#include "ModificationInfo.h"
//WDN: correlation coefficient stuff
#include "linreg.h"
//WDN: end correlation coefficient stuff
```

```
//#define PROG_NAME "xpress"
```

```
#define PROGRAM_VERSION "2.1" // 2.1 adds mzData support - Brian
Pratt Insilicos LLC 2005
#define PROGRAM_AUTHOR "Jimmy Eng"
```

```
#define HTTP_TARGET "Win1"
#define CGI_BINARY "/cgi-bin/cgixpress3" /*jke */
#define CGI_SIC_BINARY "/cgi-bin/cgiSIC3"
#define DEFAULT_TOL 1.0
#define SIZE_BUF 8192
#define MAX_MS_SCAN 100000
#define FILTER_SIZE 6

#define TRUE 1
#define FALSE 0
```

```
struct XpressStruct
{
    int iLightFirstScan;
```

```

int    iLightLastScan;
int    iHeavyFirstScan;
int    iHeavyLastScan;
int    iMetabolicLabeling;
int    bXpressLight1;
double dLightPeptideMass;
double dHeavyPeptideMass;
double dLightArea;
double dHeavyArea;
//WDN: correlation coefficient stuff
double dCorrCoef;
//WDN: end correlation coefficient stuff
char   szQuan[128];

char   szXMLFile[SIZE_FILE];
double dMassTol;
int    iChargeState;
char   szOutFile[SIZE_FILE];
};

class XPressPeptideParser:public Parser
{
public:
    XPressPeptideParser(const char *xmlfile, const InputStruct &
options, const char *testMode);
    ~XPressPeptideParser();
    void   setFilter(Tag * tag);

protected:
    void readAllModMasses(const char *xmlfile);
    void   parse(const char *xmlfile);
    Tag   *getRatio();

    Tag   *getSummaryTag(const InputStruct &opts);
    void   XPRESS_ANALYSIS(int bLightPeptide);

    void   FILTER_MS(double *dOrigMS, double *dFilteredMS, int iStart,
int iEnd);
    //void FILTER_MS(Array<double> *dOrigMS,
//                  Array<double> *dFilteredMS);

    void   FIND_ENDPOINTS(double *pdFiltered,
                        int *iPepStartScan,
                        int *iPepEndScan,
                        int iAnalysisFirstScan,
                        int iAnalysisLastScan);

    void   FIND_ENDPOINTS_FIX(double *pdFiltered,
                        int *iPepStartScan,
                        int *iPepEndScan,
                        int iAnalysisFirstScan,
                        int iAnalysisLastScan);

    void   flipRatio(char *ratio, char *flipped);

```

```

double NITROGEN_COUNT(char *szPeptide);

ModelOptions modelOpts_;
ScoreOptions scoreOpts_;

char *testMode_; // regression test stuff - bpratt Insilicos
LLC, Nov 2005

RAMPFILE *fp_;
ramp_fileoffset_t *index_;
struct ScanCacheStruct* pCache_;
char mzXMLfile_[10000];

InputStruct pInput_;
XpressStruct pXpress_;

double dTmpFilter_[MAX_MS_SCAN],
dLightMS_[MAX_MS_SCAN],
dLightFilteredMS_[MAX_MS_SCAN],
dHeavyMS_[MAX_MS_SCAN], dHeavyFilteredMS_[MAX_MS_SCAN];

#ifdef USE_STD_MODS
ModificationInfo *modinfo_;
double light_label_masses_[26];
double heavy_label_masses_[26];
double light_nterm_mass_;
double heavy_nterm_mass_;
double light_cterm_mass_;
double heavy_cterm_mass_;
Boolean monoisotopic_;
#endif
};

#endif

```

### XPressProteinRatioParser/ XPressProteinRatioParser.cxx

```

#ifndef XPRESS_PEP_PARSER_H
#define XPRESS_PEP_PARSER_H

/*

Program      : XPressPeptideParser
Author       : J.Eng and Andrew Keller <akeller@systemsbiology.org>
Date        : 11.27.02

```

added mzData support - Brian Pratt Insilicos LLC 2005

Primary data object holding all mixture distributions for each precursor ion charge

Copyright (C) 2003 Andrew Keller

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Andrew Keller  
Insitute for Systems Biology  
1441 North 34th St.  
Seattle, WA 98103 USA  
akeller@systemsbiology.org

\*/

```
#include <stdio.h>
#include <math.h>
#include <time.h>
```

```
#include "../Parser/Parser.h"
#include "TagFilter.h"
#include "Option.h"
//#include "MixtureModel.h"
#include "ramp.h"
#include "base64.h"
#include "constants.h"
#include "ResidueMass.h"
#include "ModificationInfo.h"
//WDN: correlation coefficient stuff
#include "linreg.h"
//WDN: end correlation coefficient stuff
```

```
//#define PROG_NAME "xpress"
```

```
#define PROGRAM_VERSION "2.1" // 2.1 adds mzData support - Brian Pratt Insilicos LLC 2005
```

```
#define PROGRAM_AUTHOR "Jimmy Eng"
```

```
#define HTTP_TARGET "Win1"
#define CGI_BINARY "/cgi-bin/cgixpress3" /*jke */
#define CGI_SIC_BINARY "/cgi-bin/cgiSIC3"
#define DEFAULT_TOL 1.0
#define SIZE_BUF 8192
#define MAX_MS_SCAN 100000
#define FILTER_SIZE 6

#define TRUE 1
```



```

void    FIND_ENDPOINTS_FIX(double *pdFiltered,
                          int *iPepStartScan,
                          int *iPepEndScan,
                          int iAnalysisFirstScan,
                          int iAnalysisLastScan);

void    flipRatio(char *ratio, char *flipped);
double  NITROGEN_COUNT(char *szPeptide);

ModelOptions modelOpts_;
ScoreOptions scoreOpts_;

char *testMode_;    // regression test stuff - bpratt Insilicos
LLC, Nov 2005

RAMPFILE *fp_;
ramp_fileoffset_t *index_;
struct ScanCacheStruct* pCache_;
char  mzXMLfile_[10000];

InputStruct pInput_;
XpressStruct pXpress_;

double dTmpFilter_[MAX_MS_SCAN],
       dLightMS_[MAX_MS_SCAN],
       dLightFilteredMS_[MAX_MS_SCAN],
       dHeavyMS_[MAX_MS_SCAN], dHeavyFilteredMS_[MAX_MS_SCAN];

#ifdef USE_STD_MODS
ModificationInfo *modinfo_;
double  light_label_masses_[26];
double  heavy_label_masses_[26];
double  light_nterm_mass_;
double  heavy_nterm_mass_;
double  light_cterm_mass_;
double  heavy_cterm_mass_;
Boolean monoisotopic_;
#endif
};

#endif

```

### XPressProteinRatioParser/ XPressProteinRatioParser.h

```

/*
Program      : XPressRatioParser
Author       : Andrew Keller <akeller@systemsbiology.org>
              *Jimmy Eng (jeng@systemsbiology.org)
Date        : 11.27.02

Computes XPRESS ratios and errors for proteins, then overwrites

```

that information onto ProteinProphet XML

Copyright (C) 2003 Andrew Keller, Jimmy Eng

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Andrew Keller  
Insitute for Systems Biology  
1441 North 34th St.  
Seattle, WA 98103 USA  
akeller@systemsbiology.org

\*/

```
#ifndef X_PROT_RATIO_PARSER_H
#define X_PROT_RATIO_PARSER_H
```

```
#include <stdio.h>
#include <math.h>
#include <time.h>
```

```
#include "../Parser/Parser.h"
#include "TagFilter.h"
#include "../XPressGroupPeptideParser.h"
```

```
#define SIZE_PEPTIDE 128
#define SIZE_BUF 8192
```

```
class XPressProteinRatioParser : public Parser {
```

```
public:
```

```
    XPressProteinRatioParser(const char * protxmlfile, const char
*testMode);
```

```
    XPressProteinRatioParser(Array<const char*> &input_pepxmlfiles, const
peplist &peptides, double minpepprob ); // used by
XPressCGIProteinDisplay
```

```
    ~XPressProteinRatioParser();
    const RatioStruct &getRatio() const { // used by
XPressCGIProteinDisplay
```

```

    return pRatio_;
}

protected:

void parse(const char * protxmlfile);
void setFilter(Tag* tag);
void cachePepXML(); // read the search hits out of the pepXML files

char * getPeptideString(peplist* peps, const char * link);
Boolean getRatio(const peplist* peps, double minpepprob);
void setInputFiles(const char ** inputfiles);
void enterUnique(peplist* uniques, const char* next);

XPressGroupPeptideParser* parser_;
Array<const char *> input_pepxmlfiles_;
XPressRatioSearchHitCache *searchHits_; // caches the pepXML reads

char *testMode_; // regression test stuff - bpratt Insilicos LLC, Nov
2005

int iNumRawData_;
RatioStruct pRatio_;
Boolean heavy2light_;
};

#endif

```