Title

New Tools for Chemically Directed Proteomics

Permalink

https://escholarship.org/uc/item/2300q7xg

Author

Pitcher, Austin Arlo

Publication Date

2010

Peer reviewed|Thesis/dissertation

New Tools for Chemically Directed Proteomics

by

Austin Arlo Pitcher

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Chemistry

in the

graduate division

of the

University of California at Berkeley

Committee in charge:

Professor Carolyn R. Bertozzi, Chair
Professor Matthew B. Francis
Professor Jay D. Keasling

Fall 2010

New Tools for Chemically Directed Proteomics

Abstract

New Tools for Chemically Directed Proteomics

by

Austin Arlo Pitcher

Doctor of Philosophy in Chemistry

University of California at Berkeley

Professor Carolyn R. Bertozzi, Chair

In recent years, mass spectrometry has become a staple technique in biochemistry and molecular biology, with mass spectrometry based proteomics being one of its greatest successes. The standard method for determining protein identifications though the use of mass spectrometry involves a number of steps. First, a solution of whole proteins is digested with a protease, typically trypsin. The resulting peptides are then separated by liquid chromatography, and a full scan mass spectrum is obtained for each eluting fraction. As ions themselves produce little information that can be used to determine protein identifications, peptide ions are then selected for secondary fragmentation and a tandem mass spectrum is obtained. From this secondary spectrum, peptide sequence information can be obtained after comparison to proteome databases.

However, despite being a powerful tool for peptide identification, the traditional shotgun proteomics approach often suffers from limited sensitivity and a lack of reproducibility between replicate analyses. A major source of these limitations is due to the way in which ions are chosen for fragmentation. As unmodified peptide ions are virtually indistinguishable in a full scan mass spectrum, the vast majority of experiments select ions for fragmentation based solely on the signal intensity of each ion. In complex samples, this has the often undesired consequence of biasing the search towards the most abundant, though often uninteresting peptides. Furthermore, due to the stochastic nature of ion selection, it is often difficult to reproduce a list of protein identifications even if the same biological sample is used for multiple experiments. This dissertation focuses on the idea of using chemical tagging strategies to introduce information into a complex sample that can then be used to direct MS analysis away from the most abundant species and towards those most likely to be interesting in a given biological context. The technology developed is then applied to the study of

protein glycosylation, a type of protein post-translational modification ubiquitous in eukaryotic organisms.

In Chapter 1, current technologies for studying glycoproteins using mass spectrometry are surveyed. The emphasis in this chapter is on the use of unnatural sugar substrates for the metabolic engineering of glycan structures, and applications of metabolic engineering to glycoproteomics. This chapter also reviews the use of bioorthogonal reactions in the context of glycoproteomics. Finally, the standard workflow for proteomics experiments is examined and the concept of directed mass spectrometry is introduced. Chapter 2 proposes a method for using chemistry to add information to a biological system which can then be used to direct the MS analysis of biomolecules to bias analysis towards a subset of so-called “information-rich” ions. This system uses the distinctive isotopic distribution of a chemical label to perturb the isotopic envelope of a biomolecule in a way that is detectable in a full-scan mass spectrum. Coupled with a computational algorithm described in Chapter 3, we term this methodology the IsoStamp system.

The isotopic pattern searching algorithm introduced relies on the ability to accurately predict the isotopic envelope of a peptide solely from the molecular weight of the ion. Such a system is analyzed in Chapter 4, and the scope is extended to applications in glycobiology including the prediction of isotopic envelopes of biomolecules such as mucins, where a large percentage of the molecular weight is attributed to carbohydrate content. Potential weaknesses of current metabolic oligosaccharide engineering techniques as they are employed in mass spectrometry is that they typically require a secondary labeling step, and that unnatural sugars may not be incorporated into glycan structures at stoichiometric levels. Chapter 5 introduces an alternative approach whereby an isotopically labeled mixture of a natural substrate, GlcNAc, is fed to cells and is subsequently incorporated into N-glycan structures at stoichiometric levels. N-glycosylated peptides are then targeted for MS/MS analysis based on their isotopic distribution, and sites of modification are determined by comparison to a proteome database. Finally, Chapter 6 examines the future of isotopic labeling in biological mass spectrometry, suggesting a number of applications of the IsoStamp technology.

To my parents
in gratitude and love

Contents

Appendices

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I'd like to thank a number of people for their encouragement, guidance, and support—in completely random order.

I am extremely grateful to Carolyn for creating a fantastic working environment and giving me the intellectual freedom to work on any project that seemed interesting at the time. In addition to giving us complete creative control and being a seemingly endless well of knowledge, you keep the lab well stocked with brilliant scientists that make any project more manageable. I truly cannot imagine a better lab to work in.

That leads directly into my second person on my long list of people to thank: Brian Smart. The original idea to incorporate two bromine atoms into a peptide tag was all Brian. Beyond the initial idea, much of the work presented in this dissertation was performed in collaboration with Brian. His synthetic chemistry skills alone kept this project afloat during its infancy! Brian has not only been an excellent mentor and close collaborator, but has, along with his sweetheart of a wife Laura, become a close friend. I deeply appreciate all of the advice and support you have given me.

Kanna, a more recent addition to "team dibromide", has also been an absolute pleasure to work with and has done an job of excellent balancing out our subgroup. Kanna, your feedback has been extremely helpful, and I always enjoy your company both in and out of lab.

I also want to thank Sarah Hubbard, one of the early members of the Bertozzi proteomics team, for being a friend and mentor during my first years in the lab. In addition to pioneering proteomics in the Bertozzi group, Sarah's interest in business and technology rubbed off on me, and she pushed me to think about other ways to apply the technologies we already had. Our conversations about technology and innovation did much to shape the work presented here. Whether or not I asked for it, Sarah was always willing to volunteer her opinion, and she gave me some excellent advice early on in my graduate career. I'm glad she did.

Mike Boyce has also played a formative role in scientific progression. Mike is, without question, one of the best scientists I have ever met. During my first year of graduate school, Mike was an excellent mentor; since then, I have come to value his advice, his friendship, and his razor-sharp sense of humor. (Not to mention his company on late-night BART rides across the bay!)

Along with being a rainy-day lunch mate and last-minute snow-day companion, Mark Breidenbach has also been an excellent collaborator. That he decided to engineer the perfect strain of yeast to make our collaboration possible was extremely fortuitous, and I am lucky to have had the opportunity to work with him.

Ellie Smith, one of the more recent additions to room 831, has done an excellent job keeping me sane over the last few semesters. It's been great having another Minnesotan in our room!

Though none of the work is presented here, I've had the opportunity to work with Phung Gip over the past few months looking at the implications of glycobiology during the course of stem cell differentiation. Phung has been an absolute pleasure to work with, and I've enjoyed learning from her immensely.

In no particular order, I would also like to thank John Jewett for many thoughtful discussions; Jason Hudak for being my only other surviving classmate in our lab and for his camaraderie during qualifying exam season; Kimberly Beatty for the countless lunchtime discussions and her constant enthusiasm; residents of room 831 past and present: Jason Rush, Matt Hangauer, Andy Hseih, Ellie Smith, and Brian Smart; Kamil Godula and (*Professor*) Ramesh Jasti for their scientific advice, their constantly entertaining friendship, frequent beers, and the occasional trip to Las Vegas; Olga, Sia, Asia, Karen, and Cheryl for simply making things work—our lab would cease to function without you.

To everyone else in the Bertozzi group: thank you for making this such a great place to work. Every one of you has helped me with something at one time or another. You've set a high bar for any of my future colleagues.

I owe much of my current success to a some excellent mentors I had during my undergraduate years: Dr. Sreerama and Dr. Haglin. Dr. Lakshmaiah Sreerama, my undergraduate advisor and PI, has been a constant source of advice and enthusiasm. The only reason I'm at Berkeley at all is due to one of his many suggestions. Thank you! Dr. Kevin Haglin has also earned my heartfelt thanks. Dr. Haglin was the first professor to really get me excited about science, and for that I cannot thank him enough. It is difficult to express how much I appreciate your putting up with me all these years. You are both amazing teachers, and I am fortunate to have had the opportunity to work with both of you.

I am also very grateful to Dr. John Cronn and Dr. Timothy Schuh, two excellent professors within the biology department at St. Cloud State. Dr. Cronn encouraged me to take the diversity of classes that I did, and Dr. Schuh made them work for the major requirements. I particularly need to thank Dr. Cronn for being an excellent mentor—he has always been incredibly excited about science, and truly pushed me to grow as a person. Some of the books that he gave me as a freshman undergraduate remain among my favorites of all time.

There are huge number of people outside of academics that have helped me get to this point. My maternal grandparents, John and Myrna, played a large role in my upbringing. Grandpa John was the first person to give me a paying job doing construction over summer breaks. His physics and math background certainly rubbed off on me, and I still have books on particle physics and vector calculus that he gave me in middle school. I can't say that I understood them entirely then or now, but they were certainly good motivation for me. Without question, he is the person that taught me to love solving problems. He then proceeded to equip me with an excellent toolbox with which to solve them.

My paternal grandparents, Arlo and Elsie, were no less influential. I partially blame my love of traveling on you two. You have both been incredibly inspirational.

Last, but absolutely not least, are my immediate family: my parents, Steve and Chris, and my younger sister and brother, Morgan and Alex. There is no way I could have made it to this point without your support and encouragement. You have given me every opportunity I could have hoped for, and I am beyond lucky to have you as parents. I truly don't have the words to thank you for everything you have given me.

chapter one

The use of unnatural sugars in glycoproteomics

1.1 Historical perspective

The field of genomics, the earliest and most well-known of the "omics" sciences, refers to the study of the DNA sequences that make up an organism's genome. Advances in genomics have since laid groundwork for the other "omics" sciences, including transcriptomics and proteomics, referring to the study of all transcribed genes and all proteins synthesized by a cell under a specific set of biological conditions, respectively[1–5]. The experimental techniques culminating in the solution of the complete sequence of the human genome[6] have ushered in the so-called "post-genomic era" in which informatics techniques can be applied to the study of biological systems to gain a nuanced understanding of life at both the molecular and systems levels, with one of the ultimate goals being the discovery novel disease therapies[7–10].

The term "glycoproteome", then, refers to the complete repertoire of glycoproteins produced by cells under specific conditions of time, space, and environment[11]. Glycoproteomics thus refers to studies used to profile the glycoproteome. Like nucleic acids and proteins, glycans come in a diversity of structures that underlie a vast array of biological functions. In order to understand these functions it is essential that we understand these structures at a molecular level. However, in contrast the genomes, transcriptomes, and proteomes, glycans are not primary gene products. Rather than being synthesized based on a pre-existing template, glycans are constructed by through the action of a large number of glycosyltransferases that build up oligosaccharide structures from a their monosaccharide building blocks, the most common of which are shown in Figure 1-1. This complex and highly context-dependent biosynthetic machinery, combined with the diverse set of possible linkages within an individual glycan, leads to plethora of distinct and heterogeneous structures, with some illustrative examples shown in Figure 1-2. Glycan synthesis is further obfuscated by the fact that glycan composition can be highly sensitive to the metabolic state of the cell, such that

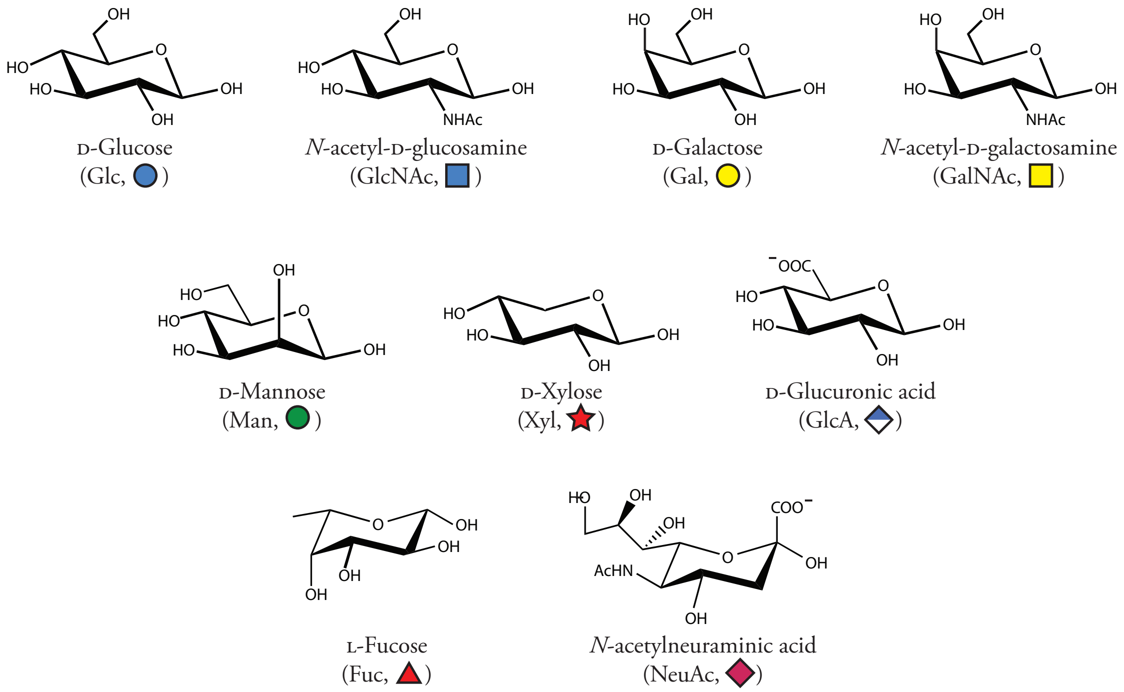


Figure 1-1: Common monosacharides found in vertebrate glycans along with their standard graphical representations. *N*-acetylneuraminic acid is the most common form of sialic acid. Figure adapted from Varki *et al.*, Figure 2.4[11].

genetically identical cells may produce different glycan structures depending on subtle differences in their local environments[12–14]. Additional complications arise from the fact that many of the most useful biochemical techniques—such as gene disruption or overexpression, GFP tagging for protein visualization, and DNA microarrays—are largely unavailable for their study. While disruption of genes responsible for glycosylation can disrupt glycan structure, redundancies among glycosyltransferases and embryonic lethality render mutant phenotypes onerous to interpret[15–17].

Despite the inherent difficulty in studying protein glycosylation, there are enticing reasons to do so. Protein glycosylation has been shown to mediate a huge number of cell-cell interactions[18–20] and is essential for a number of immune functions[21,22]. Glycosylation has also been shown to play integral roles in cancer and metastasis[23–27], cellular homeostasis[28–30], protein quality control[31,32], nutrient sensing[14,33–36], and development[37,38] in eukaryotic organisms. In addition to being physiologically essential, protein glycosylation is also abundant: informatics studies have suggested that as many as 50% of proteins in humans are subject to glycosylation[39].

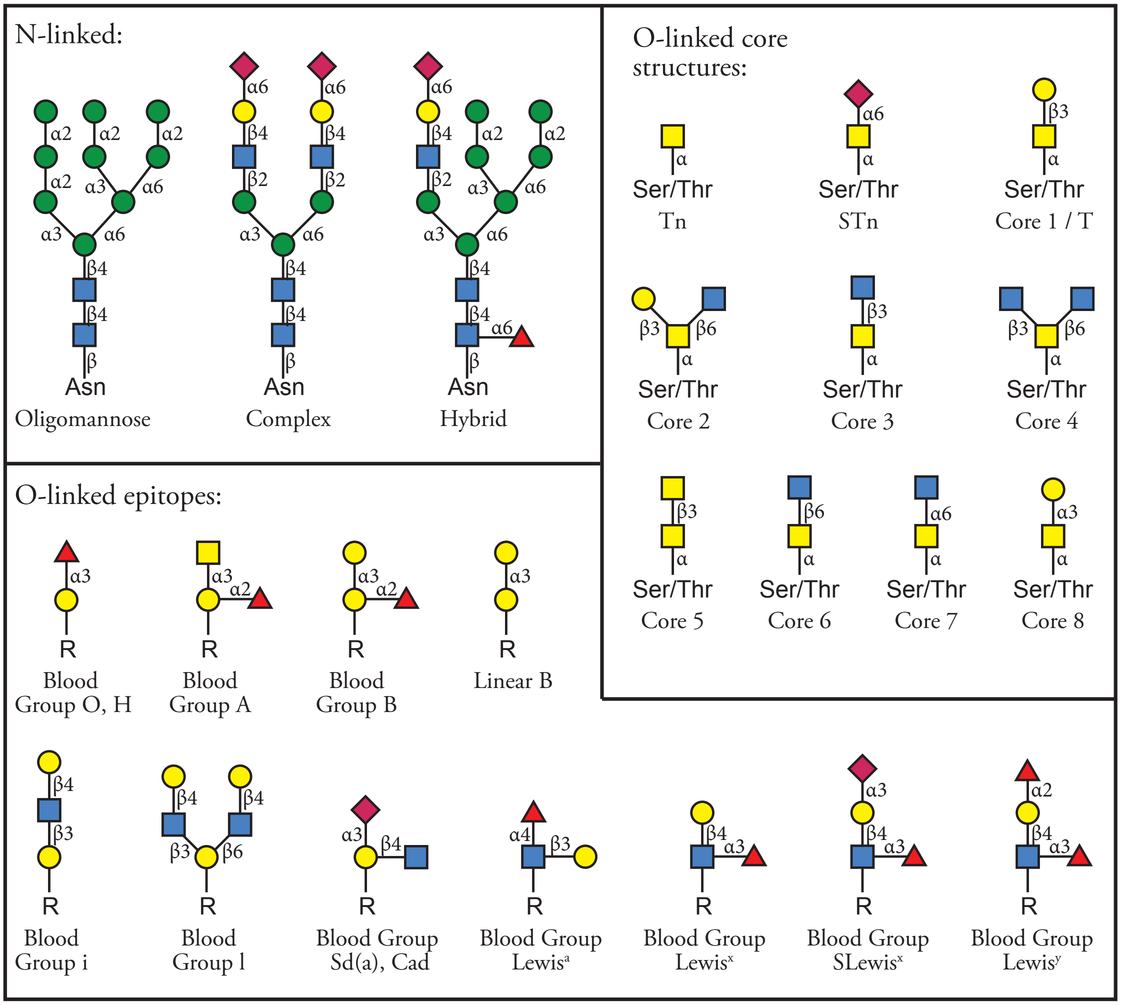


Figure 1-2: Glycans embody a highly diverse set of structures. Shown are some classes of N-linked glycans, along with O-glycan cores and epitopes found in mucins.

1.2 Mass spectrometry is a powerful tool for analyzing glycoproteins

Over the past decade, mass spectrometry has emerged as one of the most powerful tools for the analysis of large biomolecules. The core of this technique is in the ionization and analysis of biomolecular samples, allowing for the determination of mass-to-charge (m/z) ratios of the molecules in the sample[40]. Elaborations on this core technique include the addition of an on-line liquid chromatography (LC) system to increase overall resolution and sensitivity[41], and the incorporation of secondary fragmentation units—such as collision-induced dissociation (CID)[42] or electron transfer dissociation (ETD)[43]—which fragment molecules at chemically defined locations, allowing researchers to gain structural information of biomolecules of interest. Secondary fragmentation is also known as tandem fragmentation, and experiments incorporating tandem fragmentation are referred to as MS/MS, or MS^2 experiments, with ions in the tandem mass spectrum being referred to as daughter ions. Tandem fragmentation can recursively performed on daughter ions, in which case the experiment is referred to as an MS^n experiment.

Utilizing both LC and tandem fragmentation, the technique of bottom-up proteomics has had widespread success in the analysis of proteins. In this method, a sample of proteins is first subjected to proteolytic digestion (typically with the protease trypsin), after which the resulting peptides are analyzed by LC-MS. Ions in the primary mass spectrum (termed the full-scan MS) are selected, typically on the basis of intensity, for tandem fragmentation and a mass spectrum of the daughter ions is obtained. Comparison to a proteome database is then able to assign the most likely peptide sequence to each tandem MS, yielding peptide primary sequences and protein identifications[44–46]. Bottom-up proteomics has proven to be a robust platform for the analysis of protein samples[47–49] and, since its introduction, has been extended to allow relative quantification of protein levels between biological samples[50,51]. A generalized proteomics workflow is given in Figure 1-3. Bottom-up proteomics methodologies have also been extended to include the analysis of sites of protein posttranslational modifications, including phosphorylation[52–55], glycosylation[56–58], and ubiquitination[59].

Extension of this sort of analysis to heterogeneous glycan structures has also been attempted with some success. In this strategy, glycans are liberated from their parent proteins using an endoglycosidase such as PNGaseF or EndoH (N-linked glycans) or by chemical means such as base-catalyzed beta-elimination (O-linked glycans), and the freed glycan population is subsequently analyzed by LC-MS/MS[60,61] (also reviewed in North, *et al.*, 2009[62] and Harvey, 2010[63]). However, due in part to the fact that glycans with different chemical linkages can produce ions of the same molecular weight, generalization of this methodology has proven to be difficult. Furthermore, since glycans are typically released from their parent proteins prior to analysis, information about the protein-glycan relationship is lost in this analysis.

Further attempts have been made to reconcile the fields of glycomics and proteomics, with the aim to determine sites of modification and glycan structure concurrently[56]. However, in this type of experiment, the problem of protein concentration is confounded with the challenges posed glycan heterogeneity, and currently no single, widely utilized method exists for the study of glycoproteomics. Two of the major hurdles in the analysis of glycoproteins by mass spectrometry are glycan heterogene-

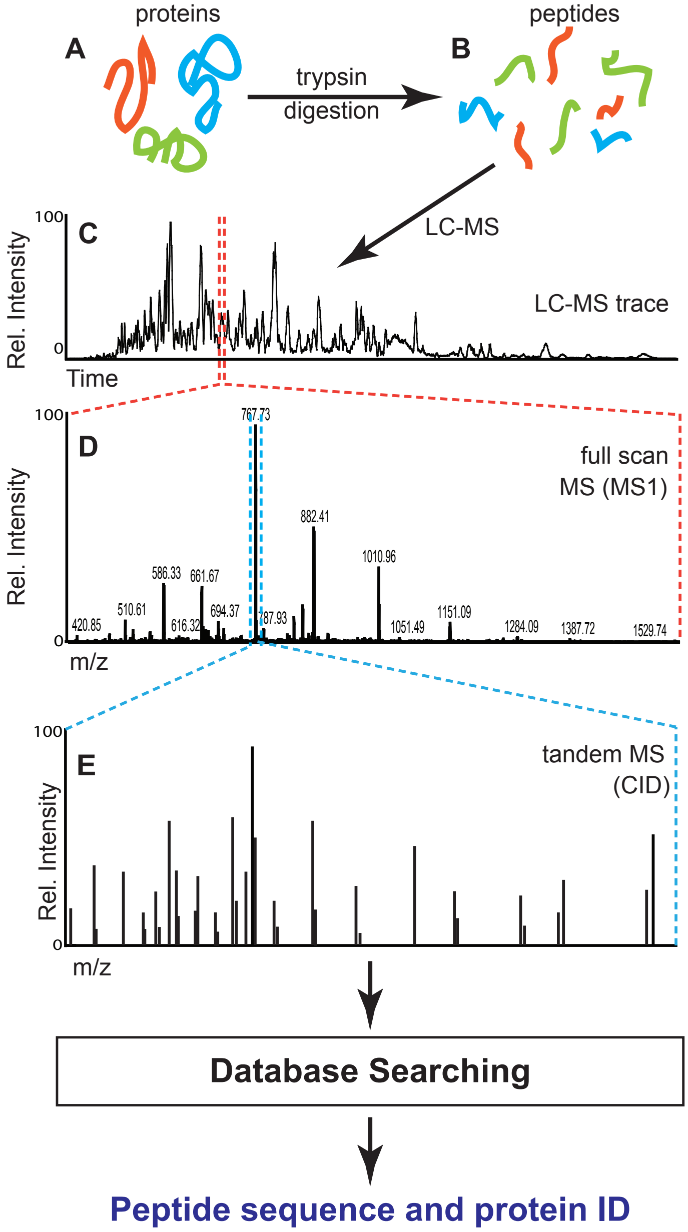


Figure 1-3: A schematic flowchart of a typical shotgun proteomics experiment. (**A**) A sample of proteins of interest is digested with a protease (typically trypsin) to (**B**) produce a solution of short peptic fragments . (**C**) These fragments are then separated by liquid chromatography , (**D**) and a mass spectrum of the eluting sample is taking at regular intervals. (**E**) From the full scan mass spectrum, ions are chosen and subjected to secondary fragmentation, and a mass spectrum of the resultion ions is obtained . Data from the second mass spectrum is analyzed by database searching to identify peptide sequences and parent protein identifications.

ity and the relative abundance of proteins of interest. Part of the challenge lies in the manner in which ions are selected for secondary fragmentation. As most biological molecules are indistinguishable by mass spectrometry without knowledge about their exact molecular weight, ions are typically selected for fragmentation based solely on ion intensity*. Since protein concentrations can span as many as eight orders of magnitude[64–66], low abundance, yet biologically interesting species—such as disease markers or transcription factors—may not be selected for fragmentation, and thus no information about these proteins will be recorded.

One widely used strategy for dealing with biological complexity is to enrich for biologically interesting species through the use of affinity purification techniques[67–70]. Affinity purification has proven to be a powerful technique to enrich for a given protein or subset of proteins. The primary hurdle in affinity purification is matching the affinity technique with the biological sample in question. Single proteins may be purified by immunoprecipitation[71] or by the creation of fusion proteins where the protein of interest is genetically modified to include an affinity tag (such as the 6xHis, myc, or FLAG tags)[72]. However, these techniques require that the identity of of the protein be known prior to the analysis, a condition often not met in proteomics experiments. Growing interest has been in developing affinity techniques that enrich for specific classes of proteins, including chromatin immunoprecipitation (ChIP) for the enrichment of chromatin-associated proteins[73] or lectin affinity chromatography for the enrichment of certain classes of glycoproteins[74,75]. A similar approach that has gained some traction in the proteomics community is to use antibody depletion column chromatography in an attempt to remove the most abundant species in a biological sample[76]. Here, the goal is simply to remove very high abundance proteins that are ubiquitous in biological samples, such as IgG and albumin from serum samples.

An alternative to sample enrichment is to direct MS/MS analysis towards the most information-rich subset of peptides, a technique termed "directed mass spectrometry"[77]. Here, the mass spectrometer is instructed to fragment ions that are the most likely to be interesting to the biological system in question. In addition to improving the biological relevance of protein identifications, directed MS also aims to improve reproducibility of results between MS experiments. This technique has also been referred to as "hypothesis-driven proteomics". The main challenge with this method is in determining at the full-scan stage which ions are most likely to be information rich.

*Often termed "data-dependent acquisition"

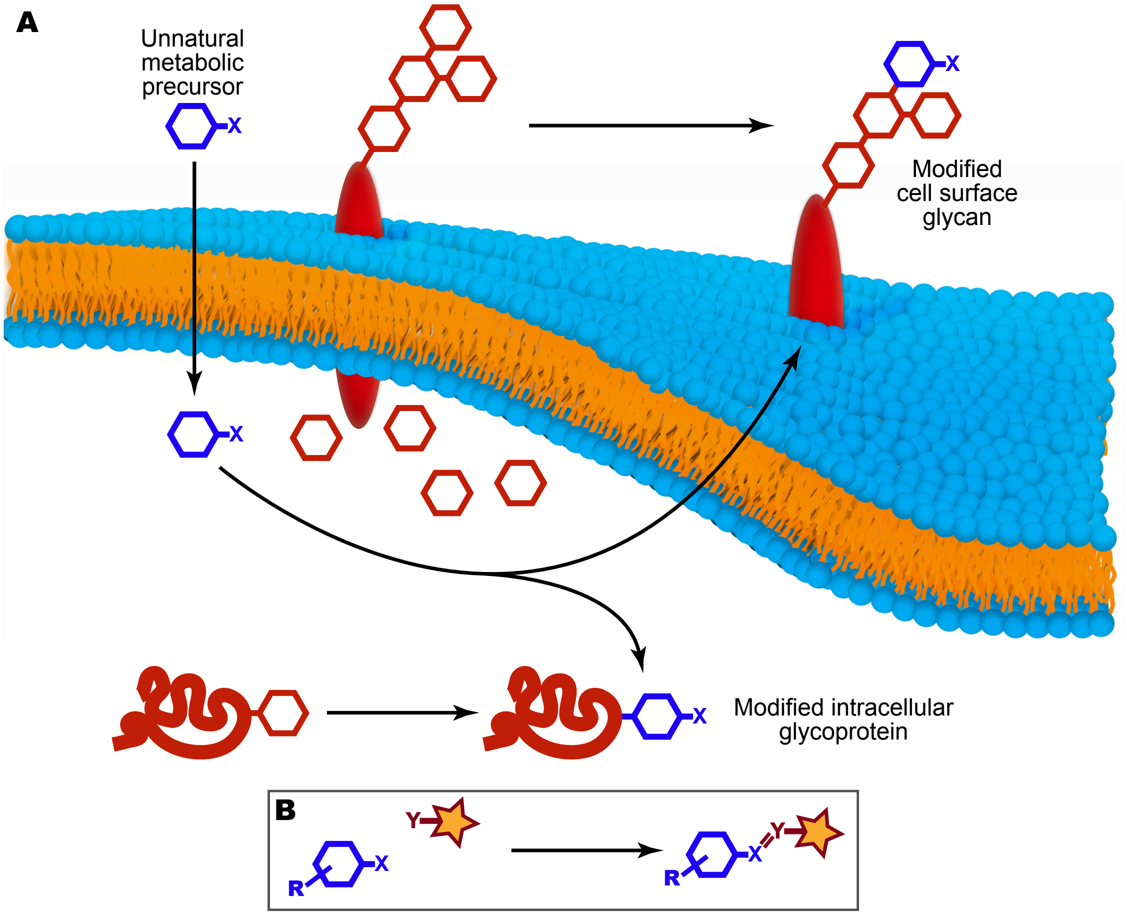

A
Unnatural metabolic precursor
X
X
Modified cell surface glycan
X
X
Modified intracellular glycoprotein
B
Y
R
X
R
X
Y

Figure 1-4: Metabolic oligosaccharide engineering. (**A**) Unnatural sugar building blocks are taken up by cells and enter the cell's biosynthetic machinery, and are ultimately incorporated into the glycoconjugates including cell-surface glycoproteins and cytosolic glycoproteins. (**B**) After incorporation into glycoconjugates of interest, the functional handle (X) can then be selectively reacted with a probe (Y) through a bioorthogonal ligation resulting in a covalent linkage between the biomolecular species of interest and a variety of probes. Figure adapted from Dube *et al.*[78]

1.3 Unnatural sugars can function as chemical handles

One approach to the analysis of glycoproteins that has gained considerable interest over the past decade is the metabolic engineering of glycoproteins through the use of unnatural monosaccharide building blocks[78]. In this strategy, cells are grown on media containing an unnatural sugar that has been functionalized with a small chemical handle. The functionalized sugar can then enter the cell, where it is processed by the cell's biosynthetic machinery and is ultimately incorporated into the cell's glycoproteins (Figure 1-4). Once incorporated, the non-biological chemical label can serve as a functional handle that can be accessed through a number of bioorthogonal reactions as discussed in below.

In contrast to other existing techniques, metabolic engineering of glycans allows for the targeting of specific glycan types based on their monosaccharide composition. For example, N-acetylgalactosamine (GalNAc) analogs are are incorporated into the core positions of mucin-type O-glycans, while analogs of N-acetylmannosamine (ManNAc) derivatives modified at the 2-position are converted by cells into sialic acid analogs, allowing for the specific targeting of sialylated glycan structures[79,80]. Some commonly used unnatural sugars and their cellular targets can be found in Table 1-1. The majority of the unnatural sugars reported in the literature are given as their peracetylated derivatives, as they have been shown to be incorporated into glycans with higher efficiency in cells lacking specific transporters for the monosaccharide of interest[81]. In addition to targeting specific monosaccharide units, the metabolic incorporation of unnatural sugars also provides a measure of the dynamic state of glycosylation as the chemical tags are only incorporated into newly synthesized glycan structures.

An alternative to metabolic incorporation of unnatural sugars was proposed by Hsieh-Wilson and coworkers whereby functionalized GalNAc derivatives are chemoenzymatically appended onto cytosolic O-GlcNAcylated proteins using a mutant beta-1,4-galactosyltransferase[89]. After labeling, the functional group can be detected using a wide variety of probes as described above. This approach has helped facilitate the analysis of the repertoire of cytosolic GlcNAcylated proteins, as was artfully demonstrated in recent work by Hart and coworkers[90]. An advantage of this strategy is that the chemoenzymatic attachment of unnatural sugars typically gives higher incorporation efficiency than the metabolic incorporation of similar sugars. However, this increase in efficiency comes at the cost of dynamic resolution, as all O-GlcNAcylated proteins will be tagged by this strategy regardless of when they were synthesized.

1.4 Using metabolic incorporation of unnatural sugars for protein cross-linking

Rather than using unnatural sugars as functional handles to facilitate the enrichment of specific subsets of the glycoproteme, work by Paulson and coworkers[87], and more recently by the Kohler lab[88], has utilized unnatural sugars to identify protein binding partners. In this strategy, cells are metabolically labeled with monosaccharides functionalized with photocrosslinking groups, such as arylazide or diazarene groups. Briefly, cells are grown in media containing the unnatural sugar containing the crosslinking group, where the sugar enters the cell and is ultimately incorporated

Table 1-1: Commonly used sugars and their glycoprotein targets.

Unnatural **Sugar**	**Glycoprotein** **Target**	**Functional** **Handle**
GlcNAz	Mucin-type O-linked[79]; cytosolic O-GlcNAc[82, a]	Azide
GalNAz	Mucin-type O-linked[79]; cytosolic O-GlcNAc[82, a]	Azide
ManNAz[80]	Sialic Acid	Azide
6-azidofucose (6AzFuc)[83]	Fucose	Azide
N-(4-pentynoyl)-glucosamine (GlcNAl)	Mucin-type O-linked[b]; cytosolic O-GlcNAc[b]; N-linked core (yeast)[84, c]	Alkyne
N-(4-pentylnoyl)-mannosamine (ManNAl)[85]	Sialic Acid	Alkyne
Alkynyl-fucose[85]	Fucose	Alkyne
N-levulinoylmannosamine (ManLev)[86]	Sialic Acid	Ketone
Crosslinking Sugars:		
9-arylazido-Neu5Ac (9-AAz-NeuAc)[87]	Sialic Acid	Aryl azide
N-(4,4'-diazobutanoyl)-mannosamine (ManNDAz)[88]	Sialic Acid	Diazarene

[a] UDP-GlcNAz and UDP-GlcNAc can be interconverted in mammalian cells through the action of a UDP-GlcNAc 4-epimerase . The choice of labeling cytosolic O-GlcNAc labeled proteins and mucin-type O-linked glycans can largely be controlled by sample preparation and choice of bioorthogonal probe.

[b] Expected

[c] Only shown in a genetically modified strain of yeast[84]

into a specific subset of glycans. After incorporation, cells are irradiated with UV light, upon which the functionalized sugars are unmasked and reactive intermediates are free to form covalent bonds with nearby biomolecules. Proteins of interest can then be purified and their interaction partners can be identified through LC-MS/MS analysis[91].

1.5 Bioorthogonal ligations used in glycoproteomics

Performing chemical reactions in living systems presents a unique challenge as living systems contain a huge range of chemical functionality, and conditions in biological systems are mild compared to those typically used in synthetic reactions. This challenge has lead to the development of "bioorthogonal chemistry" which aims to develop pairs of functional groups that can undergo a specific and rapid reaction to produce a covalent bond at physiological conditions and in the presence of the large number of potentially interfering chemical groups.[92]

The classical example of the bioorthogonal reaction is the copper-catalyzed azide-alkyne [3+2] dipolar cycloaddition (CuAAC), the so-called "copper click" reaction[93]. However, the copper click reaction has several shortcomings: the reaction is relatively slow, the Cu^I catalyst is toxic to living systems, and the reaction shows significant background in complex biological systems. In an effort to address these issues, there has been an explosion in interest in developing new bioorthogonal ligations over the past decade. The product of these efforts has been a dramatic increase in the number of bioorthogonal reactions available, many of which display improved kinetics and utilize probes with increased solubility that are better tolerated by biological systems. One of the earliest additions to this toolkit was the reaction between azides at triarylphosphine reagents through a modified Staudinger reaction, termed the Staudinger Ligation[94]. The Staudinger Ligation proceeds at physiological conditions, and is well-tolerated by live cells.

A variation on the standard click reaction is the strain-promoted [3+2] cycloaddition of cyclooctyne probes with azides, the most well known being the difluorinated cyclooctyne (DIFO) reagents in the "copper-free click" reaction[95]. Variants of the strain-promoted cycloaddition include the dibenzocyclooctynes (DIBO) reported by Boons and coworkers[96], the more soluble 6,7-dimethoxyazacyclooct-4-yne (DIMAC)[97] and, most recently, the biarylazacyclooctynones (BARAC) by reported Jewett *et al.*[98] A summary of the common bioorthogonal reactions used with azides is shown in Figure 1-5, along with structures for some recently reported azide-reactive probes.

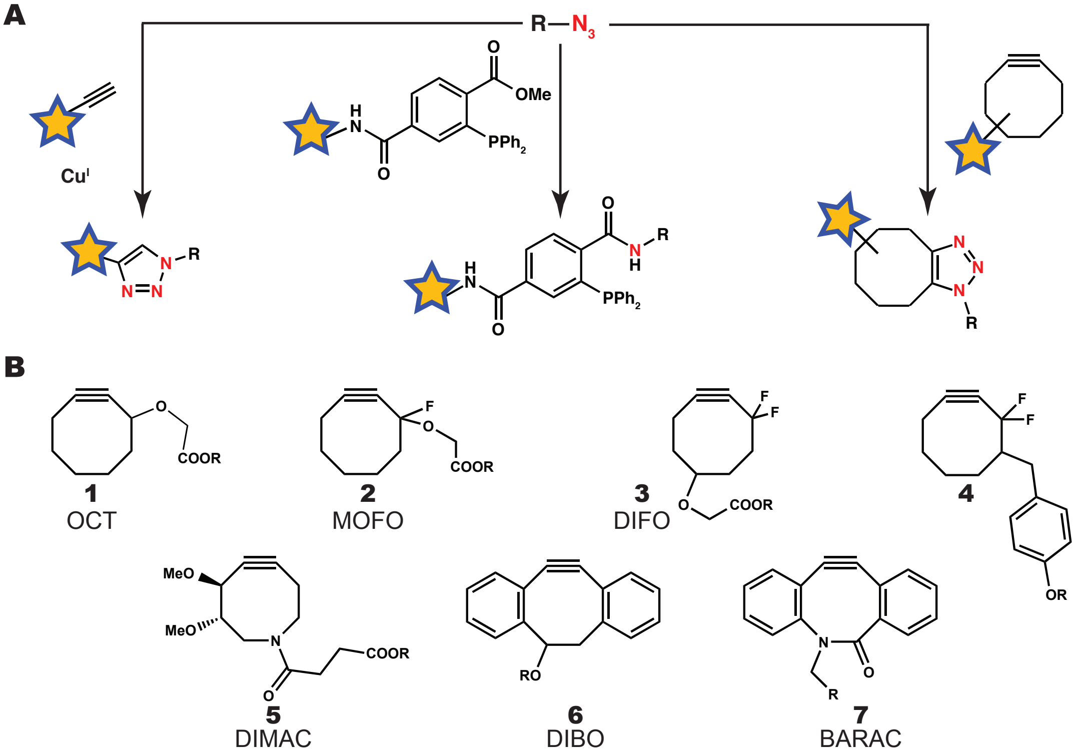


Figure 1-5: Bioorthogonal reactions with azides. (**A**) The most frequently used bioorthogonal reactions involving azides and (**B**) Structures of some recently reported cyclooctyne probes .

Aldehydes and ketones have also found utility in bioorthogonal chemistry, and a number of unnatural sugar analogs containing these functional groups have been reported. These functional groups undergo rapid reactions with hydrazides and amino-oxy reagents, but typically require higher concentration of probes and lower pH. In addition, their use is often complicated by the presence of numerous metabolites containing aldehyde or ketone functionalities. While this is by no means a comprehensive list of existing bioorthogonal reactions, these reactions constitute the vast majority of those that have to date found use in glycoproteomics. For a more comprehensive review of bioorthogonal reactions, the reader is directed to a recent review by Sletten *et al.*[99].

1.6 Facilitation of directed mass spectrometry through metabolic engineering

Many of the techniques discussed above have become widely used in the field of proteomics. Affinity purification techniques have become particularly popular among mass spectrometrists as they offer an efficient means of reducing sample complexity. However, no purification method is 100% selective or efficient. As protein concentrations can span more than eight orders of magnitude in biological systems[64], even the best purification methods can be inadequate when applied to these complex systems. Since ions are typically chosen for fragmentation based solely on signal intensity, MS analysis of peptide samples is biased towards highly abundant species. Furthermore, due to the stochastic nature in which ions are chosen for fragmentation, it can be difficult to reproduce results between MS experiments, even when analyzing identical biological samples. Directed mass spectrometry aims to alleviate these issues by focusing tandem analysis on a small subset of m/z values that have been determined to contain the most information-rich ions[77].

This thesis introduces the concept of chemically directed proteomics. This technology employs an isotopic labeling strategy, whereby the isotopic envelopes of information-rich peptides are rendered sufficiently unique such that they are computationally detectable in the full scan MS, allowing the instrument to direct the analysis to these species. Chapter 2 introduces the concept of chemically directed proteomics in more detail, as well as a labeling strategy coupled to a computational algorithm that, in combination, allow for the detection of labeled species in a full scan MS. Due to the complexity of LC-MS data obtained for biological samples, a chemical labeling strategy in isolation is of little use as manual analysis of the data can be extremely cumbersome. This is addressed by the introduction of an algorithm that can automate the detection of labeled species, introduced in Chapter 3. As the isotopic contribution from peptides themselves is non-negligible in the searching processes, their presence must be accounted for. Chapter 4 analyzes and expands upon an estimation system that has been used for this purpose, and extends its application to biomolecules in which a major fraction of their molecular weight is derived from carbohydrates.

Expanding on this isotopic labeling strategy, Chapter 5 introduces a new application of metabolic engineering using an isotopically labeled mixture of GlcNAc to perform chemically directed mass spectrometry on a sample of yeast N-linked glycoproteins. Here, UDP-GlcNAc auxotrophic yeast were grown in media containing an isotopically labeled mixture of a natural sugar, GlcNAc. After entering the cell, the isotopically labeled GlcNAc is converted into the UDP-sugar, and is subsequently incorporated into the core of all N-glycoproteins. Proteins can then be purified and treated with Endo H, cleaving off all but the most proximal GlcNAc residue containing the isotopic signature. The distinct isotopic pattern produced by the sugar can then be recognized computationally, and the mass spectrometer can be directed to fragment ions containing the incorporated pattern, in effect directing the analysis away from more abundant, but undesirable peptides. Finally, Chapter 6 examines the future of chemically directed proteomics, suggesting a number of applications of this and related technologies.

References

[1] Velculescu, V. E et al. (1997) Characterization of the yeast transcriptome. *Cell* **88**: 243–251.

[2] Hegde, P. S, White, I. R, & Debouck, C. (2003) Interplay of transcriptomics and proteomics. *Curr. Op. Biotechnol.* **14**: 647–651.

[3] Betts, J. C. (2002) Transcriptomics and proteomics: tools for the identification of novel drug targets and vaccine candidates for tuberculosis. *IUBMB* **53**: 239–242.

[4] Kahn, P. (1995) Molecular biology–from genome to proteome–looking at a cells proteins. *Science* **270**: 369–370.

[5] Pandey, A & Mann, M. (2000) Proteomics to study genes and genomes. *Nature* **405**: 837–846.

[6] Venter, J. C et al. (2001) The sequence of the human genome. *Science* **291**: 1304–1351.

[7] Jensen, P. R. (2009) Linking species concepts to natural product discovery in the post-genomic era. *J. Ind. Microbiol. Biotechnol.* **37**: 219–234.

[8] Winzeler, E. A. (2008) Maleria research in the post-genomic era. *Nature* **455**: 751–756.

[9] Thorisson, G. A, Muilu, J, & Brookes, A. J. (2009) Genotype-phenotype databases: challenges and solutions for the post-genomic era. *Nat. Rev. Genet.* **10**: 9–18.

[10] Eisenberg, D, Marcotte, E. M, Zenarios, I, & Yeates, T. O. (2000) Protein function in the post-genomic era. *Nature* **405**: 823–826.

[11] Varki, A et al. (1999) *Essentials of Glycobiology*, pp1-653. (Cold Spring Harbor Press, NY: Cold Spring Harbor).

[12] Sasai, K, Ikeda, Y, Fujii, T, Tsuda, T, & Taniguchi, N. (2002) UDP-GlcNAc concentration is an important factor in the biosynthesis of beta 1,6-branched oligosaccharides: regulation based on the kinetic properties of N-acetylglucosaminyltransferase V. *Glycobiology* **12**: 119–127.

[13] Ohtsubo, K et al. (2005) Dietary and genetic control of glucose transporter 2 glycosylation promotes insulin secretion in suppressing diabetes. *Cell* **123**: 1307–1321.

[14] Taylor, R. P et al. (2008) Glucose deprivation stimulates O-GlcNAc modification of proteins through up-regulation of O-linked N-acetylglucosaminyltransferase. *J. Biol. Chem.* **283**: 6050–6057.

[15] Lowe, J. B & Marth, J. D. (2003) A genetic approach to mammalian glycan function. *Annual Rev. Biochem.* **72**: 643–691.

[16] Dennis, J. W, Nabi, I. R, & Demetriou, M. (2009) Metabolism, cell surface organization, and disease. *Cell* **139**: 1229–1241.

[17] Prescher, J. A & Bertozzi, C. R. (2006) Chemical technologies for probing glycans. *Cell* **126**: 851.

[18] Sperandio, M, Gleissner, C. A, & Ley, K. (2009) Glycosylation in immune cell trafficking. *Immunol. Rev.* **230**: 97–113.

[19] Zhao, Y et al. (2008) Branched N-glycans regulate the biological functions of integrins and cadherins. *FEBS J.* **275**: 1939–1948.

[20] Levy, Y. (2001) Galectin-8 functions as a multicellular modulator of cell adhesion. *J. Biol. Chem.* **276**: 31285–31295.

[21] Stillman, B. N et al. (2006) Galectin-3 and galectin-1 bind distinct cell surface glycoprotein receptors to induce T cell death. *J. Immun.* **176**: 776–789.

[22] Rudd, P. M et al. (1999) Roles of glycosylation of cell surface receptors involved in immune recognition. *J. Mol. Biol.* **293**: 351–366.

[23] Yoshimura, M, Ihara, Y, Matsuzawa, Y, & Taniguchi, N. (1996) Aberrant glycosylation of E-cadherin enhances cell-cell binding to suppress metastasis. *J. Biol. Chem.* **271**: 13811–13815.

[24] Krishnan, V, Bane, S. M, Kawle, P. D, Naresh, K. N, & Kalraiya, R. D. (2005) Altered melanoma cell surface glycosylation mediates organ specific adhesion and metastasis via lectin receptors on the lung vascular endothelium. *Clin. Exp. Metastasis* **22**: 11–24.

[25] Dennis, J. W, Laferte, S, Waghorne, C, Breitman, M. L, & Kerbel, R. S. (1987) Beta-1-6 branching of Asn-linked oligosaccharides is directly associated with metastasis. *Science* **236**: 582–585.

[26] Kato, K et al. (2010) Loss of UDP-GalNAc:polypeptide N-acetylgalactosaminyltransferase 3 and reduced O-glycosylation in colon carcionoma cells selected for hepatic metastasis. *Glycoconjugate J.* **27**: 267–276.

[27] Zhang, G et al. (2010) Suppression of human prostate tumor growth by a unique prostate-specific monoclonal antibody F77 targeting a glycolipid marker. *Proc. Nat'l. Acad. Sci.* **107**: 732–737.

[28] Cheung, P et al. (2007) Matabolic homeostasis and tissue renewal and dependent on beta-1,6-GlcNAc-branched N-glycans. *Glycobiology* **17**: 828–837.

[29] Jethmalani, S. M & Henle, J. K. (1994) Prompt glycosylation of calreticulin is independent of ca^{2+} homeostasis. *Biochem. Biophys. Res. Commun.* **205**: 780–787.

[30] Li, J. J, Dickson, D, Hof, P. R, & Vlassara, H. (1998) Receptors for advanced glycosylation endproducts in human brain: role in brain homeostasis. *Mol. Medicine* **4**: 46–60.

[31] Roth, J. (2002) Protein N-glycosylation along the secretory pathway: relationship to organelle topography and function, protein quality control, and cell interactions. *Chem. Rev.* **102**: 285–303.

[32] Helenius, A & Aebi, M. (2004) Roles of N-linked glycans in the endoplasmic reticulum. *Annual Rev. Biochem.* **73**: 1019–1049.

[33] Dentin, R, Hedrick, S, Xie, J, Yates III, J, & Montminy, M. (2008) Hepatic glucose sensing via the CREB coactivator CRTC2. *Science* **319**: 1402–1405.

[34] Yang, X et al. (2008) Phosphoinositide signalling links *O*-GlcNAc transferase to insulin resistance. *Nature* **451**: 961–967.

[35] Goldberg, H. J, Whiteside, C. I, Hart, G. W, & Fantus, I. G. (2006) Posttranslational, reversible *o*-glycosylation is stimulated by high glucose and mediates plasminogen activator inhibitor-1 gene expression and Sp1 transcriptional activity in glomerular mesangial cells. *Endocrinology* **147**: 222–231.

[36] Cooksey, R. C, Pusuluri, S, Hazel, M, & McClain, D. A. (2005) Hexosamines regulate sensitivity of glucose-stimulated insulin

secretion in β-cells. *Am. J. Physiol. Endocrinol. Metab.* **290**: E334–E340.

[37] Poirier, F & Kimber, S. (1997) Cell surface carbohydrates and lectins in early development. *Mol. Hum. Reprod.* **3**: 907–918.

[38] Lau, K. S. (2007) Complex N-glycan number and degree of branching cooperate to regulate cell proliferation and differentiation. *Cell* **129**: 123–134.

[39] Apweiler, R, Hermjakob, H, & Sharon, N. (1999) On the frequencey of protein glycosylation, as deduced from analysis of the SWISS-PROT database. *Biochim. Biophys. Acta-Gen. Subj.* **1473**: 4–8.

[40] Sparkman, O. D. (2000) *Mass spectrometry desk reference*. (Global View Pub.).

[41] Tomer, K. B & Parker, C. E. (1989) Biochemical applications of liquid chromatography-mass spectrometry. *J. Chromatography* **492**: 189–221.

[42] Wells, J. M & McLuckey, S. A. (2005) Collision-induced dissociation (CID) of peptides and proteins. *Methods Enzymol.* **402**: 148–185.

[43] Syka, J. E. P, Coon, J. J, Schroeder, M. J, Shabanowitz, J, & Hunt, D. F. (2004) Peptide and protein sequence analysis by electron transfer dissociation mass spectrometry. *Proc. Natl. Acad. Sci.* **101**: 9528–9533.

[44] Mann, M, Hojrup, P, & Roepstorff, P. (1993) Use of mass-spectrometric molecular-weight information to identify proteins in sequence databases. *Biol. Mass. Spectrom.* **22**: 338–345.

[45] Yates III, J. R, Eng, J. K, Caluser, K. R, & Berlingame, A. L. (1996) Search of sequence databases with uninterpreted high-energy collision induced dissociation spectra of peptides. *J. Am. Soc. Mass. Spectrom.* **7**: 1089–1098.

[46] Craig, R, Cortens, J. P, & Beavis, R. C. (2004) Open source system for analyzing, validating, and storing protein identification data. *J. Prot. Res.* **3**: 1234–1242.

[47] Domon, B & Aebersold, R. (2006) Mass spectrometry and protein analysis. *Science* **312**: 212–217.

[48] Aebersold, R & Goodlett, D. R. (2001) Mass spectrometry in proteomics. *Chem. Rev.* **101**: 269–295.

[49] Aebersold, R & Mann, M. (2003) Mass spectrometry-based proteomics. *Nature* **422**: 198–207.

[50] Gygi, S. P et al. (1999) Quantitative analysis of complex protein mixtures using isotope-coded affinity tags. *Nature Biotechnology* **17**: 994–999.

[51] Ross, P. L et al. (2004) Multiplexed protein quantitation in *saccharomyces cerevisiae* using amine-reactive isobaric tagging reagents. *Mol. Cell. Proteomics* **3**: 1154–1169.

[52] Blethrow, J. D, Glavy, J. S, Morgan, D. O, & Shokat, K. M. (2008) Covalent capture of kinase-specific phosphopeptides reveals Cdk-1-cyclin B substrates. *Proc. Natl. Acad. Sci.* **105**: 1442–1447.

[53] Olsen, J. V et al. (2006) Global, in vivo, and site-specific phosphorylation dynamics in signalling networks. *Cell* **127**: 635–648.

[54] Salih, E. (2005) Phosphoproteomics by mass spectrometry and classical protein chemistry approaches. *Mass. Spectrom. Rev.* **24**: 828–846.

[55] Yu, L, Issaq, H. J, & Veenstra, T. D. (2007) Phosphoproteomics for the discovery of kinases as cancer biomarkers and drug targets. *Proteomics Clin. Appl.* **1**: 1042–1057.

[56] Harvey, D. J. (2001) Identification of protein-bound carbohydrates by mass spectrometry. *Proteomics* **1**: 311–328.

[57] Wells, L et al. (2002) Mapping sites of O-GlcNAc modification using affinity tags for serine and threonine post-translational modifications. *Mol. Cell. Proteom.* **1**: 791–804.

[58] Khidekel, N et al. (2007) Probing the dynamics of O-GlcNAc glycosylation in the brain using quantitative proteomics. *Nat. Chem. Biol.* **3**: 339–348.

[59] Peng, J. (2008) Evaluation of proteomic strategies for analyzing ubinquitinated proteins. *BMB Rep.* **41**: 177–183.

[60] Yoshinao et al. (2007) Comparison of the methods for profiling glycoprotein glycans—HUPO human disease glycomics/proteome initiative multi-institutional study. *Glycobiology* **17**: 411–422.

[61] Harvey, D. J, Bateman, R. H, Bordoli, R. S, & Tyldesley, R. (2000) Ionisation and fragmentation of complex glycans with a quadrupole time-of-flight mass spectrometer fitted with a matrix-assisted laser desorption/ionisation ion source. *Rapid Commun. Mass Spectrom.* **14**: 2135–2142.

[62] North, S. J, Hitchen, P. G, Haslam, S. M, & Dell, A. (2009) Mass spectrometry in the analysis of N-linked and O-linked glycans. *Curr. Opin. Struct. Biol.* **19**: 498–506.

[63] Orlando, R. (2010) Quantitative glycomics. *Methods Mol. Biol.* **600**: 31–49.

[64] Anderson, N. L et al. (2004) The human plasma proteome: a nonredundant list developed by combination of four separate sources. *Mol. Cel. Proteomics* **3**: 311–326.

[65] Anderson, N. L & Anderson, N. G. (2002) The human plasma proteome: history, character, and diagnostic prospects. *Mol. Cell. Proteomics* **1**: 845–867.

[66] Ghaemmaghami, S et al. (2003) Global analysis of protein expression in yeast. *Nature* **425**: 737–741.

[67] Block, H et al. (2009) Immobilized-metal affinity chromatography (IMAC): a review. *Methods Enzymol.* **463**: 439–473.

[68] Urh, M, Simpson, D, & Zhao, K. (2009) Affinity chromatography: general methods. *Methods Enzymol.* **463**: 417–438.

[69] Jiang, D, Jarrett, H. W, & Haskins, W. E. (2009) Methods for proteomic analysis of transcription factors. *J. Chrom. A* **1216**: 6881–6889.

[70] Pernemalm, M, Lewensohn, R, & Lehtiö, J. (2009) Affinity prefractionation for MS-based plasma proteomics. *Proteomics* **9**: 1420–1427.

[71] Kaboord, B & Perr, M. (2008) Isolation of proteins and protein complexes by immunoprecipitation. *Methods Mol. Biol.* **424**: 349–364.

[72] Terpe, K. (2003) Overview of tag protein fusions: from molecular and biochemical fundamentals to commercial systems. *Appl. Microbiol. Biotechnol.* **60**: 523–533.

[73] Collas, P. (2010) The current state of chromatin immunoprecipitation. *Mol. Biotechnol.* **45**: 87–100.

[74] Fang, X & Zhang, W. W. (2008) Affinity separation and enrichment methods in proteomic analysis. *J. Proteomics* **71**: 284–303.

[75] Mechref, Y, Madera, M, & Novotny, M. V. (2008) Glycoprotein enrichment through lectin affinity techniques. *Methods Mol. Biol.* **424**: 373–396.

[76] Rengaranjan, K, de Smet, M. D, & Wiggert, B. (1996) Removal of albumin from multiple human serum samples. *Biotechniques* **20**: 30–32.

[77] Schmidt, A, Claassen, M, & Aebersold, R. (2009) Directed mass spectrometry: towards hypothesis-driven proteomics. *Cur. Op. Chem. Biol.* **13**: 1–8.

[78] Dube, D. H & Bertozzi, C. R. (2003) Metabolic oligosaccharide engineering as a tool for glycobiology. *Curr. Op. Chem. Biol.* **7**: 616–625.

[79] Dube, D. H, Prescher, J. A, Quang, C. N, & Bertozzi, C. R. (2006) Probing mucin-type O-linked glycosylation in living animals. *Proc. Natl. Acad. Sci.* **103**: 4819.

[80] Luchansky, S. J, Yarema, K. J, Takahashi, S, & Bertozzi, C. R. (2003) GlcNAc 2-epimerase can serve a catabolic role in sialic acid metabolism. *J. Biol. Chem.* **278**: 8035–8042.

[81] Collins, B. E, Fralich, T. J, Itonori, S, Ichikawa, Y, & Schnaar, R. L. (2000) Conversion of cellular sialic acid expression from N-acetyl to N-glycolylneuraminic acid using a synthetic precursor, N-glycolylmannosamine pentaacetate: inhibition of myelin-associated glycoprotein binding to neural cells. *Glycobiology* **10**: 11–20.

[82] Vocadlo, D. J, Hang, H. C, Kim, E.-J, Hanover, J. A, & Bertozzi, C. R. (2003) A chemical approach for identifying O-GlcNAc-modified proteins in cells. *Proc. Natl. Acad. Sci.* **100**: 9116–9121.

[83] Rabuka, D, Hubbard, S. C, Laughlin, S. T, Argade, S. P, & Bertozzi, C. R. (2006) A chemical reporter strategy to probe glycoprotein fucosylation. *J. Am. Chem. Soc.* **128**: 12078–12079.

[84] Breidenbach, M. A et al. (2010) Targeted metabolic labeling of yeast N-glycans with unnatural sugars. *Proc. Natl. Acad. Sci.* **107**: 3988–3993.

[85] Hsu, T.-L et al. (2007) Alkynyl sugar analogs for the labeling and visualization of glycoconjugates in cells. *Proc. Natl. Acad. Sci.* **104**: 2614–2619.

[86] Yarema, K. J, Mahal, L. K, Bruehl, R. E, Rodriguez, E. C, & Bertozzi, C. R. (1998) Metabolic delivery of ketone groups to sialic acid residues. *J. Biol. Chem.* **273**: 31168–31179.

[87] Han, S, Collins, B. E, Bengtson, P, & Paulson, J. C. (2005) Homomultimeric complexes of CD22 in B cells revealed by protein-glycan cross-linking. *Nat. Chem. Biol.* **1**: 93–97.

[88] Bond, M. R, Zhang, H, Vu, P. D, & Kohler, J. J. (2009) Photocrosslinking of glycoconjugates using metabolically incorporated diazirine-containing sugars. *Nat. Protocols* **4**: 1044–1063.

[89] Khidekel, N et al. (2003) A chemoenzymatic approach toward the rapid and sensitive detection of *o*-glcnac posttranslational modifications. *J. Am. Chem. Soc.* **125**: 16162–16163.

[90] Wang, Z et al. (2010) Extensive crosstalk between O-GlcNAcylation and phosphorylation regulates cytokinesis. *Science Signaling* **3**: ra2.

[91] Gingras, A. C, Gstaiger, M, Raught, B, & Aebersold, R. (2007) Analysis of protein complexes using mass spectrometry. *Net. Rev. Mol. Cell. Biol.* **8**: 645–654.

[92] Prescher, J. A & Bertozzi, C. R. (2005) Chemistry in living systems. *Nat. Chem. Biol.* **1**: 13–21.

[93] Kolb, H. C, Finn, M. G, & Sharpless, K. B. (2001) Click chemistry: diverse chemical function from a few good reactions. *Agnew. Chem. Int. Ed. Eng.* **40**: 2004–2021.

[94] Saxon, E & Bertozzi, C. R. (2000) Cell surface engineering by a modified staudinger reaction. *Science* **287**: 2007–2010.

[95] Baskin et al. (2007) Copper-free click chemistry for dynamic *in vivo* imaging. *Proc. Natl. Acad. Sci.* **104**: 16793–16797.

[96] Ning, X, Guo, J, Wolfert, M. A, & Boons, G. J. (2008) Visualizing metabolically labeled glycoconjugates of living cells by copper-free and fast guisgen cycloadditions. *Angnew. Chem. Int. Ed. Engl.* **47**: 2253–2255.

[97] Sletten, E. M & Bertozzi, C. R. (2008) A hydrophilic azacyclooctyne for Cu-free click chemistry. *Org. Lett.* **10**: 3097–3099.

[98] Jewett, J. C, Sletten, E. M, & Bertozzi, C. R. (2010) Rapid Cu-free click chemistry with readily synthesized biarylazacyclooctynones. *J. Am. Chem. Soc.* **132**: 3688.

[99] Sletten, E. M & Bertozzi, C. R. (2009) Bioorthogonal chemistry: fishing for selectivity in a sea of functionality. *Angnew. Chem. Int. Ed.* **48**: 6974–6998.

chapter two

Isotopic signature transfer and mass pattern prediction (IsoStamp) as a new tool for chemically-directed proteomics

joint work with Brian P. Smart and Krishnan K. Palaniappan

2.1 Introduction

Common goals of MS-based proteomics experiments are to identify, characterize, and quantify proteins and their posttranslational modifications from cells or tissues[1–6]. A popular strategy for protein identification is the bottom-up shotgun proteomics approach[7]. In this method, a mixture of proteins from a biological sample is subjected to proteolytic digestion, the resulting peptides are separated by liquid chromatography (LC), and their parent proteins are identified through the detection of peptides by MS[8,9]. Two approaches can be taken to convert MS data acquired from proteolytic digests into protein identifications. The first method is based on a single-stage MS measurement with high mass accuracy. Peptide ions are rendered sufficiently unique such that their parent proteins can be identified by comparison to an *in silico* proteolytic digest of the organism's proteome[10,11]. The challenge in this approach lies in identifying an adequate number of peptide ions with sufficient mass accuracy to have confidence in a protein assignment. The second method utilizes tandem MS to obtain sequence information for individual peptides, followed by comparison against proteome databases[12,13]. Typically, only the most abundant peptides are selected for fragmentation, while data for those peptides in relatively low quantities are not obtained[3].

Inherent to the shotgun proteomics approach is the problem of identifying proteins of low abundance, such as biomarkers for disease states, against a background of proteins whose concentrations can span up to 12 orders of magnitude[1,3,14]. The

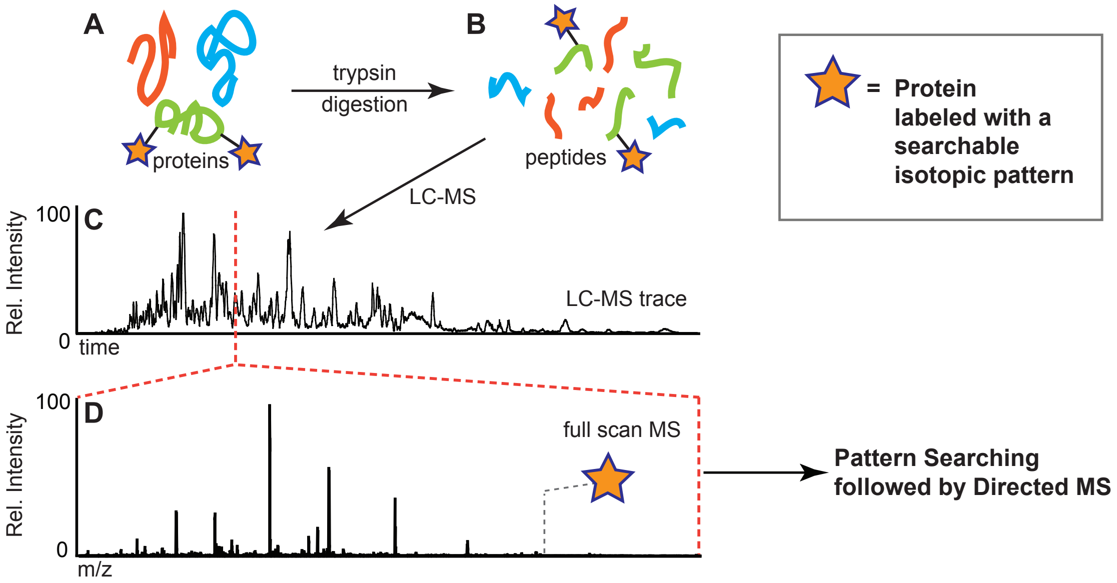


Figure 2-1: The dibromide tag improves the traditional shotgun proteomics technique by allowing chemically interesting species to be detectable in the full scan MS. **(A)** A mixture of proteins in which a small number are chemically tagged (star) is subjected to proteolytic digestion producing **(B)** a mixture of peptides. **(C)** The peptides are then separated using liquid chromatography and **(D)** full scan mass spectra are collected at regular intervals, from which tagged species identified by computational pattern searching. The tagged species can then be selected for further analysis.

molecular complexity of cell and tissue lysates renders biologically interesting peptides difficult to distinguish from a vast population of more abundant, though often uninteresting, peptides. Directed proteomics strategies seek to address the sample complexity problem by focusing the analysis on a specific protein subset[15]. In one approach, proteins of interest are selectively enriched prior to proteolytic digestion, thereby forgoing the shotgun method altogether[16–19]. Alternatively, there is growing interest in the use of chemical tags that perturb the mass envelope of target peptides so as to render them more detectable. The progenitors of this approach are the isotope-coded affinity tag (ICAT) and isobaric tags for relative and absolute quantitation (iTRAQ) techniques now commonly used for quantitative comparative proteomics[20–23]. These methods capitalize on isotopic labeling to distinguish peptides from different samples that were combined prior to MS analysis. Chemical tags have been elegantly employed to mark sites of protein posttranslational modifications[24] including glycosylation[25], lipidation[26,27] and phosphorylation[28], as well as for labeling protein N-termini[29], sites of cysteine oxidation[30] and active sites of enzymes[31].

The halogens bromine and chlorine can be advantageous components of chemical tags by virtue of their unique isotopic distributions[32–34]. Unlike the common biological elements, which exist as one predominant isotope, bromine and chlorine have two significant isotopes that create unique signatures in a mass spectrum: ^{79}Br and ^{81}Br are found in a 1:1 ratio, and ^{35}Cl and ^{37}Cl are found in a 3:1 ratio (isotopic ratios of other biologically relevant elements are given in Table 2-1)[35]. Recently, these features have been exploited in proteomics-related applications. Aebersold and coworkers used a dichloride tag to discriminate between peptides with and without cysteine from digested protein samples[36]. Likewise, N-terminal labeling of peptides with a monobromide tag has facilitated sequence identification by tandem MS[37]. In addition to their distinctive isotopic signatures, bromine and chlorine have a negative mass defect that can endow a modified peptide with a unique fractional mass[38–40]. Amster and coworkers made artful use of this property to facilitate the detection and identification of peptides from a small archaeal proteome[39]. Specifically, brominated tags were employed to select cysteine-alkylated species for peptide mass fingerprinting analysis[38,39]. Despite these achievements, halogen profiling methods have not been extended to directed proteomic analysis of samples as complex as human cell or tissue lysates. To achieve this goal would require the ability to discriminate the tag's signature on peptides over a wide mass range, in multiple charge states, and against a background of $> 100,000$ peptides[41,42], capabilities that present methods lack.

Here we report that a dibromide tag in concert with a novel computational pattern searching algorithm enables detection of labeled peptides from complex biological samples with unprecedented sensitivity and fidelity. The overall approach, which we term isotopic signature transfer and mass pattern prediction (abbreviated IsoStamp), was employed as illustrated in Figure 2-1. Cell lysates containing chemically-tagged proteins were digested with trypsin and the resulting peptides were analyzed by full scan LC-MS. Tagged peptides were detected using the pattern searching algorithm and verified by comparison to LC-MS data of a pure sample. In model shotgun proteomics experiments, we were able to discern femtomole (fmole, 10^{-15}M) quantities of labeled peptides from whole cell lysate digests at signal-to-noise ratios as low as 2.5:1. IsoStamp can enhance any proteomics platform that employs chemical labeling for targeted protein identification. By rendering labeled peptides detectable in a full-scan mass spectrum, IsoStamp is an enabling tool for "chemically directed proteomics."

Table 2-1: The natural abundance of stable isotopes of elements contained in the standard amino acids. The natural isotopic distribution of bromine (and to a lesser extent, chlorine) is unique compared to these elements as its heavy isotope is found naturally in significant quantities.

Element	Isotopes[a]	Natural Abundance
Hydrogen	^{1}H ^{2}H	99.9885% 0.0115%
Carbon	^{12}C ^{13}C	98.93% 1.07%
Nitrogen	^{14}N ^{15}N	99.636% 0.364%
Oxygen	^{16}O ^{17}O ^{18}O	99.757% 0.038% 0.205%
Phosphorus	^{31}P	>99.99%
Sulfur[b]	^{32}S ^{34}S	94.99% 4.25%
Chlorine	^{35}Cl ^{37}Cl	75.76% 24.24%
Bromine	^{79}Br ^{81}Br	50.69% 49.31%

[a] Stable isotopes
[b] ^{33}S is an insignificant contributor

2.2 Results

2.2.1 Bromine and chlorine atoms impart unique isotopic signatures on labeled molecules

The natural abundances of the isotopes of bromine and chlorine, elements not commonly found in proteins, impart distinct isotopic signatures on small molecules (i.e., MW < 500). Compounds bearing a single bromine or chlorine atom appear in the mass spectrum as two major ions, M and $M + 2$, with equal or skewed peak heights, respectively. Compounds with two bromine or chlorine atoms appear as symmetrical or skewed triplets, respectively, with major peaks at M, $M + 2$ and $M + 4$. These unique isotopic patterns are evident in the mass spectra (Figure 2-2B) for the halogated tyrosine analogs **1-4** shown in Figure 2-2A, which we synthesized as iodoacetamide derivatives capable of alkylating cysteine residues (details of the syntheses are provided in at the end of the chapter, Figure 2-6).

In principle, the uniqueness of the triplet patterns associated with the dibro-

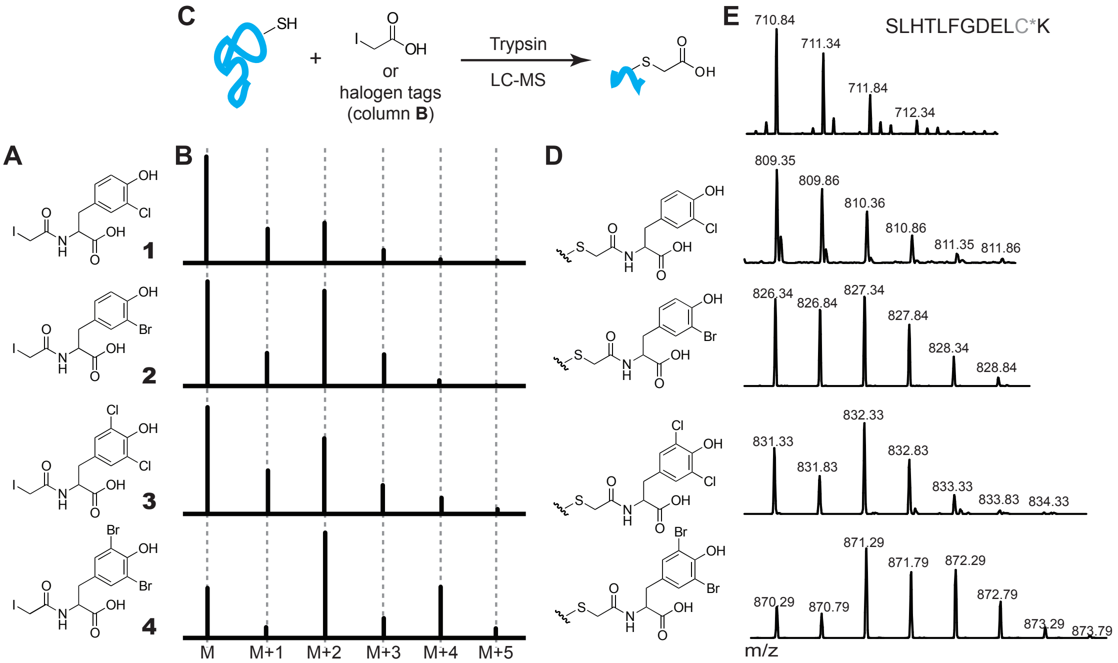


Figure 2-2: Halogenated tags impart distinct isotopic patterns on peptides. (**A**) Iodoacetamide-derivatized halogen tags were synthesized from tyrosine, resulting in chemical tags (**B**) bearing unique isotopic patterns. (**C**) In a model experiment, BSA was alkylated on cysteine residues with each tag or iodoacetic acid and then digested with trypsin. (**D**) Mass spectra of the modified tryptic peptide corresponding to residues 89-100. C* refers to a cysteine residue that was alkylated with each tag shown in **A**. (**E**) The natural abundances of isotopes of common biological elements introduce a skewing effect on the halogen signature that must be compensated for computationally.

mide and dichloride motifs could facilitate the identification of tagged peptides from complex proteolytic digests. However, the isotopic patterns are obscured in larger molecules (i.e., MW > 1000) wherein heavy isotopes of C (^{13}C, 1%), H (^{2}H, 0.02%) and N (^{15}N, 0.1%) skew the triplet motif. To illustrate the point, we alkylated the surface exposed cysteine residues of bovine serum albumin (BSA) with the tags shown in Figure 2-2A, digested the modified protein with trypsin, and analyzed the modified peptides (Figure 2-2C) by MS. Representative data corresponding to the tryptic peptide SLHTLFGDELC*K (residues 89-100, where C* denotes a tag-conjugated cysteine residue) are shown in Figure 2-2D. The isotope envelope of each tagged peptide reflects the parent peptide's intrinsic isotopic distribution, as reflected in the MS of the iodoacetic acid-alkylated version (Figure 2-2E), modified by the halogen pattern. We noted from these data that the dibromide tag imparted a more distinctive signature on the peptide's mass envelope than the other halogen tags. Computational simulations suggested a similar advantage of the dibromide tag for peptides of molecular weight up to at least 5000 Da (see Chapter 3, Figure 3-1). Still, the complexity of the tagged peptide's mass spectrum suggested that dibromide-labeled species above a certain molecular weight would not be readily discerned from complex mixtures by manual searching. Thus, we sought to develop a computer algorithm that could extract such patterns from complex MS data files.

2.2.2 Development of a pattern searching algorithm

Overview

We developed an algorithm that analyzes peaks from the full scan mass spectrum and matches the real data with simulated data generated by convoluting each predicted peptide's isotopic envelope with the pattern produced by a given tag. The algorithm receives four inputs from the user: (1) the raw mzXML data file, (2) the elemental composition of the tag, (3) a weighting factor used to tune the selectivity of the algorithm, and (4) the charge states to be considered in the search. The output comprises the mass and charge states of tagged species as well as elution times of those peptides. This information can be assembled to form an inclusion list for further tandem MS sequencing.

The algorithm comprises two major steps. First, the full scan data are analyzed to identify putative isotopic signature matches to a given elemental composition. Key to this step is a data-dependent approximation of the contributions of non-halogens to the observed isotopic envelope. As well, the algorithm allows for the inevitable imperfections in MS data derived from complex protein samples. Second, the putative matches identified in the first step are analyzed using a graph theoretic construct to reduce false positives. In this step, peaks contributing to a putative pattern match are tracked as a function of LC elution time or with respect to multiple charge states to add confidence that they derive from a single species. Central features of the algorithm are described below. Full details can be found in Chapter 2, and the source code is available in Appendix B.

Identifying putative pattern matches

The algorithm takes a list of peaks from the full scan mass spectrum and first divides them into sets that are possibly isotopically related, i.e., the peaks are separated by $1/c$ m/z units where c is the peptide's charge state. After this initial simplification, each of these sets is searched for the presence of a desired isotopic pattern as follows. First, each peak in the chosen data set is presumed to represent a peptide. Knowing the charge state and m/z for that hypothetical peptide, the program predicts its actual mass and estimates its elemental composition using the averagine system (Chapter

4)[43]. We confirmed the accuracy of the averagine method by comparing the predicted elemental compositions of 20,000 tryptic peptides from the human proteome with their actual elemental compositions, which revealed a mean deviation of less than 4%.

From the estimated elemental composition, an isotopic pattern of the unlabeled hypothetical peptide is predicted. Then the isotopic pattern of the chemical tag (i.e., dichloride tag **3** or dibromide tag **4**) is convoluted with the predicted peptide isotopic envelope, generating a reference pattern that is compared with the actual data set to determine a fitness score. The program also samples reference patterns that model untagged peptides, doubly tagged peptides, and instrument noise. Additional reference patterns can be incorporated to account for common sources of false positives in a sample-dependent manner.

Each reference pattern ($\vec{a}$) is scaled in the intensity dimension to produce an optimal alignment with the data ($\vec{d}$). This is accomplished by determining the scaling factor z by a binary search such that the sum of the squared difference (SSD) between each peak in the reference pattern ($a_i \in \vec{a}$) and the its counterpart in the actual data set ($d_i \in \vec{d}$) is minimized:

$$\text{SSD} = \sum_i (d_i - za_i)^2 \tag{2.1}$$

After intensity alignment, the score for the entire pattern is then calculated as

$$\text{Score} = \prod_i f\left(\frac{|d_i - za_i|}{\sigma\sqrt{2}}\right) \tag{2.2}$$

where σ is a measure of peak intensity variance and f is a scoring function for each peak that produces a value in the range $[10^{-\epsilon}, 1]$ given by

$$f(x) = (1 - 10^{-\epsilon})\text{erfc}(x) + 10^{-\epsilon} \tag{2.3}$$

in which erfc(x) is the complement of the Gaussian error function and the parameter ϵ is a measure of the "tightness" of the peak matching in the intensity dimension. The lower bound of ϵ is imposed on the function to reduce round-off errors in floating point arithmetic and to allow for robustness against contaminating peaks when used in a Bayesian system. In short, this system allows the identification of isotopic envelopes in actual MS data that do not perfectly match idealized mass pattern signatures by virtue of overlapping peaks from other molecular species.

Finally, after scores of all patterns of interest have been determined, the best match can be found using a Bayesian approach:

$$P(\vec{a} \mid \vec{d}\,) = \frac{P(\vec{d} \mid \vec{a_i})P(\vec{a_i}\ )}{\sum\limits_k P(\vec{d} \mid \vec{a_k}\ )P(\vec{a_k})} \tag{2.4}$$

$$= \frac{\text{Score}(\vec{a_i},\ \vec{d})P(\vec{a_i})}{\sum\limits_k \text{Score}(\vec{a_k},\ \vec{d})P(\vec{a_k})} \tag{2.5}$$

where the $P(pattern_i)$ terms are user-defined weighting factors that describe the probability that any peak in the dataset is caused by a molecular species with the isotopic distribution described by patterni, and are determined experimentally. These weighting factors allow one to increase the specificity of the program for a selected pattern, thereby eliminating common false positives, or conversely, to increase the number of hits, though potentially at the cost of more false positives. It is important to note that while the value $P(\vec{a}\ |\vec{d}\,)$ will always be within the range $[0, 1]$, it is not strictly a probability as we have not rigorously defined the value for σ nor are we able to account for every possible source of the data. Fortunately, this is not a prerequisite for pattern matching and this algorithm is able discriminate strongly between different isotopic patterns while still allowing for imperfect data, including data with obscured peaks.

Reducing false positives with a graph theoretic approach

Naive pattern matching, as described above, may produce significant numbers of false positive matches depending on the complexity of the data. To reduce the false positive rates, our algorithm exploits two features of LC-MS data: 1) peptides are often detected in multiple charge states, and 2) the same peptide is likely to be detected in several adjacent scans due to the nature of LC elution profiles. To take advantage of these features, a graph theoretic approach employed used wherein each potential match is treated as a node in a graph. Edges are then drawn between two nodes if 1) the nodes could have come from the same molecular species, and 2) the nodes have sufficiently similar LC elution times. After edges are built, the graph can be decomposed into disjoint subsets, in which all nodes in a given subset could have been produced by the same molecular species and each node eluted close in time to another node in that subset. Each of these subsets is then scored on a number of factors, including the number of nodes in the set and the number of unique charge states detected. Because matches that were made by chance are unlikely to score highly on these criteria, this process acts as an efficient filter to remove false positive matches. A detailed description of the searching algorithm along with an analysis of its performance is given in Chapter 3 and computer source code can be found in Appendix B.

2.2.3 Application of IsoStamp in a model shotgun proteomics experiment

As mentioned previously, the extreme complexity of unfractionated cell or tissue lysates renders the identification of low abundance proteins by shotgun proteomics a challenging endeavor. We therefore sought to test the sensitivity of IsoStamp in identifying labeled proteins from whole cell lysates. BSA was chosen as a model protein for labeling because it contains 35 cysteine residues that are spread throughout the entire protein, and produces 80 tryptic peptides of which 25 possess cysteine residues including 16 with a single cysteine residue (no missed cleavages).

We generated detergent lysates of Jurkat cells, a human T-lymphoma cell line, and added known amounts of BSA that had been alkylated on its cysteine residues with dibromide tag **4**. After digestion with trypsin, the sample was separated by inline reversed-phase LC and analyzed on an Orbitrap mass spectrometer. Figure 2-3 shows the base peak chromatogram (Figure 2-3A) and a representative full scan mass spectrum (Figure 2-3B) of the sample derived from whole cell lysate (10 μg of total protein) to which 150 femtomoles of **4**-labeled BSA had been added. The peptide identified, LKPDPNTLC*DEFK (corresponding to residues 139-151), showed the correct isotopic envelope for a dibromide-labeled species at that mass (Figure 2-3C). Other labeled BSA-derived peptides were also identified in the full scan MS, collectively reflecting 30% coverage of cysteine-containing peptides. Notably, the pattern in Figure 2-3C (*black*) was found computationally at a signal-to-noise ratio (S/N) of 2.5:1 despite the presence of intervening peaks (*light gray*) within the envelope (Figure 2-3C). Using conventional shotgun proteomics methodologies, peaks at this low level of intensity would likely be excluded from tandem MS analysis. Without some sort of enrichment, a protein at this low level of abundance in such a complex sample would go undetected.

2.2.4 The dibromide tag is superior to the other halogen tags with respect to false positive rates and sensitivity

Using BSA as a substrate, we compared the performance of the dibromide tag with those of other halogen tags, first focusing on sensitivity. A central feature of the IsoStamp algorithm is that the user can tune its parameters to balance detection sensitivity against false positive rate. To determine the relative sensitivities of the tags, we

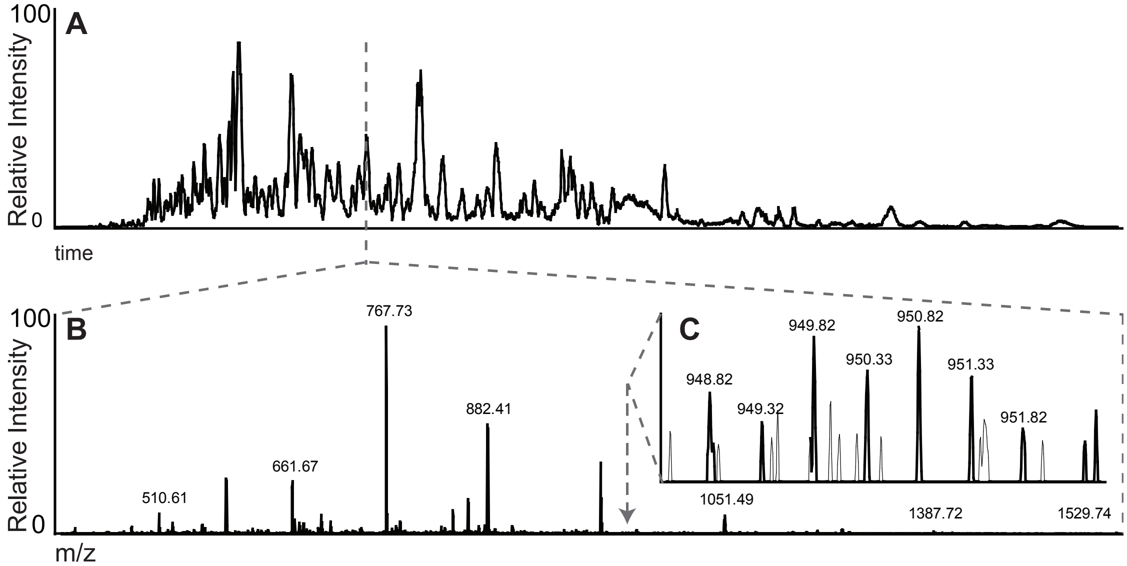


Figure 2-3: The dibromide motif can be recognized at extremely low signal to noise ratios. (**A**) The base peak chromatogram and (**B**) a representative spectrum of a proteolytic digest of Jurkat cell lysate (10 mg total protein) with 150 fmole of dibromide-labeled BSA added prior to trypsinization. (**C**) The expanded region shows the dibromide-labeled peptide LKPDPNTLC*DEFK at a S/N of 2.5:1.

therefore fixed the maximum allowed number of false positive identifications found by searching full-scan MS data derived from Jurkat cell lysates with no added BSA (and thus no real positives). We then performed a titration experiment where known quantities of labeled BSA were added to a fixed amount of Jurkat cell lysate. Each mixture was then proteolytically digested, subjected to LC-MS analysis, and the resulting data were searched for the tag's isotopic pattern. The number of true positive identifications as a function of protein concentration are shown in Figure 2-4A.

At all protein concentrations, the dibromide-labeled peptide were detected with a higher frequency than peptides labeled with any other tag. The data appear to converge at lower protein concentrations, but this may reflect the detection limits of the instrument rather than capabilities of the isotopic pattern searching algorithm (as mentioned above, the pattern is detectable at a S/N ratio as low as 2.5:1, Figure 2-3C). Overall, the dibromide isotopic signature was detected approximately twice as often as the dichloride and three times as often as the monobromide signatures (Figure 2-4A).

To analyze the relative false positive rates of the halogenated tags, the true positive detection rate was fixed in a sample containing 450 fmoles of labeled BSA in 10μg of Jurkat cell lysate. The false positive rate at these settings was then determined by searching the no-BSA data described above. In this setting, true positives were determined by a laborious manual analysis of the MS data with comparison to a list

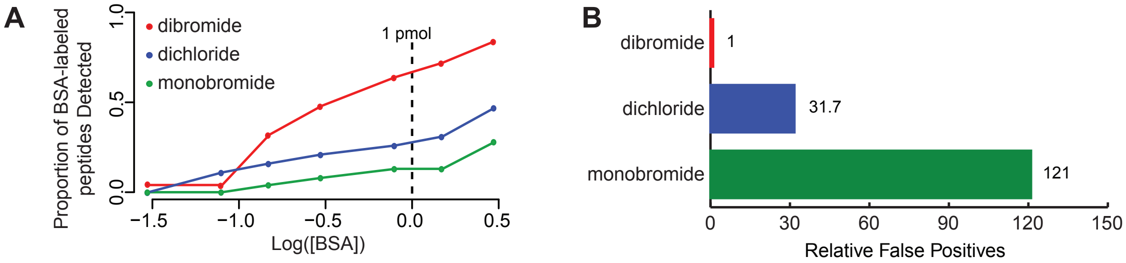


Figure 2-4: The dibromide motif is superior to other halogen tags with respect false positive rate and sensitivity. (**A**) Tryptic digest of Jurkat cell lysates were analyzed by LC-MS and the data were searched for signature motifs of the dibromide, dichloride and monobromide tags under searching conditions that found 50% of true positives when tagged BSA (150 fmoles) was initially added to the lysate. (**B**) Sensitivity was determined by titrating various amounts of halogen-tagged BSA into 10 μg of Jurkat cell lysate (based on total protein) and analyzing the digest by LC-MS. The number of peptides corresponding to tagged BSA was determined as a proportion of the maximum detectable by manual searching of the full scan MS data.

of predicted masses for tagged BSA species. True positives were compared to data derived from MS analysis of a tryptic digest of pure tagged BSA, which provided a measure of elution time and ionization efficiency for each authentic peptide.

The numbers of false positive identifications made using the dibromide, dichloride, and monobromide tags are shown in Figure 2-4B. Compared to the dibromide tag, the dichloride tag produced greater than 30-fold more false hits while the monobromide tag produced $>$ 120-fold more false hits. Overall, the dibromide tag outperforms the dichloride and monobromide tags by a substantial margin. We were unable to determine the sensitivity and false positive rate for the monochloride tag; reasonable searching parameters to detect 50% of the true positives in the pure BSA sample could not be found due to the minimal influence of this tag on the natural isotopic pattern of peptides.

2.2.5 The dibromide tag can be detected on small proteins

The impressive performance of the dibromide tag as a means to distinguish labeled from unlabeled tryptic peptides motivated us to explore its potential use in the identification of larger peptides or small proteins, a central challenge in the emerging area of middle down proteomics[44]. In addition to improving coverage and confidence in protein identifications, the analysis of larger protein fragments enables studies of multiple posttranslational modifications that might occur combinatorially on a single protein molecule[45]. Studies of this latter type could benefit tremendously from chemical tagging methods.

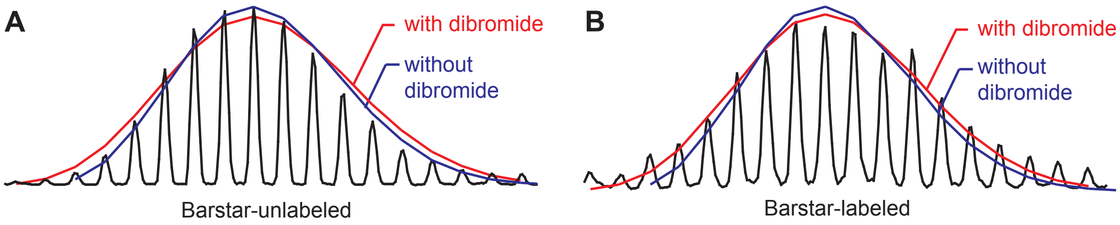


Figure 2-5: The dibromide motif is distinguishable on small proteins. The predicted isotopic envelope with (*red*) and without (*blue*) the dibromide tag fit to mass spectra of (**A**) barstar and (**B**) dibromide-labeled barstar. Barstar's predicted MW = 11.7 kDa. Spectra shown are from the +9 charge state.

To determine whether the dibromide tag's isotopic signature can be detected on larger peptides, we labeled the small protein Barstar from B. cenocepacia (11,659.8 Da, including a C-terminal 6xHis tag) with dibromide tag **4** on a single cysteine residue introduced by site-directed mutagenesis (I26C). The labeled intact protein was analyzed by LC-MS on an LTQ-Orbitrap XL mass spectrometer. Shown in Figure 2-5 are the mass spectra of unlabeled (A) and labeled (B) Barstar (*black*) in the +9 charge state. Using the averagine system, we predicted the mass envelope of the protein with and without the dibromide tag and depicted the peak intensities in the form of red and blue curves, respectively. An overlay of the two curves suggested that addition of the dibromide tag should cause a detectable widening of the mass envelope. We calculated the rms difference between the peak intensities from the actual MS data and each of the predicted mass envelopes. The experimental data for the dibromide tagged protein showed a considerably lower rms deviation from its averagine-predicted mass spectrum than from the predicted spectrum of the untagged protein (and vice versa). The fact that labeled and unlabeled species with masses approaching 12 kDa are computationally distinguishable suggests applications of the dibromide tag in middle down proteomics analyses.

2.3 Discussion

2.3.1 The uniqueness of the dibromide tag

The dibromide tag imparts a distinct isotopic signature on peptides by elevating the intensity of the $M+2$ peak with respect to the two leading peaks (Figure 2-2). By contrast, untagged peptides up to ~3000 Da typically display mass envelopes in which the leading peak is highest in intensity. The dichloride tag falls short because its isotopic signature enhances the intensity of M rather than $M+2$, reinforcing the mass pattern of the underlying peptide. The monobromide signature also enhances the intensity of the tagged peptide's $M+2$ peak relative to M, but not as dramatically as the dibromide pattern. Interestingly, the dichloride pattern has a similar impact on the relative intensity of $M+2$ compared to the monobromide motif, but is overall more detectable due to a broadening of the isotopic envelope. One might consider building additional halogens or other heavy elements into a tag to create a more distinct mass pattern, but there are diminishing returns. As the same total signal intensity is split among more peaks, sensitivity for their detection will be compromised. The dibromide signature strikes a balance by enabling high-fidelity pattern matching with good sensitivity.

The IsoStamp method can be employed in any chemically directed proteomics experiment in which a tag is covalently bound to target proteins; one need simply endow the tagging reagent with the dibromide signature. Such experiments include ICAT and iTRAQ, as well as emerging bioorthogonal ligation strategies that install uniquely reactive functionalities at sites of posttranslational modifications[24]. Affinity-based proteomics experiments in which tags are covalently bound to enzyme active site residues[31] and protein chemical crosslinking studies[46] can also benefit from integration of the IsoStamp method. In all cases, including a dibromide signature in the covalently bound tag will improve detection and identification of labeled peptides. Finally, the detectable mass pattern employed in the IsoStamp method can be generated in ways other than covalent chemical labeling strategies and using isotopic mixtures other than the naturally occurring dibromide isotopomers. For example, metabolic labeling with isotopomeric substrate mixtures can, in principle, endow biomolecules with unique mass patterns that are detectable without need for chemical labeling. Consequently, we envision numerous future application of IsoStamp in glycomics and metabolomics in addition to proteomics.

2.4 Materials and Methods

Synthesis

General. All chemical reagents were of analytical grade, obtained from commercial suppliers, and used without further purification unless otherwise noted. All reaction flasks were oven dried prior to use. Reactions were performed in an N_2 atmosphere and liquid reagents were added with a syringe unless otherwise noted. Reactions were analyzed with Analtech 250-mm silica gel G plates and visualized by staining with ceric ammonium molybdate, ninhydrin, or by absorbance of UV light at 254 nm. Organic extracts were dried over $MgSO_4$, and solvents were removed with a rotary evaporator at reduced pressure (20 torr), unless otherwise noted. Proton NMR spectra were obtained with a 400 MHz Bruker spectrometer. Chemical shifts are reported in ppm referenced to the solvent peak for ^{1}H NMR. Coupling constants (J) are reported in Hz. Reversed-phase HPLC was performed by using a Rainin Dynamax SD-200 HPLC system with 210-nm and 254-nm detection on a Microsorb C18 analytical or preparative column.

Halogenated tyrosine salts. The halogenated tyrosine salts **5**, **7**, **9**, and **11** were prepared according to literature procedures[47–49].

General procedure for preparation of alkylating agents. To the halogenated tyrosine salt (100 mg) in anhydrous DMF (0.25 mL) was added anhydrous sodium carbonate (2 equiv.) and the mixture left to stir for 0.5 hr at room temperature under a nitrogen atmosphere. Chloroacetyl chloride (1 equiv.) was added dropwise over 5 minutes and the reaction was stirred at room temperature under a nitrogen atmosphere for 1 hr before the reaction was transferred to a separatory funnel with EtOAc (15 mL). The organic layer was washed with 1 M HCl (5 mL) and the layers separated. The aqueous layer was extracted twice with EtOAc (5 mL) and the combined organic extracts were dried over $MgSO_4$ prior to removal of solvent by rotary evaporation. The crude material was then dissolved in anhydrous DMF (0.5 mL) and sodium iodide was added (6 equiv.). The mixture was stirred at room temperature in the dark under a nitrogen atmosphere for 24 hours before the mixture was transferred to a separatory funnel with EtOAc (15 mL). The organic layer was washed with water (5 mL) and the layers were separated. The aqueous layer was extracted twice with EtOAc (5 mL) and the combined organic extracts were dried over $MgSO_4$. The crude product was purified using reverse-phase HPLC with a gradient of 15%-60% acetonitrile containing 0.1% TFA over 40 minutes. All fractions were kept in foil until the solvent was removed by rotary evaporation to yield white solid. The synthetic scheme can be seen in Figure 2-6. **Compound 1.** ^{1}H NMR (400 MHz, CD3OD) d 7.36 (s, 2H), 4.59-4.55 (m, 1H), 3.73 (d, 1H, J = 9.6 Hz), 3.64 (d, 1H, J = 9.6 Hz), 3.11 (dd, 1H, J = 4.8 Hz, 14 Hz), 2.89-2.83 (m, 1H). **Compound 2.** ^{1}H NMR (400 MHz, CD3OD) d 7.33 (d, 1H, J = 2 Hz), 7.03 (dd, 1H, J = 2 Hz, 8.4 Hz), 6.80 (d, 1H, J = 8.4 Hz), 4.57-4.54 (m, 1H), 3.72 (d, 1H, J = 10 Hz), 3.65 (d, 1H, J = 10 Hz), 3.09 (dd, 1H, J = 4.8 Hz, 14 Hz), 2.88-2.83 (m, 1H). **Compound 3.** ^{1}H NMR (400 MHz, CD3OD) d 7.17 (d, 1H, J = 2 Hz), 6.99 (dd, 1H, J = 2 Hz, 8.4 Hz), 6.81 (d, 1H, J = 8.4 Hz), 4.58-4.54 (m, 1H), 3.72 (d, 1H, J = 10 Hz), 3.65 (d, 1H, J = 10 Hz), 3.09 (dd, 1H, J = 4.8 Hz, 14 Hz), 2.89-2.83 (m, 1H). **Compound 4.** ^{1}H NMR (400 MHz, CD3OD) d 7.16 (s, 2H), 4.56-4.53 (m, 1H), 3.73 (d, 1H, J = 9.6 Hz), 3.64 (d, 1H, J = 9.6 Hz), 3.11 (dd, 1H, J = 4.8 Hz, 14 Hz), 2.89-2.83 (m, 1H).

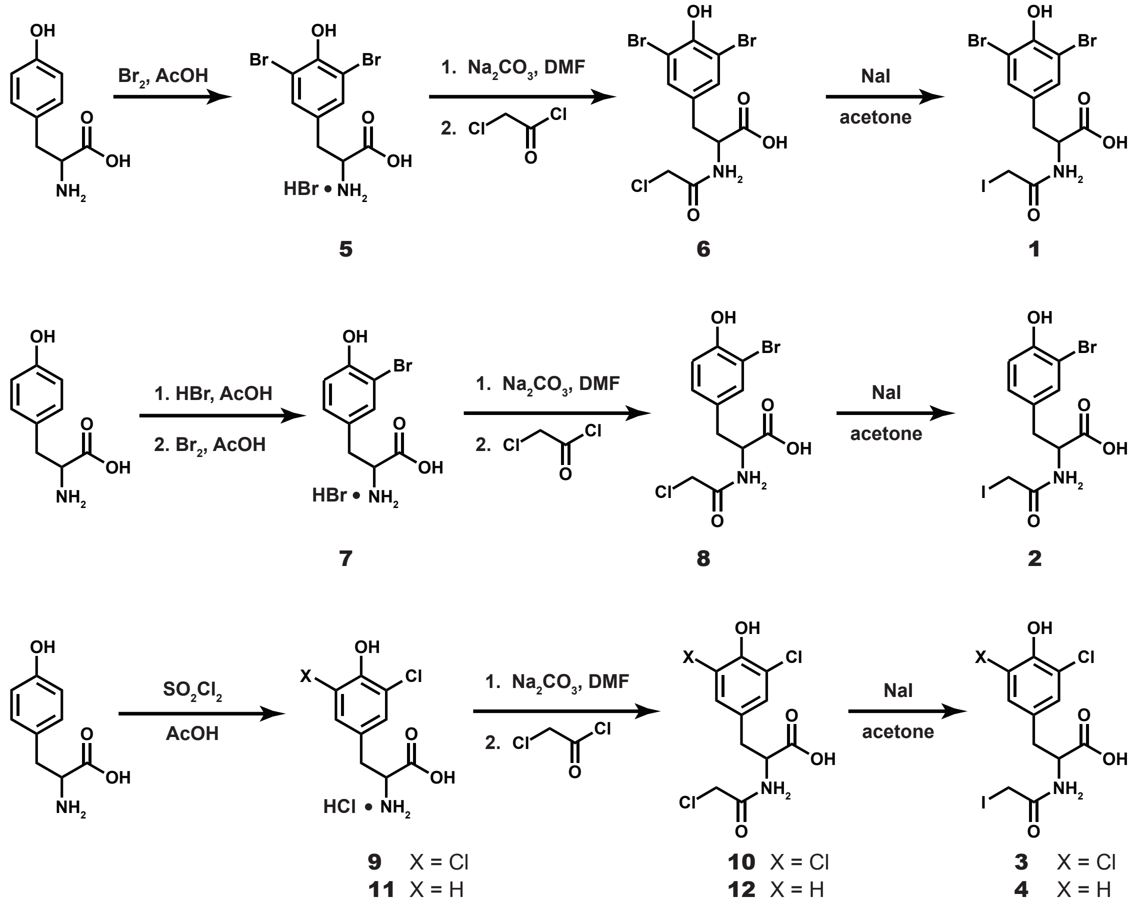


Figure 2-6: The synthesis of alkylating halide tags. See text for details.

Protein Labeling

A 2 mg/mL solution of BSA (100 μg) in 250 mM ammonium bicarbonate was reduced by adding DTT to a concentration of 2.5 mM and placed at 56°C for 30 min. After cooling to room temperature, the halogenated tag was added to a concentration of 10 mM from a 500 mM solution in DMF. The reaction was allowed to proceed at room temperature in the dark for 1 h. before quenching excess tag with 5 μL of a 1 M dithiothreitol solution for 30 min. The sample was subjected to size exclusion chromatography using a Bio-Rad Micro Bio-Spin 6 column to removed excess tag.

Lysate Preparation

Whole-cell Jurkat lysate was prepared from cultures that were lysed in a buffer containing 1% Triton-X100, 20 mM Tris pH 7.4, 150 mM NaCl and protease inhibitors (inhibitor cocktail III from Calbiochem). Following lysis, the sample was precipitated using 9 volumes of acetone and placed at -20°C for two hours followed by centrifugation at 13,000 rpms for 20 minutes at 4°C . The supernatant was removed and the pellet was resolubilized in 8 M urea buffered to pH 8.0. A BCA assay was performed to determine protein concentration followed by dilution to 1 mg/mL.

Serial Dilutions and Protein Digestion

Labeled BSA samples were serially diluted into 10mL of 1mg/mL whole-cell Jurkat lysate at concentrations of 0.03, 0.08, 0.15, 0.30, 0.80, 1.50, 3.0, and 30.0 picomoles. The samples were then subjected to trypsin digestion (50:1 protein/enzyme) at 37°C for 16 h. Prior to MS analysis, the peptides were desalted using Millipore C18 zip tips according to the manufacturer's instructions.

LC-MS Analysis

All samples were subjected to reversed-phase capillary chromatography with an Agilent 1200 LC system using a 100-μm X 1-cm (5-μm, 200 Magic C18AQ resin; Michrom Bioresources, Auburn, CA) fritted capillary pre-column and a 100-μm X 10-cm self-packed C18 column (5-μm, 100 Magic C18AQ resin; Michrom Bioresources, Auburn, CA). A binary solvent system consisting of buffer A (0.1% formic acid) and buffer B (0.1% formic acid in acetonitrile) was employed. After a 10 min. loading step in 2% buffer B, a gradient was employed from 10% to 40% buffer B for 62 min., followed by a washing step in 99% buffer B for 10 min. A solvent split was used to maintain a flow rate of 400 nL/min at the column tip. Data were collected on a Thermo-Finnigan Orbitrap mass spectrometer set to 60,000 resolution in full scan mode with an m/z scan range of 400-2000.

Data Processing

All data was collected in profile mode. Noise reduction and peak detection was performed using software developed in house based on the method described by Du *et al.* which makes use of a continuous wavelet transform[50]. The resulting centroided mzXML files were then searched for the presence of a desired isotopic pattern using software developed in house following the algorithm described above.

True Positive Determination

In order to analyze the performance of the searching algorithm, it was necessary to analyze the data independently to determine the presence of labeled BSA peptides. This was done by performing an *in silico* digestion of BSA with up to two missed tryptic cleavage sites, and predicting the mass of tagged cysteinyl peptides. From this list of putative masses, the raw data was then analyzed manually by obtaining an extracted ion chromatogram (EIC) for each predicted mass and allowing for up to five charges. Each EIC was then used to determine the visual presence of the dibromide pattern. Peptides were considered to be 'found' if at least one charge state of the predicted mass value with an appropriate isotopic signature was detected. It should be noted that this method is extremely time consuming and only possible if the masses of the peptides are known in advance.

Barstar Mutagenesis and protein purification

A plasmid containing the Bacillus amyloliquefaciens protein Barstar as a 6xHis fusion in a pQE30 expression vector was obtained from D. Tirrell (California Insititue of Technology). A construct encoding the single point mutant I26C was prepared using the Quickchange protocol (Stratagene) using the primers:

Forward: $5'$—GGG GAA CAA ATC AGA AGT TGC AGC GAC CTC CAC CAG AC—$3'$
Reverse: $5'$—GTC TGG TGG AGG TCG CTG CAA CTT CTG ATT TGT TCC CC—$3'$

The mutant was expressed in M15-MA[pREP4] cells obtained from D. Tirrell (California Institiue of Technology). Individual transformants were used to inoculate 5mL LB starter cultures supplemented with 200 μg/mL Amp and 35 μg/mL Kan. After an overnight incubation at 37°C with shaking, 1 mL was transferred to 50 mL of the same media. Protein expression was induced with IPTG when the OD_{600} reached 0.7. Cultures were clarified by centrifugation 4 h post induction and Barstar was purified under denaturing conditions using Ni-NTA spin columns according to manufacturer's specifications (Qiagen). Mass was verified by high-resolution mass spectrometry, expected 11667.13 Da found 11667.2 Da.

Barstar Labeling

20 μg of I26C Barstar in 50 μL of 250 mM ammonium bicarbonate was reduced by adding DTT to a concentration of 2.5 mM and placed at 56 °C for 30 min. After cooling to room temperature, the dibromo tag was added to a concentration of 10 mM from a 500 mM solution in DMF. The reaction was allowed to proceed at room temperature in the dark for 1 h. before quenching excess tag with 10 μL of a 1 M DTT solution at r.t. for 30 min. The sample was subjected to size exclusion chromatography using a Bio-Rad Micro Bio-Spin 6 column to removed excess tag.

LC-MS analysis of intact proteins

Intact protein samples were analyzed using an Agilent 1200 series liquid chromatograph (LC; Santa Clara, CA) that was connected in-line with an LTQ Orbitrap XL hybrid mass spectrometer equipped with an Ion Max electrospray ionization source (ESI; Thermo Fisher Scientific, Waltham, MA). The LC was equipped with C8 guard (Poroshell 300SB-C8, 5 μm, 12.5 × 2.1 mm, Agilent) and analytical (75 × 0.5 mm) columns and a 100 μL sample loop.

Solvent A was 0.1% formic acid/99.9% water and solvent B was 0.1% formic acid/99.9% acetonitrile (v/v). Sample solutions contained in 0.3 mL polypropylene snap-top vials sealed with rubber septa caps (Wheaton Science, Millville, NJ) were loaded into the Agilent 1200 autosampler compartment prior to analysis. For each sample, approximately 100 to 200 picomoles of protein analyte was injected onto the column. Following sample injection, analyte trapping was performed for 5 min with 99.5% A at a flow rate of 90 μL/min. The elution program consisted of a linear gradient from 30% to 95% B over 19.5 min, isocratic conditions at 95% B for 5 min, a linear gradient to 0.5% B over 0.5 min, and then isocratic conditions at 0.5% B for 9.5 min, at a flow rate of 90 μL/min.

The column and sample compartments were maintained at 35°C and 10°C , respectively. Solvent (Milli-Q water) blanks were run between samples, and the autosampler injection needle was rinsed with Milli-Q water after each sample injection, to avoid cross-contamination between samples. The connections between the LC column exit and the mass spectrometer ion source were made using PEEK tubing (0.005” i.d. × 1/16” o.d., Western Analytical, Lake Elsinore, CA). External mass calibration was performed prior to analysis using the standard LTQ calibration mixture containing caffeine, the peptide MRFA, and Ultramark 1621 dissolved in 51% acetonitrile/25% methanol/23% water/1% acetic acid solution (v/v).

The ESI source parameters were as follows: ion transfer capillary temperature 275°C , normalized sheath gas (nitrogen) flow rate 25%, ESI voltage 2.5 kV, ion transfer capillary voltage 33 V, and tube lens voltage 125 V. Mass spectra were recorded in the positive ion mode over the range m/z = 500 to 2000 using the Orbitrap mass analyzer, in profile format, with a full MS automatic gain control target setting of 5 × 105 charges and a resolution setting of 6 × 104 (at m/z = 400, FWHM). Raw mass spectra were processed using Xcalibur software (version 4.1, Thermo) and measured charge state distributions were deconvoluted using ProMass software (version 2.5 SR-1, Novatia, Monmouth Junction, NJ), using default “small protein” parameters.

Analysis of Barstar data

Barstar data was analyzed for goodness of fit against an averagine model with and without a dibromide tag (Figure 2-5). Averagine isotopic envelopes were predicted based on the molecular weight of barstar. Models were fit against the centroided data by doing a binary search on the parameter z to minimize the sum-of-squares difference between the data and the model (Equation 2.1). After alignment, the fit was scored according to Equation 2.2 with $\epsilon = 10^{-5}$, σ = 25% RMS intensity, and $N = 16$. As scores produced in this manner are typically very small, the log values of the scores were compared:

model	**unlabeled barstar**	**labeled barstar**
averagine	-17.2	-35.5
averagine + Br_2	-21.7	-20.5

indicating that the tagged model fits the tagged data most strongly than the untagged model, and vice versa.

Acknowledgements

Dr. Brian Smart was a very close collaborator on this project, and is responsible for the initial idea of adding two bromines to a chemical tag. Brian performed all of the synthesis and most of the MS experiments presented here. Kanna Palaniappan performed the barstar mutigenesis and purification, and assisted in a number of the MS experiments. Special thanks are also due to Dr. Mike Boyce for his help in the preparation of biological samples.

References

[1] Anderson, N. L & Anderson, N. G. (2002) The human plasma proteome: history, character, and diagnostic prospects. *Mol. Cell. Proteomics* **1**: 845–867.

[2] Dell, A & Morris, H. R. (2001) Glycoprotein structure determination by mass spectrometry. *Science* **291**: 2351–2356.

[3] Domon, B & Aebersold, R. (2006) Mass spectrometry and protein analysis. *Science* **312**: 212–217.

[4] Griffiths, W. J & Wang, Y. (2009) Mass spectrometry: from proteomics to metabolomics and lipidomics. *Chem. Soc. Rev.* **38**: 1882–1896.

[5] Olsen, J. V et al. (2006) Global, *in vivo*, and site-specific phosphorylation dynamics and signaling networks. *Cell* **127**: 635–648.

[6] Yu, L.-R, Issaq, H. J, & Veenstra, T. D. (2007) Phosphoproteomics for the discovery of kinases as cancer biomarkers and drug targets. *Proteomics Clin. Appl.* **1**: 1042–1057.

[7] Wisniewski, J. R, Zougman, A, Nagaraj, N, & Mann, M. (2009) Universal sample preparation method for proteome analysis. *Nature Methods* **6**: 359–362.

[8] Conrads, T. P, Anderson, G. A, Veenstra, T. D, Pasa-Tolic, L, & Smith, R. D. (2000) Utility of accurate mass tags for proteome-wide protein identification. *Anal. Chem.* **72**: 3349–3354.

[9] Link, A. J et al. (1999) Direct analysis of protein complexes using mass spectrometry. *Nat. Biotechnol.* **17**: 676–682.

[10] Bruce, J. E, Anderson, G. A, Wen, J, Harkewicz, R, & Smith, R. D. (1999) High-mass-measurement accuracy and 100% sequence coverage of enzymatically digested bovine serum albumin from an ESI-FTICR mass spectrum. *Anal. Chem.* **71**: 2595–2599.

[11] Patterson, S. D & Aebersold, R. (1995) Mass spectrometric approaches for the identification of gel-separated proteins. *Electrophoresis* **16**: 1791–1814.

[12] Eng, J. K, McCormack, A. L, & Yates III, J. R. (1994) An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database. *J. Am. Soc. Mass. Spectrom.* **5**: 976–989.

[13] Mann, M & Wilm, M. (1994) Error-tolerant identification of peptides in sequence databases by peptide sequence tags. *Anal. Chem.* **66**: 4390–4399.

[14] Picotti, P. R, Aebersold, R, & Domon, B. (2007) The implications of proteolytic background for shotgun proteomics. *Mol. Cell. Proteomics* **6**: 1589–1598.

[15] Schmidt, A, Claassen, M, & Aebersold, R. (2009) Directed mass spectrometry: towards hypothesis-driven proteomics. *Cur. Op. Chem. Biol.* **13**: 1–8.

[16] Leitner, A & Lindner, W. (2004) Current chemical tagging strategies for proteome analysis by mass spectrometry. *J. Chrom. B. Analyt. Technol. Biomed. Life Sci.* **813**: 1–26.

[17] Block, H et al. (2009) Immobilized-metal affinity chromatography (IMAC): a review. *Methods Enzymol.* **463**: 439–473.

[18] Jiang, D, Jarrett, H. W, & Haskins, W. E. (2009) Methods for proteomic analysis of transcription factors. *J. Chrom. A* **1216**: 6881–6889.

[19] Pernemalm, M, Lewensohn, R, & Lehtiö, J. (2009) Affinity prefractionation for MS-based plasma proteomics. *Proteomics* **9**: 1420–1427.

[20] Gygi, S. P et al. (1999) Quantitative analysis of complex protein mixtures using isotope-coded affinity tags. *Nature Biotechnology* **17**: 994–999.

[21] Ross, P. L et al. (2004) Multiplexed protein quantitation in *saccharomyces cerevisiae* using amine-reactive isobaric tagging reagents. *Mol. Cell. Proteomics* **3**: 1154–1169.

[22] Whetstone, P. A, Butlin, N. G, Corneille, T. M, & Meares, C. F. (2004) Element-coded affinity tags for peptides and proteins. *Bioconjugate Chem.* **15**: 3–6.

[23] Adamczyk, M, Gebler, J. C, & Wu, J. (1999) A simple method to identify cysteine residues by isotopic labeling and ion trap mass spectrometry. *Rapid Commun. Mass Spectrom.* **13**: 1813–1817.

[24] Heal, W. P & Tate, E. W. (2010) Getting a chemical handle on protein post-translational modification. *Org. Biomol. Chem.* **8**: 731–738.

[25] Saxon, E & Bertozzi, C. R. (2000) Cell surface engineering by a modified staudinger reaction. *Science* **287**: 2007–2010.

[26] Heal, W. P, Wickramasinghe, S. R, Leatherbarrow, R. J, & Tate, E. W. (2008) N-myristoyl transferase-mediated protein labeling *in vivo*. *Org. Biomol. Chem.* **6**: 2308–2315.

[27] Roth, A. F et al. (2006) Global analysis of protein palmitoylation in yeast. *Cell* **125**: 1003–1013.

[28] Blethrow, J. D, Glavy, J. S, Morgan, D. O, & Shokat, K. M. (2008) Covalent capture of kinase-specific phosphopeptides reveals Cdk-1-cyclin B substrates. *Proc. Natl. Acad. Sci.* **105**: 1442–1447.

[29] Agard, N. J, Maltby, D, & Wells, J. A. (2010) Inflammatory stimuli regulate caspace substrate profiles. *Mol. Cell. Proteomics* **9**: 880–893.

[30] Reddie, K. G, Seo, Y. H, Muse III, W. B, Leonard, S. E, & Carroll, K. S. (2008) A chemical approach for detecting sulfenic acid-modified proteins in living cells. *Molecular Biosystems* **4**: 521–531.

[31] Meier, J. L et al. (2009) An orthogonal active site identification system (OASIS) for proteomic profiling of natural product biosynthesis. *ACS Chem. Biol.* **4**: 948–957.

[32] Carlson, E. E & Cravatt, B. F. (2007) Enrichment tags for enhanced-resolution profiling of the polar metabolome. *J. Am. Chem. Soc.* **129**: 15780–15782.

[33] Li, M & Kinzer, J. A. (2003) Structural analysis of oligosaccarides by a combination of electrospray mass spectrometry and bromine isotope tagging of reducing-end sugars with 2-amino-5-bromopyridine. *Rapid Commun. Mass Spectrom.* **17**: 1462–1466.

[34] Paulick, M. G et al. (2006) Cleavable hydrophilic linker for one-beat-one-compound sequencing of oligomer libraries by tandem mass spectrometry. *J. Comb. Chem.* **8**: 417–426.

[35] Mirzaei, H et al. (2009) Halogenated peptides as internal standards (H-PINS): introduction of an MS-based internal standard

set for liquid chromatography-mass spectrometry. *Mol. Cell. Proteom.* **8**: 1934–1946.

[36] Goodlett, D. R et al. (2000) Protein identification with a single accurate mass of a cysteine-containing peptide and constrained database searching. *Anal. Chem.* **72**: 1112–1118.

[37] Miyagi, M, Nakao, M, Nakazawa, T, Kato, I, & Tsunasawa, S. (1998) A novel derivatization method with 5-bromo-nicotinic acid n-hydroxysuccinamide for determination of the amino acid sequences of peptides. *Rapid Commun. Mass Spectrom.* **12**: 603–608.

[38] Hall, M. P & Schneider, L. V. (2004) Isotope-differentiated binding energy shift tags (IDBEST) for improved targeted biomarker discovery and validation. *Expert Rev. Proteomics* **1**: 421–431.

[39] Hernandez, H et al. (2006) Mass defect labeling of cysteine for improving peptide assignment in shotgun proteomic analyses. *Anal. Chem.* **78**: 3417–3423.

[40] Zhang, H, Zhu, M, Ray, K. L, Ma, L, & Zhang, D. (2008) Mass defect profiles of biological matrices and the general applicability of mass defect filtering for metabolite detection. *Rapid Commun. Mass Spectrom.* **22**: 2082–2088.

[41] Desiere, F et al. (2004) Integration with the human genome of peptide sequences obtained by high-throughput mass spectrometry. *Genome Biology* **6**: R9.

[42] Tanner, S et al. (2007) Improving gene annotation using peptide mass spectrometry. *Genome Research* **7**: 231–239.

[43] Senko, M. W, Beu, S. C, & McLafferty, F. W. (1995) Determination of monoisotopic masses and ion populations for large biomolecules from resolved isotopic distributions. *J. Am. Soc. Mass. Spectrom.* **6**: 229–233.

[44] Garcia, B. A. (2010) What does the future hold for top down mass spectrometry? *J. Am. Soc. Mass. Spectrom.* **21**: 193–202.

[45] Wu, S, Kim, J, Hancock, W. S, & Karger, B. (2005) Extended range proteomic analysis (ERPA): a new and sensitive LC-MS platform for high sequence coverage of complex proteins with extensive post-translational modifications–comprehensive analysis of β-casein and epidermal growth factor receptor (EGFR). *J. Prot. Res.r* **4**: 1155–1170.

[46] Melcher, K. (2004) New chemical crosslinking methods for the identification of transient protein-protein interactions with multiprotein complexes. *Curr. Protein Pept. Sci.* **5**: 287–296.

[47] Ding, W et al. (2004) The synthesis, distribution, and anti-hepatic cancer activity of YSL. *Bioorg. Med. Chem.* **12**: 4989–4994.

[48] Prieto, M, Mayor, S, Rodriguez, K, Lloyd-Williams, P, & Giralt, E. (2007) Racemization in Suzuki couplings: a quantitative study using 4-hydroxyphenylglycine and tyrosine derivatives as probe molecules. *J. Org. Chem.* **72**: 1047–1050.

[49] McCubbin, J. A, Maddess, M. L, & Lautens, M. (2006) Total synthesis of cryptophycin analogues via a scaffold approach. *Org. Lett.* **8**: 2993–2996.

[50] Du, P, Kibbe, W. A, & Lin, S. M. (2006) Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching. *Bioinformatics* **22**: 2059–2065.

chapter three

Automated isotopic pattern searching in complex mass spectral data

3.1 Introduction

The isotopic pattern imparted by the dibromide tag and similar isotopic fingerprint labeling strategies allows tagged molecules to be detected among a sea of untagged species. However, as a typical proteomics experiment can produce upwards of 1 GB of raw data*, such a tagging strategy is of limited utility if the patterns must be matched by hand. Thus, it is extremely advantageous, if not absolutely necessary, to automate this process. In the absence such an automated processes, the human input required to locate tagged species is considerable. Furthermore, in the absence of automation, the requirement for performing two LC-MS experiments on every sample is absolute—a first experiment must be done to identify patterns of interest, and a second experiment must be done to perform MS^2 experiments on detected species. With a reasonably fast automated searching algorithm, it is theoretically possible to perform both experiments concurrently, though this will likely require collaboration with instrument manufacturers to embed the searching code into the machine's operational software. To achieve these aims, a software package was developed to facilitate the rapid and tunable analysis of large LC-MS data sets.

3.2 Design of an automated pattern matching algorithm

3.2.1 Overview

There are a number of properties desirable in an isotopic pattern-searching algorithm. Ideally, any automated system for pattern searching should have number of properties, as outlined in Table 3-1.

*mzXML files generated from Orbitrap RAW files are typically in the 1-3 GB range, while those from time-of-flight instruments can be even larger

Table 3-1: Desired properties of a pattern searching system

1. Able to detect the desired pattern over a large range of signal intensities
2. Able to match imperfect spectra
3. Produces a reasonably low number of false-positive IDs
4. Able to search a large dataset in a short period of time
5. Capable of detecting patterns in the presence of contaminating isotopic peaks (e.g., ^{13}C, ^{15}N, etc)
6. Readily adaptable to searching for other isotopic patterns, and the tolerances should be readily tunable

The first three requirements are essential to due to the nature of LC-MS data: searching must be possible over the entire dynamic range of the instrument, often spanning over five orders of magnitude in signal intensity. The algorithm must also be able to account for noise endemic in complex peptide LC-MS data, including such issues as different ions with overlapping spectra and imperfect reproduction of an ion's true isotopic envelope, particularly at low signal intensities[1]. Because we wish to use this algorithm to perform directed proteomics, the searching must produce a low number of false positive matches. The number of false-positive identifications that is reasonable depends on a number of factors, but it is usually desirable that this number be significantly less than 1,000 so that the instrument does not spend an unreasonable amount of time performing secondary fragmentation on untagged molecules. In the case of doing secondary fragmentation with ETD (currently the slowest method, requiring 200 ms for each fragmentation event), 1,000 secondary fragmentation events would take 200 seconds, or 5% of the total time allotted to perform secondary fragmentation in a 90 minute LC-MS experiment.

The requirement for the algorithm run-time is desirable so that this methodology can eventually be performed in a single experiment, *i.e.*, so that the instrument can detect an isotopic pattern and immediately perform tandem fragmentation on an ion at runtime. Requirements five is due to the chemical nature of large biomolecules such as peptides. As the molecular weight increases, the effects of naturally occurring heavy isotopes of common elements can skew a given isotopic pattern, as is demonstrated in Figure 3-1. It is essential that the algorithm accounts for the isotopic contributions from the peptides themselves, as this contribution is non-negligible (see Table 2-1). In particular, ^{13}C, ^{15}N, ^{18}O, and ^{34}S make significant contributions to the isotopic pattern of peptides. This effect becomes more pronouced as the molecular weight of peptides becomes larger, as can be seen in Figure 3-1. A method for rapidly determining the contribution of such 'contaminating' isotopes to the overall isotopic envelope is described in Chapter 4.

Finally, to add flexibility, requirement six was added to facilitate adaptation of the algorithm to new experimental conditions, including the possible use of other isotopic fingerprinting tags. These requirements impose a number of restrictions on the algorithms we can use. The first restriction prevents us from using any scoring system that depends absolutely on intensities, while the second requirement demands pattern matching to be approximate—due to signal variability, it is quite likely that isotopic envelopes of real labeled ions may not be faithfully reproduced in real-world data.

Of those requirements listed in Table 3-1, the third is by far the most difficult. We require an algorithm that can find (ideally) all real labeled species while minimizing the number of false positives. Furthermore, we want this requirement to hold in extremely complex samples to the extent that we can find low-abundance labeled species against a background of unlabeled peptides (*e.g.*, from total cell lysate). This requirement conflicts with the tolerance requirement, as it is difficult to have a tolerant pattern matching system that also does not produce false positives. Indeed, the stronger you make the tolerance requirements, the harder this requirement becomes to realize. To overcome this problem, certain properties of LC-MS data and mass spectrometry in general can be taken into account. Firstly, since chromatography is involved, one can look for the same molecular species eluting over a number of spectral scans close in time. Secondly, since it is common for peptides to appear in numerous charge states, we can look for several ions of the same molecular species. While none of these properties in guaranteed, by scoring matches based on pattern quality, temporal correlation, and the existence of multiple charge states we can eliminate a large number of false positives that would be identified by naïve algorithm. A graph-theoretic construct which takes these factors into account to reduce false positive rates is introduced in Section 3.8.

3.2.2 Isotopic correction in pattern searching

There are primarily two approaches to dealing with this isotopic contamination. The first method is to "de-isotope" mass spectral data before analysis[2,3]. While there are some algorithms available for this method, most are not sufficiently robust for the purpose of pattern matching in real time. This approach is made more complicated by the fact that we don't want to remove *all* isotopic peaks, but rather only those that do not come from the tag we are interested in. This is a significant hurdle for any de-isotoping strategy. The second approach works in reverse—rather than try to remove isotopic contamination, an attempt can be made to predict it. Predicting the isotopic

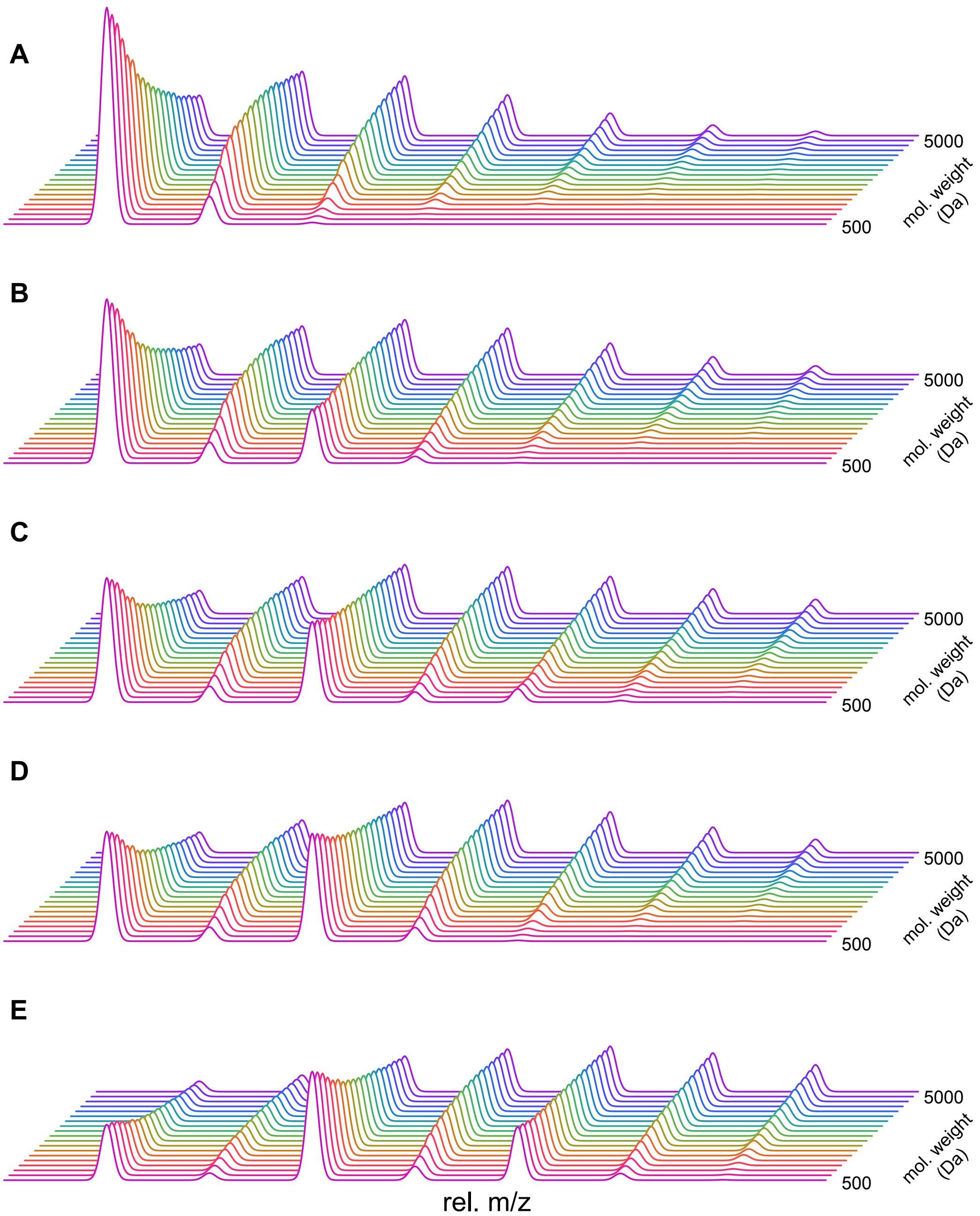


Figure 3-1: The natural abundance of biological elements produce a significant skewing effect that can be seen in the models here. (**A**) The change in isotopic distribution in an average peptide ranging in weight from 500 Da to 5 kDa . The isotopic distribution of the same peptides can be seen with the addition of a (**B**) monochloride, (**C**) dichloride, (**D**) monobromide, or (**E**) a dibromide label.

distribution of a molecule is a known problem, and requires only that we know the number and natural abundances of all elements in a molecule of question.

Since the natural abundances of the elements typically found in peptides are known, the problem reduces to determining the number of atoms of each element. This problem is also solvable when proteins are examined on the atomic level. Where the amino acid composition varies greatly between peptides, the elemental composition remain constant. Such a system has been described by Senko, *et al.*, and is termed the 'averagine' system[4]. This system as applied to pattern matching is analyzed in Section 4.2 and extended so as to be applicable to a wider range of biomolecules in Section 4.3. With knowledge about the natural abundance of the relevant elements, and can estimate the number of each one of these in a peptide if we know the molecular weight. Rather than attempting to de-isotope data prior to pattern searching, we can instead leave the data in its original format and search for an adaptable "smart" pattern—one that is updated to reflect what the pattern would look like when attached to a peptide of the appropriate molecular weight. This system also helps facilitate the final goal of facile adaptation of the algorithm to search for other (non-dibromide) patterns: rather than change the de-isotoping algorithm to permit other isotopes to exist (a rather complex problem), one can simply start with a different pattern and adapt it to the data as it is searched.

3.3 The structure of peptide LC-MS data

Before continuing to the description of the searching algorithm employed here, it is necessary to briefly describe the structure of peptide LC-MS data. To a first, data collected from an LC-MS experiment is simply a two-dimensional set of intensities, with each intensity being a function of elution time, t, and of the mass-to-charge ratio, m/z:

$$I(t, m/z)$$

However, due to the nature in which the data are collected, the time domain consists of a discrete set of slices through the m/z domain termed *scans*. Depending on the exact instrument used, scans may or may not be separated by a constant time.

A complication in processing LC-MS data is that, at present, there is no standard binary representation of the data in use, and each manufacturer uses a different format for storing the data. This has lead to the adoption of the open formats mzXML[5,6] and, more recently, mzML[7]. At the time of this writing, mzML has yet to gain wide support. As such, mzXML was chosen as the primary means of storing and accessing data, and a C++ library to read and write this data format can be found in Appendix B.2. For LC-MS data, an mzXML file consists header data along with a set of scan blocks, each of which contains a set of peaks. Each set of peaks is a list of floating point values representing m/z–intensity pairs. The entire mzXML schema can be found in Pedrioli *et al.*, 2004[5]. Due to the structure of the files, it is possible to randomly access the entire set of peaks for an individual scan, but accessing the intensity of an individual m/z value across multiple scans is computationally intensive as it requires reading and decoding the entire set of peaks for each scan of interest. Thus, it is typically advantageous to processes each scan in its entirety before moving to the next scan to prevent re-reading the same data. It should also be noted that, presently, it is not reasonable to load the entire set of peaks into memory at once, as an entire mzXML file can be several gigabytes in size.

3.4 A multi-step pattern searching process

The entire pattern searching process can be broken down into a number of steps as outlined in Figure 3-2. The first step, peak detection is a well documented process[8–11] (reviewed by Zhang *et al.*[12]). For our purposes, a continuous wavelet transform was chosen for this process as it produces accurate intensities and m/z values while providing strong noise filtration[8]. Details of the algorithm along with computer source code can be found in Appendix C. After peak detection, each full scan from an LC-MS experiment is analyzed independently. Each scan is first separated into sets of peaks that could potentially be isotopically related (see Section 3.5). This is done to simplify the computational complexity of pattern searching on large data sets by significantly reducing the search space. Once these peak sets are obtained, each is searched for the presence of the isotopic pattern of interest, with the pattern modified to account for contaminating isotopes (Section 3.7). If a match is made that is sufficiently good, information on that match (m/z ratio, signal intensity, charge state, retention time, etc...) is saved for further analyses. The above processes is repeated for each scan in the LC-MS file.

Once a list of putative matches has been obtained, they are further analyzed for the presence of multiple charge states due to the same molecular species and temporal elution profile (Section 3.8). Potential matches are then scored on a variety of criteria, and features that meet a minimum cutoff are then exported for further analysis or use in future experiments (Section 3.9)[†].

[†] Features can be output as a list of likely peptide molecular weights (with associated charge states

1. Perform peak detection to obtain a 'centroided' set of peaks for each scan.
2. Identify putative matches in a full scan mass spectrum
   (a) Divide each full scan into sets of peaks that could be isotopically related
   (b) Identify potential patterns in each of these sets
   (c) Save sufficiently good matches for further analysis
3. Analyze putative matches using a graph theoretic construct
   (a) Treat each putative match as a node in a graph
   (b) If two nodes have similar R_f values and could have been produced by the same molecular species, create an edge between those nodes.
   (c) Decompose the graph into disjoint sets based on the edges created
4. Score analyzed matches

Figure 3-2: An overview of the pattern searching process.

3.5 Simplifying full scan mass spectral data

The first step in the searching process is to divide each full scan into sets of peaks that are potentially isotopically related. This is done to reduce the searching space for the pattern matching step. To achieve this goal, a graph construct has been employed. In this construct, each peak is treated as a node on a graph, and edges are drawn between two nodes if the corresponding peaks could be isotopically related. Peaks are considered to be potentially related if their m/z values differ by $\frac{1}{charge}$ or $\frac{2}{charge}$ units (with some tolerance), where $charge$ is an integer in a predetermined range representing the possible charge state of the ion (see Algorithm 3-1). Separations of $\frac{2}{charge}$ are allowed to account for the possibility that peaks may be absent from the isotopic envelope.

Once a graph has been constructed for the scan and edges have been created where appropriate, the graph is decomposed into disjoint sets[‡]. Subsets produced in this way are typically much smaller that the total set of peaks for the scan, so this is an efficient method of reducing the complexity of the data without losing meaningful

observed) or a list of ions for an inclusion list. If the searching code can be embedded in the instrument control software, it would be trivial to output commands to perform MS^{2}to the mass spectrometer.

[‡] Sets such that if two nodes $x \in X$ and $y \in Y$ shared an edge, then $X = Y$.

```
difference = $|mz_1 - mz_2|$
related = $false$
for $charge = charge\_min$ to $charge\_max$ do
    if $|\text{difference} - 1.0/\text{charge}| \leqslant \delta m$
      $related = true$
    fi
    if $|\text{difference} - 2.0/\text{charge}\,| \leqslant \delta m$
      $related = true$
    fi
od
return related
```

Algorithm 3-1: Determining if two peaks could be isotopically related. mz_1 and mz_2 are the m/z values of two peaks of interest, $charge\_min$ and $charge\_max$ are the minimum and maximum charge states to be considered, and δm is the m/z tolerance defined for the algorithm (dependent on the instrument). The second conditional is added to make the algorithm robust against missing peaks.

information, as no isotopic patterns can be lost using this method unless they are missing two or more of the isotopic peaks. It is important to note that at this point, peaks in a given subset are not necessarily charge-state consistent. That is, in a given subset, there could be peaks separated by 1.0 $\pm\delta m$ mass units as well as peaks separated by 0.33 $\pm\delta m$ mass units (representing the charge states +1 and +3 respectively), and so forth. These possible inconsistencies are handled at a later stage of the searching process.

Pattern intensity matching is subsequently performed on sets of peaks that are charge-state consistent (*i.e.*, evenly spaced, with spacing of 1.0, 0.5, 0.33, 0.25, 0.2, $\cdots$ m/z units). These are easily obtained from the peak sets obtained in the previous section, and each set of charge-state consistent peaks is a subset of the isotopically related subsets generated in the previous step. From the peak sets previously obtained, each charge state is searched for subsets of peaks that line up with the correct spacing, allowing for missed peaks in the set. Missing peaks are added as peaks with the correct m/z value but having zero intensity.

3.6 Pattern scoring

Essential to the pattern matching step used in this algorithm is a method to score individual patterns. Here are discussed various schemes for scoring an isotopic pattern. For purposes here, a "pattern" is considered to be a set of intensities assumed to be separated by appropriate (and known) m/z distances, and can be represented mathematically by a vector $\vec{a}$.

3.6.1 Comparing multiple patterns

The purpose of pattern scoring is to differentiate between the isotopic pattern produced by a real labeled peptide and various other patterns—real unlabeled peptides, low intensity noise, and other contaminates that may be in the sample (polymers, etc). Because we want the scoring to be extendable so that if a new contaminate arises we can account for it, a Bayesian type scoring system makes sense. Bayes theorem, originally stated[13,14] as

$$P(x|data) = \frac{P(data|x)P(x)}{\sum_i P(data|x_i)P(x_i)} \tag{3.1}$$

where the term $P(x)$ is the probability that a given event x occurs at all (termed the prior probability), the term $P(data|x)$ is the probability that, *given event x occured*, the data would be observed, and the term $P(x|data)$ is the probability that the event x occurred given the data that was observed (termed the *posterior probability*). The denominator is a normalizing factor so that $P(data|x) \in [0, 1]\forall x$, which takes into account information known about possible alternative hypotheses.

This system can be adapted to pattern scoring purposes by defining each event x as the existence of a certain isotopic pattern or tag, represented by the intensity vector $\vec{a}$ and defining the $data$ is the observed intensity vector, $\vec{d}$. Equation 3.1 then becomes:

$$P(\vec{a} \mid \vec{d}) = \frac{P(\vec{d} \mid \vec{a})P(\vec{a})}{\sum_i P(\vec{d_i} \mid \vec{a_i})P(\vec{a_i})} \tag{3.2}$$

Prior probabilities for each tag can then be defined as the probability that any given peak set in our sample is due to the presence of an ion labeled with that tag with the isotopic envelope $\vec{a}$.

Generally speaking, it is difficult to define a probability function for the values $P(\vec{d}|\vec{a})$. In addition, it is not possible to account for every possible source of a given signal. As such, it is somewhat misleading to term the value $P(\vec{a}|\vec{d})$ as being a probability. Instead, we treat this value as a score that is roughly proportional to this probability. We do so at the loss of some mathematical rigor but no loss of generality with regards to pattern scoring. The more closely the pattern scoring method is associated with this probability, the more accurate our results using this method. This last point will influence our choice of pattern scoring functions.

Such a system is desirable as it makes it trivial to add another possible contaminating pattern—one simply has to create a new "tag" $\vec{a_i}$ and define an associated prior probability $P(\vec{a_i})$. This system also lends itself easily to adjustment, as prior probabilities can be determined empirically and are trivial to adjust.

3.6.2 Scoring individual patterns

From the properties we want the overall algorithm to have, we require certain characteristics of our pattern scoring system. We want the scoring algorithm to be independent of the absolute intensity of the data, we want it to be able to match imperfect data whenever possible, and we want it to be fast. Additionally, in order to use this score with the Bayesian approach defined in Equation 3.2, we want the score to be linearly proportional to the probability of that pattern (the value $P(\vec{a} \mid \vec{d}\,)$), or as close as possible to it. Some likely candidates for this function are outlined in Table 3-2, along with pros and cons of using each function.

The cosine cross-correlation function is often a first choice when testing for the overlap of two datasets, as defined by

$$\text{Cross Correlation } = \frac{\displaystyle\sum_k a_k d_k}{\left[\left(\displaystyle\sum_k a_k^2\right)\left(\displaystyle\sum_k d_k^2\right)\right]^{1/2}} \tag{3.3}$$

again where $a_i \in \vec{a}$ are the intensities of the peaks in the ideal (model) spectrum and $d_i \in \vec{d}$ are the corresponding peak intensities observed in the actual data. This function as some nice properties: it is easy to calculate and does not require that patterns be aligned in the intensity dimension before scoring can take place. Unfortunately, this ease comes at a cost, and it does not strongly discriminate between different patterns, and the value produced is certainly not proportional to the probability that we would like.

A second function that discriminates more strongly between patterns is the sum of squared difference between and the data and an ideal pattern that is aligned so that the intensities are as close as possible. In this function, the score is calculated as

$$\text{SSD} = \sum_k (d_k - za_k)^2 \tag{3.4}$$

where z is a constant scaling factor used to align the pattern with the data. Using this system, a lower score suggests a higher quality match. While this scheme is able to discriminate between two patterns more readily than than the cosine cross correlation coefficient can, it is plagued by other difficulties. First, the pattern must be aligned to the data (e.g., the optimal value of z must be determined) before scoring. More importantly, however, is that it is not generally proportional (or inversely proportional, for that matter) to the probability of the pattern being real, which would bring into question the use of Bayes' theorem.

If we assume that peak intensities are normally distributed around the true intensity with standard deviation σ, we can determine the probability for each peak independently. Using this method, we can determine the probability that a peak would be 'off' by a given amount using the complement of the Gaussian error function, defined by:[15]

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt \tag{3.5}$$

which has the property that the value $\text{erfc}\left(\frac{|i - i_0|}{\sigma\sqrt{2}}\right)$ is the probability that a value i was produced by a Gaussian distributed function centered at i_0 with standard deviation σ. As expected, this function has the appropriate bounds, $\text{erfc}(0) = 1$ and $\lim_{x\to\infty} \text{erfc}(x) = 0$. If the true value and the standard deviation of the distribution are known, then this function produces a real probability for each peak. From this, we can determine the total probability for a set of peaks by:

$$P(\vec{d} \mid \vec{a}\,) = \prod_k \text{erfc}\left(\frac{|d_k - za_k|}{\sigma\sqrt{2}}\right) \in [0, 1] \tag{3.6}$$

This function has many of the properties we desire: it represents a real probability with the appropriate bounds and can efficiently discriminate between different patterns. However, it fails the test of being able to match imperfect patterns, and also has the disadvantage that the extremely small values produced by erfc() at large intensity

Table 3-2: A comparison of four potential pattern scoring functions.

Scoring System	**Advantages**	**Disadvantages**
Cross Correlation	• Fast • Easy to compute • Does not require pattern intensity alignment • Intensity invariant	• Poor differentiation between patterns
Sum of squared difference	• Better differentiation between patterns • Simple to compute	• Requires intensity alignment • Dependent on pattern intensity • Not linearly proportional to probability
Error function	• Strong differentiation between patterns • Reflects a real probability distribution	• Slower • Computational problems - floating point errors can be significant • Requires intensity alignment
Modified error function	• Strong differentiation • Computationally reasonable • Can account for contaminating peaks	• Slowest • Requires intensity alignment

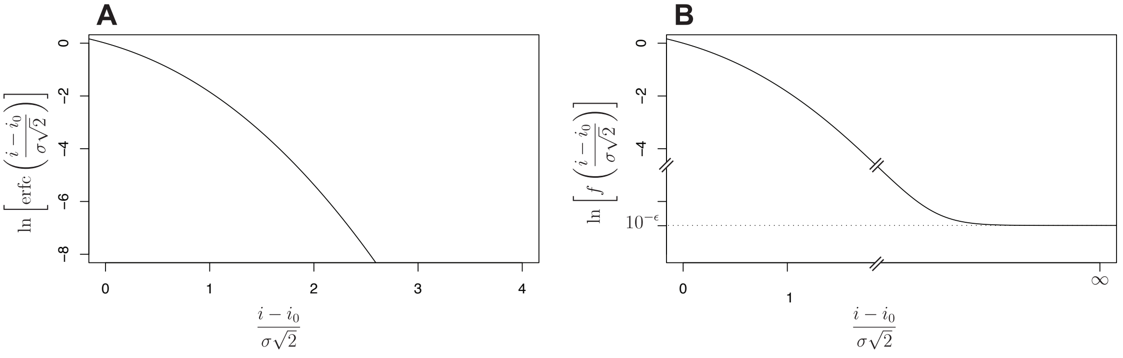


Figure 3-3: The behavior of a modified erfc function. (**A**) The natural logarithm of the error function compliment and (**B**) a modified version with an implemented minimum value $10^{-\epsilon}$. Due to precision limitations in floating point arithmetic, the standard $erfc$ function produces zero values (rather than infinitesimally small values) at large distances . The modified function eliminates this problem and introduces a tolerance for contaminating or obscured peaks.

differences leads to roundoff errors, in the extreme producing:

$$\prod_k \mathrm{erfc}\left(\frac{|d_k - za_k|}{\sigma\sqrt{2}}\right) = 0 \tag{3.7}$$

if the value $|d_k - za_k|$ is too large. The use of this function in its current form makes the matching algorithm fragile if there are significant errors in the data. However, this can be overcome by a simple modification to this function:

$$f(x) = (1 - 10^{-\epsilon})\mathrm{erfc}(x) + 10^{-\epsilon} \tag{3.8}$$

where ϵ is some constant. In non-mathematical terms, the value $10^{-\epsilon}$ represents some nonzero probability that a single peak is missing or contaminated in some way (*e.g.*, the peak from another ion happened to fall at the same m/z value, making the signal unexpectedly high). The behavior of this modification compared to that of the original function can be seen in Figure 3-3. Now, the product is:

$$P(\vec{d} \mid \vec{a}\,) = \prod_{k=1}^{N} f\left(\frac{|d_k - za_k|}{\sigma\sqrt{2}}\right) \in [10^{-N\epsilon}, 1] \tag{3.9}$$

with the nonzero lower bound $10^{-N\epsilon}$. By adding a nonzero lower bound to the function, roundoff errors are reduced to non-catastrophic levels. The minimum value requirement also makes the algorithm robust to missing peaks in the data—since missing peaks will produce a constant multiple in the pattern probability function, the score for every pattern will be affected in the same way, and the constant factor will be divided out when Bayes' theorem is applied. Consider a set of possible isotopic patterns, $\vec{a_k} \in \mathbf{A}$ where the last peak of the data is missing:

$$P(\vec{a_x} \mid \vec{d}\,) = \frac{P(\vec{a_x})10^{-\epsilon} \prod_{k=1}^{N-1} f\left(\frac{d_k - z_x a_{xk}}{\sigma\sqrt{2}}\right)}{\sum_{i=1}^{T} P(\vec{a_i})10^{-\epsilon} \prod_{k=1}^{N-1} f\left(\frac{d_k - z_i a_{ik}}{\sigma\sqrt{2}}\right)} \tag{3.10}$$

$$= \frac{P(\vec{a_x})10^{-\epsilon} \prod_{k=1}^{N-1} f\left(\frac{d_k - z_x a_{xk}}{\sigma\sqrt{2}}\right)}{10^{-\epsilon} \sum_{i=1}^{T} P(\vec{a_i}) \prod_{k=1}^{N-1} f\left(\frac{d_k - z_i a_{ik}}{\sigma\sqrt{2}}\right)}. \tag{3.11}$$

$$= \frac{P(\vec{a_x}) \prod_{k=1}^{N-1} f\left(\frac{d_k - z_x a_{xk}}{\sigma\sqrt{2}}\right)}{\sum_{i=1}^{T} P(\vec{a_i}) \prod_{k=1}^{N-1} f\left(\frac{d_k - z_i a_{ik}}{\sigma\sqrt{2}}\right)} \tag{3.12}$$

where the constant term $10^{-\epsilon}$ cancels out. Here, T is the number of tags being searched again, d_k is the data at index k, $a_{x,k}$ is the intensity of the k^{th} peak of pattern x, and z_x is a scaling constant for pattern x. This cancellation adds robustness against missing or contaminated peaks.[§]

At this point, there are still two problems with this function. Before a pattern can be scored, the optimal value of the scaling factor z must be determined for each pattern as will be discussed in Subsection 3.6.3. The other issue is that in its current form, this function is very much dependent on the absolute intensity of the data being observed due to the constant value of σ. However, this can be easily overcome by allowing σ itself to be a variable that is dependent on the intensity of the data

$$\sigma = m\langle J^2\rangle^{1/2} \tag{3.13}$$

[§] The same analysis can be done for any given missing or contaminating peak, so long as the peak is far enough away from the expected intensities of each pattern that the relation $f\left(\frac{I - I_o}{\sigma\sqrt{2}}\right) = 10^{-\epsilon}$ holds. In general, this will be true for any significant interference in the data, depending on the value of σ chosen for the analysis.

where m is now a constant parameter. Setting σ proportional to the RMS signal intensity removes the absolute intensity bias from the function and eliminates the need to determine the value of σ independently for each dataset. Appropriate values of m can be determined empically with little difficulty, with a value of $\sigma = 0.08$ producing good results for data obtained on a Thermo-Finnigan LTQ Orbitrap XL.

3.6.3 Aligning pattern intensities

Before the scoring function described above can be used, we must determine the optimal value for the scaling parameter z. The first step in achieving this is defining a best-fit function. One of the most straightforward methods is to minimize the sum of squared difference between the reference pattern $\vec{a}$ and the data, $\vec{d}$, given by Equation 3.4.[¶] The sum of squared difference is sensitive to small differences between the two patterns and is simple to calculate. A binary search on the parameter z can then be used to find the optimal value to within a predefined tolerance ϵ (Algorithm 3-2).

To make this even slightly more robust, we can ignore points in the data with zero intensity (i.e., missing peaks) when computing the sum of squared difference between the two patterns. A similar (though more elaborate) strategy could be devised for ignoring unusually intense people during the alignment, but in practice more elaborate approaches have proven to be unnecessary.

3.6.4 Adjusting patterns to account for peptide isotopic envelopes

The isotopic contributions from peptides themselves cannot be ignored in pattern matching. Thus, we to be able to update the patterns we are matching to based on the data we are interested in. If we can estimate the molecular weight of the peptide in question, we can predict how the pattern of the tag will change.

From the m/z separation the charge state can be determined. Working backwards, if we assume that the pattern we are examining was created by a peptide, we then know what the m/z value and charge state are. From this, if we make the additional assumption that all charges on the peptide are due to the gain of an H^+ ion, then the

[¶] We could also have tried to use the erfc() or (the modified version) we introduced in the previous section. Unfortunately, the same problems encountered above remain a problem for the erfc() function here. The non-zero lower bound that we introduced to make the modified version makes this not suitable as a 'goodness of fit' function, as two different values of $|z|$ can produce the same score.

```
$\mathsf{SSD}(\mathsf{J},\mathsf{I},\mathsf{z}) = \sum_k (d_k - z a_k)^2$
$\mathsf{z} = \frac{\langle \vec{d}\,^2 \rangle^{1/2}}{\langle \vec{a}\,^2 \rangle^{1/2}}$
step $= z/2$
direction $= 0$
$score$ = SSD($\vec{d},\ \vec{a},$ z)
while $step > \epsilon$ do
      score_up = SSD($\vec{d},\ \vec{a},$ z + step)
      score_down = SSD($\vec{d},\ \vec{a},$ z − step)
      if score_down < score
        z = z − step
        score = score_down
        if direction $\geqslant 0$
          step = step$/2$
        fi
        direction $= -1$
   else
        z = z + step
        score = score_up
        if direction $\leqslant 0$
          step = step$/2$
        fi
        direction $= 1$
      fi
od
```

Algorithm 3-2: An algorithm to minimize the sum of squared difference (SSD) between the pattern and the data to obtain the optimal value for the scaling factor z. The value of z is initialized to the ratio of the rms intensity of the data to the rms intensity of the pattern. A step factor of $z/2$ is then chosen as a starting point, and a binary search is performed on the value of z, halting when the step size is $\leqslant \epsilon$. This algorithm will produce the ideal value for $z \pm \epsilon$, where ϵ is a constant tolerance.

m/z value for the peptide with n excess protons (charge = n) is given by

$$m/z = \frac{MW + n}{n} \tag{3.14}$$

rearranging,

$$MW = \left(\frac{m}{z} - 1\right) * charge \tag{3.15}$$

From knowing the m/z value and the charge state, we therefore know the molecular weight of the peptide. Using this information, we can correct the isotopic distribution of any patterns of interest by assuming they are attached to a peptide of the determined molecular weight, allowing the algorithm to account for the natural abundance of biological elements as described in Chapter 4.

3.7 The identification of putative matches

From this point, isotopic pattern matching is a relatively simple endeavor. Starting with the point where related peak subsets have been obtained, all possible charge-state consistent peak subsets can be scored against a data-dependent pattern using the known charge state, m/z value, and the averagine model. For each relevant charge state, each reference pattern $\vec{a_k}$ is updated with the isotopic information predicted for the peptide itself. Then, each of these updated reference patterns, $\vec{a'_k}$ is scored against the set of peaks from the data, $\vec{d}$ using Equation 3.9 to obtain a list of scores, s_k. Finally, the scores are combined using the previously discussed Bayesian approach, leading to

$$P(\vec{a_x}|\vec{d}) = \frac{s_x P(\vec{a_x})}{\sum\limits_k s_k P(\vec{a_k})} \tag{3.16}$$

to obtain a final score for the putative match. Peak subsets that have a score exceeding a pre-determined threshold are then saved, along with pertinent data such as charge state and predicted parent ion molecular weight. The search then continues to the next scan of the LC-MS data file, until each scan has been analyzed. Putative matches are then subjected to further analysis to reduce the net false positive identification rate of the algorithm.

3.8 A graph theoretic framework can reduce false positive matches

3.8.1 Putative matches as nodes in a graph

Because we want to be able to match patterns in non-ideal data, searching for the pattern alone with no secondary validation or filtering can produce significant false positives due to low-intensity noise or a number of other factors. In an effort to reduce these numbers, we can use two features of LC-MS data to help refine our results. The reasoning is as follows: if a given match is really from a peptide, then it is likely that more of the same ions will be found in neighboring scans. In addition, real peptides are often detected in more than one charge state.

Both of these factors can be taken into account using a graph theoretic approach. Generically, a graph in this context is a structure comprised of nodes and edges, where edges are used to describe relationships between nodes, with various operations defined on the resulting structure[16–18]. Though there are a number of different types of graphs, differing in how edges can be defined, here we restrict ourselves to undirected graphs. Undirected graphs consist of nodes and a corresponding set of unweighted, undirected edges between those nodes. In the usage here, each putative match found by pattern searching alone can be treated as a node, and edges can then be drawn between two nodes if the patterns could have been produced by the same molecular species (see Algorithm 3-3).

As before, once the graph and edges have been constructed, the graph can then be deconstructed into disjoint sets of matches based on the edges defined. Each one of these sets has the property that all matches came from a molecular species of similar mass, and that each element in a set eluted close in time to at least one other element in the set. These disjoint sets can be treated as a single molecular species, and further filtering can be performed. The simplest filter that can be applied is simply to discard any nodes that have no neighbors, while more elaborate systems score sets based on the number of elements in the set, the number of unique charge states observed, and the quality of the individual patterns involved. A visual representation of this technique can be seen in Figure 3-4.

3.8.2 Implications on false positive rate

To analyze the ability of a graph construct to improve discrimination between true and false positive hits, a Monte Carlo simulation was performed. The same graph geometry used in the true searching was used, and is depicted in Figure 3-5. Nodes

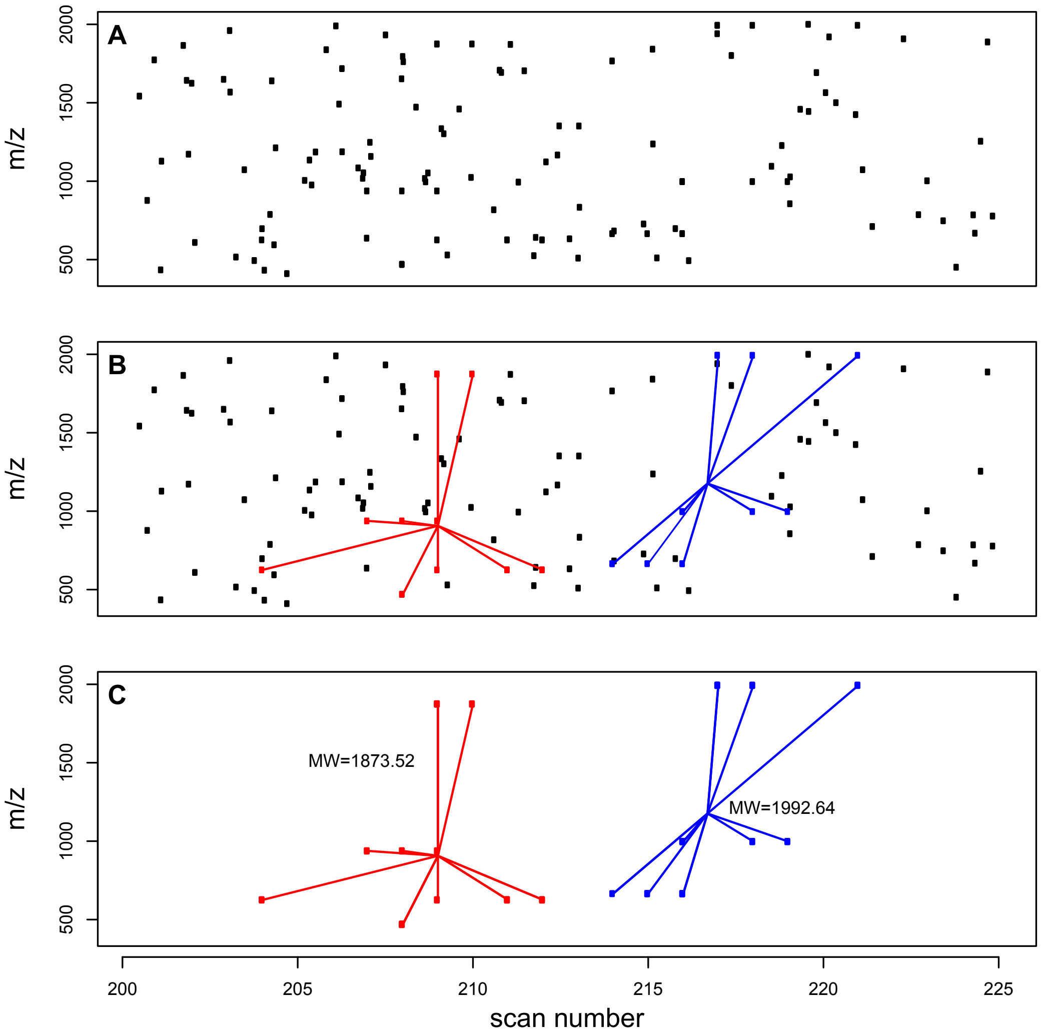


Figure 3-4: Data from the elution profile and additional charge states can be used to filter out false positives. (**A)** A profile of putative matches found based on pattern alone illustrates the case where a naïve filtering method produces a significant number of false positives . (**B)** If all putative matches are treated as nodes in a graph, edges can be drawn between nodes that (a) eluted close in time and (b) could have been produced by the same molecular species . (**C)** Finally, the graph can be decomposed into disjoint sets which can then be scored and subjected to further analysis .

were considered to be neighbors if they were separated by at most one scan number, *e.g.*, nodes in scan 0 and scan 2 are neighbors, but nodes in scan 0 and in scan 3 are not. Here, the charge state of each node can be ignored in determining if two nodes are neighbors as this information is encapsulated by the geometry of the graph.

Using the assumption that each node has an indepdent probability of being assigned a positive value, The Monte Carlo simulation was performed by assigning nodes a binary value of true or false ('matched' or 'not-matched', respectively) based

```
if $|match1.scan - match2.scan| > scan\_tolerance$
  return $false$
fi
if |match1.MW − match2.MW| $> \delta$m
  return $false$
fi
return $true$
```

Algorithm 3-3: An algorithm for determining whether or not two putative matches could have been produced by the same molecular species. Here, $match1$ and $match2$ are two putative matches, each with the attributes $scan$ (the scan number in which they were found) and MW (the weight of the molecular species that could have produced this pattern). Additionally, there are two parameters, $scan\_tolerance$ and δm. The former is the maximum number of scans two matches can be apart and still be considered related, while the second is the m/z tolerance of the searching. This function returns false if it is unlikely that the supplied nodes are related, true otherwise.

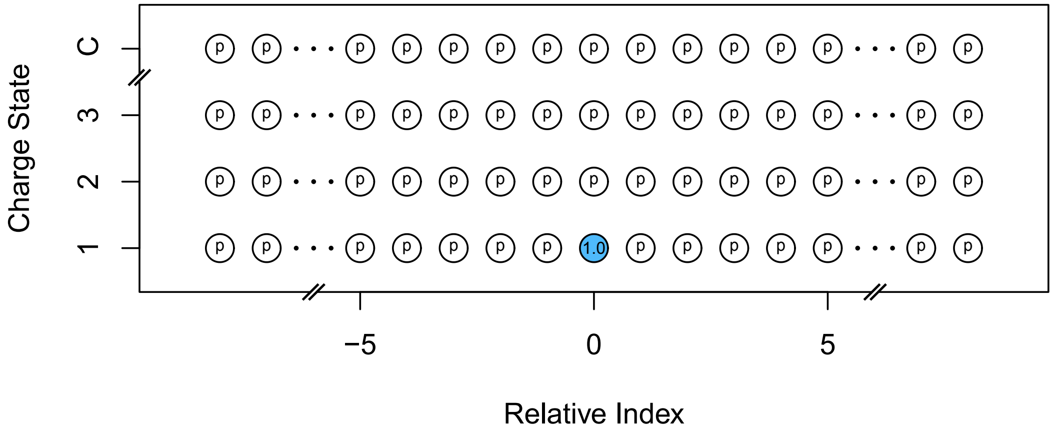


Figure 3-5: A schematic of the graph layout used to analyze the effect of the graph theoretic approach on detection rate. As in real data, nodes are arranged in a grid, where there are C nodes in the charge-state dimension, and a large number of scans are included in the temporal dimension. Using a Monte Carlo simulation, each node is randomly assigned as a positive or negative hit based on the probability p for neighboring nodes, or assigned as a positive hit with probability 1.0 for the reference node shown in blue.

on an assigned probability p, except for a reference node at scan 0, charge state +1 which was defined as true. Edges were then drawn between all neighboring nodes assigned a positive truth value, and the final graph was decomposed into disjoint sets. The set containing the reference node was then analyzed to determine the size of the set.

The simulation described was performed for values of p in the range [0,1], at

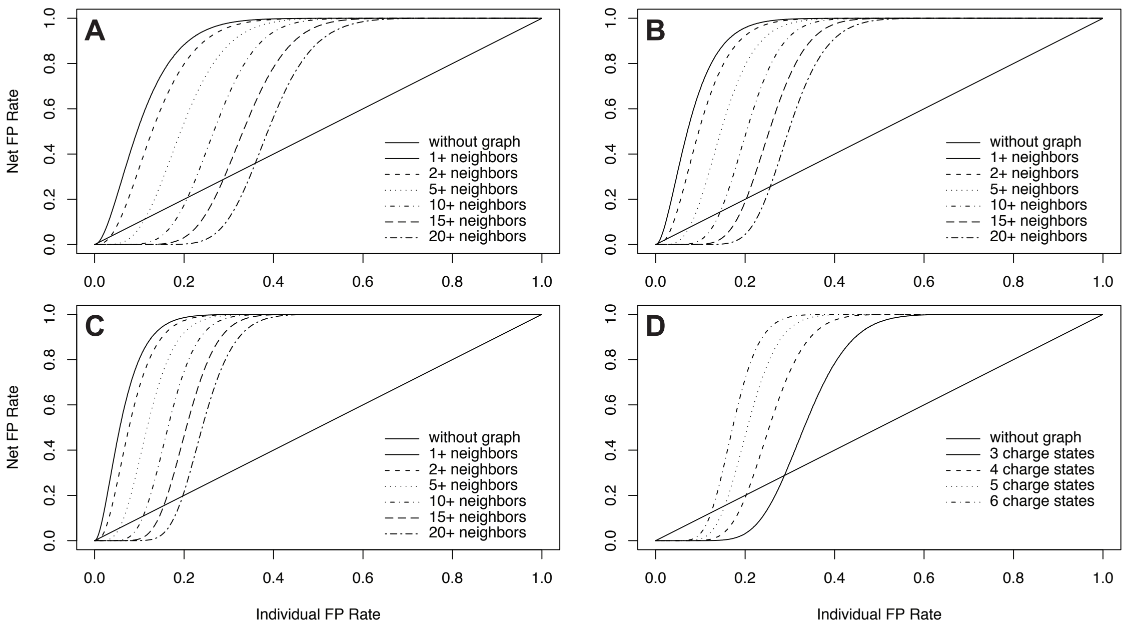


Figure 3-6: An analysis of the graph construct used in the pattern matching algorithm shows that this structure is able to amplify differences between the false positive and true positive rates in the naïve algorithm. The graph was analyzed by correlating peaks within two scans of a matched peak with (**A**) 3, (**B**) 4, and (**C**) 5 charge states with the indicated number of neighbors required in the graph before a match is considered valid. (**D**) A plot showing a comparison of graph performance with different numbers of charge states when 15+ nodes are required for a match to be validated .

intervals separated by 0.001 units, with 500,000 replicates at each value of p. The simulation was also performed over various values of C, the number of charge states included in the analysis. The results of this simulation are given in Figure 3-6.

These results shows that the graph construct employed here is able to strongly amplify differences in positive determination rate, leading to a higher confidence in true positives (the right-most portion of each plot shown in Figure 3-6) and a lower incidence in false positive rate (the left-most region of each plot in Figure 3-6).

3.9 From peaks to hits: the overall searching process

To detect isotopic patterns in a large LC-MS file, each individual scan is processed separately, where each scan is first broken down into subsets of potentially related isotopic peaks. These subsets are then further processed by analyzing all possible charge-state-consistent subsets and performing a data-dependent pattern search on each of the subsets. In this step, each subset that scores above a predetermined threshold is saved as a putative match, and is further analyzed after the enter LC-MS dataset has been searched. By searching the entire LC-MS dataset in such a manner, a list of all putative matches along with relevant information about their temporal and m/z coordinates is created.

After processing all scans, potential matches are filtered using the graph-theoretic construct described above. The resulting graph is then decomposed into disjoint sets, each of which is scored according to the number of nodes in the graph, the intensity of the data in each node, the number of charge states detected, and the quality of the patterns involved. Sets that score above a second pre-determined threshold can then be saved, and summary statistics (charge states found, time of elution, parent ion molecular weight, etc) can be output. This final filter acts as a strong filter for false positive identifications, and greatly improves the fidelity of the overall searching algorithm.

3.10 Algorithm performance

An important aspect of pattern searching in the context of the dibromide tag is absolute searching time. Using a modern desktop computer (3.66 GHz, 4 GB ram), an average LC-MS data file can be searched in less than two minutes using standard settings. This processing time suggests that it would be possible to perform the same isotopic searching on data in real time, where the same amount of data is collected over a 60-90 minute period.

The analysis of data generated by labeling BSA with a cysteine-alkylating dibromide tag (see Chatper 2) shows that this algorithm is able to detect a sufficiently unique pattern at low signal to noise ratios in extremely complex biological samples, while producing surprisingly few false positives. Furthermore, this algorithm has proven to be adaptable to the detection of other isotopic signatures, including the dichloro-, monobromo-, and tetrabromo- (doubly dibromide-tagged) motifs, and it is expected that searching for other isotopic patterns would be similarly facile.

References

[1] Dass, C. (2007) *Fundamentals of Contemporary Mass Spectrometry*. (Wiley).

[2] Weheofsky, M & Hoffmann, R. (2002) Automated deconvolution and deisotoping of electrospray mass spectra. *J. Mass. Spectrom.* **37**: 223–229.

[3] Mujezinovic, N et al. (2006) Cleaning of raw peptide MS/MS spectra: Improved protein identification following deconvolution of multiply charged peaks, isotope clusters, and removal of background noise. *Proteomics* **6**: 5117–5131.

[4] Senko, M. W, Beu, S. C, & McLafferty, F. W. (1995) Determination of monoisotopic masses and ion populations for large biomolecules from resolved isotopic distributions. *J. Am. Soc. Mass. Spectrom.* **6**: 229–233.

[5] Pedrioli, P et al. (2004) A common open representation of mass spectrometry data and its application to proteomics research. *Nat. Biotechnol.* **22**: 1459–1466.

[6] Lin, S. M, Zhu, L, Winter, A. Q, Sasinowki, M, & Kibbe, W. A. (2005) What is mzXML good for? *Expert Rev. Proteomics* **2**: 839–845.

[7] Deutsch, E. W. (2008) A single, unifying data format for mass spectrometer output. *Proteomics* **8**: 2776–2777.

[8] Du, P, Kibbe, W. A, & Lin, S. M. (2006) Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching. *Bioinformatics* **22**: 2059–2065.

[9] Coombes, K. R et al. (2005) Improved peak detection and quantification of mass spectrometry data acquired from surface-enhanced laser desorption and ionization by denoising spectra with the undecimated discrete wavelet transform. *Proteomics* **5**: 4107–4117.

[10] Hargrove, W. F & Rosenthal, D. (1981) Improvement of algorithm for peak detection in automatic gas chromatography-mass spectrometry. *Anal. Chem.* **53**: 538–539.

[11] Hasting, C. A, Norton, S. M, & Roy, S. (2002) New algorithms for processing and peak detection in liquid chromatography/-mass spectrometry data. *Rapid Commun. Mass Sectrom.* **16**: 462–467.

[12] Zhang, J, Gonzalez, E, Hestilow, T, Haskins, W, & Huang, Y. (2009) Review of peak detection algorithms in liquid-chromatography-mass spectrometry. *Current Genomics* **10**: 388–401.

[13] Howson, C & Urbach, P. (1990) *Scientific Reasoning: The Bayesian Approach*. (Open Court Publishing Company).

[14] Jaynes, E. T. (2003) *Probability Theory: The Logic of Science*. (Carbridge University Press).

[15] Abramowitz, M & Stegun, I. A. (1964) *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. (Dover, New York), ninth dover printing, tenth gpo printing edition.

[16] Balakrishnan, V. K. (1997) *Graph Theory*. (McGraw-Hill), 1 edition.

[17] Bollobas, B. (2002) *Modern Graph Theory*. (Springer), 1 edition.

[18] Gross, J. L & Yellen, J. (1998) *Graph Theory and Its Applications*. (CRC Press).

chapter four

Estimation of the elemental composition and isotopic distribution of peptides and glycopeptides

4.1 Overview

Central to the work presented in Chapters 2 and 3 is the idea that it is possible to predict the skewing effect imparted on a given isotopic distribution by the contribution of elements in the biological molecules, namely those shown in Table 2-1. While the contribution of these elements is negligible in small molecules, the effect becomes dramatic in large biomolecules such as peptides, as can be seen in Figure 3-1, and their effect on the isotopic distribution of a molecule of interest must be accounted for. In order to do so, a method of predicting the isotopic distribution of the contaminates alone with some degree of confidence is needed. Such a method, termed the "averagine" system, was proposed by Senko *et al.*[1], and was further elaborated by Valkenborg *et al.*[2] In this system, the average molecular formula per unit mass of peptide is used to predict the average molecular formula for a peptide of arbitrary mass. In this chapter, the averagine system is analyzed with regard to its ability to accurately predict the isotopic envelope of a large number of peptides, and is extended to applications in predicting molecular formulae for putative glycoproteins through the development of a "glycoaveragine" system.

With an estimate of the molecular formula for a peptide in hand, it is then possible the predict the isotopic distribution expected for that formula computationally. For the purpose of rapid pattern searching in real time, it is desirable for such a prediction algorithm to be extremely rapid, even at the cost of some accuracy. Such an algorithm is discussed in Section 4.4. Taken together, the ability to estimate an accurate molecular formula and rapidly convert that formula into an isotopic distribution will determine the limitations of the isotopic labeling and pattern searching technology.

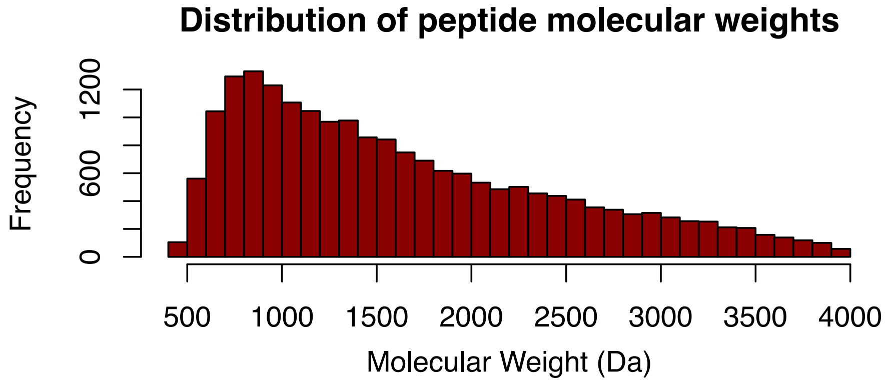


Figure 4-1: The distribution of molecular weights from a random sample of human tryptic peptides

4.2 Analysis of the averagine system

4.2.1 Construction of an averagine model

To get the best results for our specific application, an averagine model was constructed using a human proteome database downloaded from the ExPASy proteomics server[3], which was subsequently digested *in silico* to generate a list of tyrptic peptides[4]. From this list of all putative tryptic peptides in the human proteome, 20,000 unique peptides were randomly sampled, and their molecular weights and formulae were calculated (the molecular weight distribution of the sampled peptides can be seen in Figure 4-1. The only criteria for inclusion was that the peptides must contain at least six amino acid residues. No attempt was made to normalize the distribution of molecular weights was made as this distribution reflects the natural distribution of molecular weights of human tryptic peptides, and thus reflects the distribution expected in a natural sample. Based on these data, the average molecular formula per unit mass was calculated, and the results along with standard deviations of the estimates are given in Table 4-1. The convention of reporting the averagine system as element counts per unit mass was chosen as it facilitates rapid interpolation from molecular weight to a predicted molecular formula simply by multiplying the averagine constants by the molecular weight.

An interesting feature of the averagine system is that it is able to predict the number of hydrogen, carbon, nitrogen and oxygen atoms in a peptide to a high degree of accuracy (s.d. $<$ 15%), but is unable to reliably predict the number of sulfur atoms in peptides (s.d. $>$ 100%). This is due to the fact that sulfur is rare in peptides when compared to hydrogen, carbon, nitrogen, and oxygen. A weighted percent deviation

Table 4-1: The 'averagine' peptide. Numbers shown are the number of atoms of each element type per Da of peptide. [1] These numbers allow the elemental composition of a peptide to be easily predicted based solely on its molecular weight.

Element	Number per Da. Peptide[a] (± std. dev.)
Hydrogen	0.0710 ± 0.0047
Carbon	0.0435 ± 0.0052
Nitrogen	0.0126 ± 0.0018
Oxygen	0.0138 ± 0.0017
Sulfur[b]	0.00037 ± 0.00052

[a] Calculated on a random selection of 20000 human tryptic peptides with a sequence length of ≥ 6 amino acids

[b] Because of its relative rarity in peptides, there is a high error associated with predicting the number of sulfur atoms in relatively small peptides

calculated as

$$\frac{\sum_i \mu_i \left(\sigma_i/\mu_i\right)}{\sum_i \mu_i} = \frac{\sum_i \sigma_i}{\sum_i \mu_i} < 0.1 \tag{4.1}$$

Where σ_i is the standard deviation for the averagine estimate of element i, and μ_i is the averaging estimate for element i. The total deviation calculated in this manner is $< 10\%$, which is astonishingly accurate given the huge diversity of peptides found an organism's proteome.

4.2.2 The averagine model is a strong predictor of isotopic envelopes

The essential property of the averagine system for use in isotopic pattern matching is not in its ability to accurately predict molecular formulae, but in its ability to accurately predict the isotopic envelopes of peptides. Though these two abilities are no doubt related, there is reason to expect that they not be directly proportional*. To investigate the ability of the averaging model to predict peptide isotopic envelopes, the weighted root-mean-square deviation (RMSD) between the isotopic distribution

*For an extreme example, no peptide could exist such that its molecular formula was one standard deviation below the averagine prediction for all elements, as that peptide would not be of sufficient molecular weight. Peptides deficient in one element must have an overabundance of another to compensate for the lost molecular weight, and this overabundance would be expected to compensate the isotopic envelope.

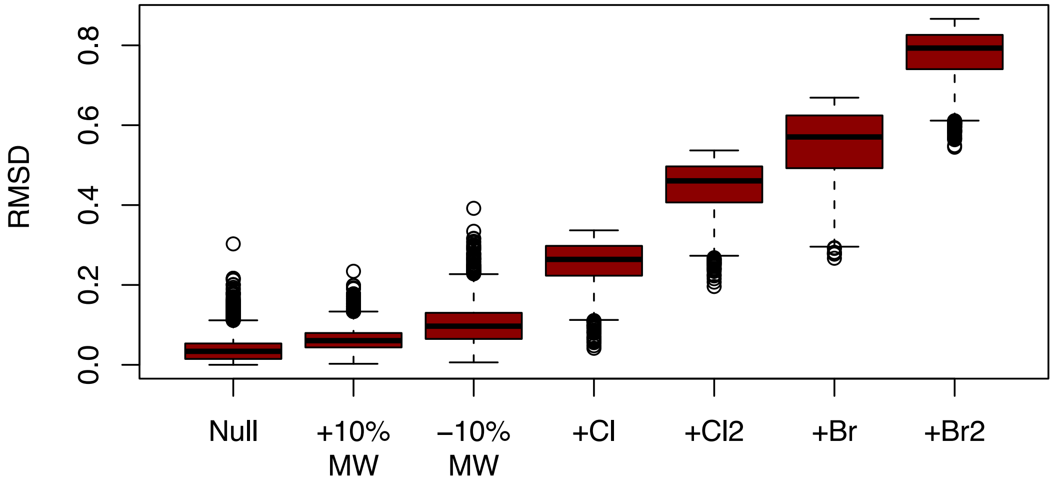


Figure 4-2: The averagine estimate accurately predicts peptide isotopic envelopes. A boxplot of the rms intensity differences between mass spectra predicted for 20,000 random human peptides based on actual elemental composition or based on the averagine estimate (leftmost), with an error of ±10% mass, or with the addition of the indicated tag.

predicted by the true molecular formula and the averagine peptide of the same molecular weight was computed. The root-mean-square deviation is a strong measure of of 'tightness of fit', where a value of 0 means a perfect fit and larger values mean a poor fit. This measure is commonly used in statistics[5], and has been used to *e.g.* analyze protein structure models[6,7], molecular dynamics[8], and molecular interfaces[9]. For each peak in the averagine-predicted isotopic envelope ($a_i \in \vec{a}$) and the envelope predicted from the actual molecular formula ($r_i \in \vec{r}$), the weighted RMSD is calculated as:

$$RMSD = \left[\frac{\sum_i (a_i - r_i)^2}{\sum_i r_i^2}\right]^{1/2} \tag{4.2}$$

To analyze the overall performance of the averagine system, a new sample of 20,000 tryptic peptides from an *in silico* digest of the human proteome was obtained (with MW distribution similar to that tabulated in Figure 4-1), and the RMSD as define above was calculated for each peptide. In addition, to analyze the error introduced by the averagine estimate as compared to the perturbation introduced by adding various halogenated tags, the isotopic envelopes for the true molecular formulae were compared to the averagine estimate at ±10% molecular weight, or with the addition of one- or two- chlorine or bromine atoms. The distribution of RMSD values obtained in this way is shown in Figure 4-2. From this, it is clear that the averagine estimate is a strong predictor of peptide isotopic envelopes, predicting distributions with a mean RMS Error of $< 4\%$. Furthermore, the averagine model shows a very gradual change is isotopic distributions with respect to peptide molecular weight, indicating that the system is tolerant to moderate errors in estimating peptide mass. In stark contrast, the addition of halogenated tags, in particular two bromine atoms, introduces a significant perturbation in the isotopic envelope. The analysis here agrees with experimental results in Chapter 2 in that the monochloride motif is not a sufficient perturbation to the peptide pattern as to be detectable, whereas the dichloride and monobromide are somewhat more distinct, and the dibromide is the largest perturbation. The superior performance of the monobromide compared to the dichloride tag here can be explained at least in part by the size of the pattern that can be used for pattern searching (4-5 peaks for the monobromide vs. 6-7 peaks for the dichloride), as more matchable peaks lead to more sensitivity in matching and a correspondingly lower false positive rate. A histogram of RMSD values for the averagine system with and without two bromine atoms can be seen in Figure 4-3, clearly showing the perturbing effect of two bromines on the average peptide isotopic envelope.

4.2.3 The averaging estimate holds accurate for peptides of moderate molecular weight

To analyze the performance of the averagine-based estimation of isotopic envelopes as a function of peptide molecular weight, a scatter plot of the 20,000 sampled peptides was created. For this analysis, the sampled peptides were partitioned into those with- and without sulfur-containing residues, as sulfur is likely to be the major source of error in isotopic distribution due to its relative scarcity and the relatively high (4%) abundance of its heavy isotopes. As can be seen in Figure 4-4, the averagine estimate for non-sulfur-containing peptides (*black*) is extremely accurate, and the distribution of RMSD values is tight even at molecular weights as high as 5,000 Da. In contrast, the sulfur-containing peptides (*red*) produce a broader range of RMSD values. The discontinuity in the plot apparent at $MW = 2.2$ kDa is due to the discrete nature of the averagine system, and corresponds to the molecular weight at which the model shifts from predicting zero to predicting one sulfur atom.

This analysis suggests that the averagine system will provide a reliable prediction for the vast majority of peptides at moderate molecular weights, but that deviations in the sulfur content of peptides at higher molecular weights will be a significant source of error. In the case that this is limiting, it should be possible to correct for this effect by growing cells on media depleted of ^{34}S.

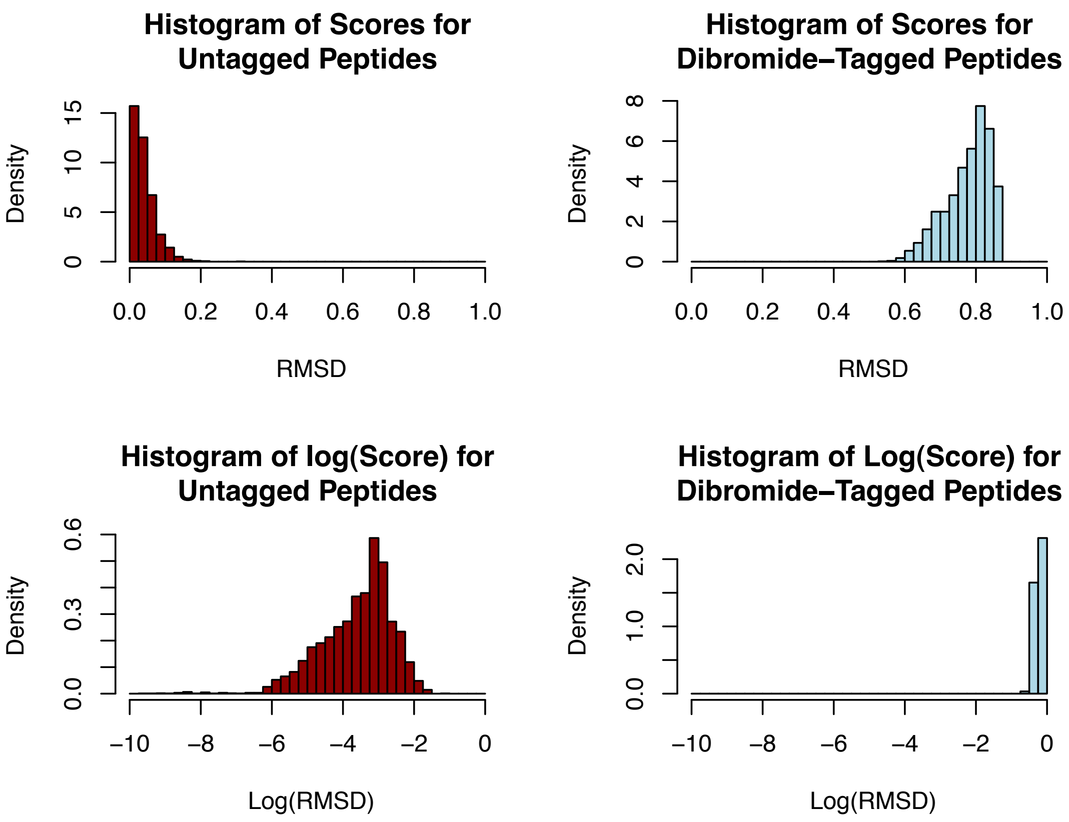


Figure 4-3: A histogram of rms intensity difference between mass spectra predicted for 20,000 random human peptides based on actual element composition or based on the averagine estimate (left) or the averagine estiate plus two bromine atoms (right).

4.3 Extension of the averagine system to glycoproteins

An important application of chemically directed proteomics is in the analysis of glycoproteins, including those where a large part of the mass is comes from glycosylation. In order to facilitate the extension of the isotopic labeling strategy to this set of biologically interesting proteins, it is necessary to analyze the performance of the averagine model in the prediction of isotopic envelopes for this heterogeneous collection of biomolecules. However, as glycosylation is heterogeneous, and in the case of mucin-type O-linked glycans, the modification's presence is difficult to predict from protein primary sequence alone, it is difficult to sample real glycopeptide molecular formulae in a reasonable, consistent, and biologically relevant manner. To skirt this difficulty, the present analysis focuses on determining the deviation between the standard averagine model and collection of modified averagine models consisting of predefined proportions of averagine peptide, hexose (Hex), and *N*-acetylhexosamines

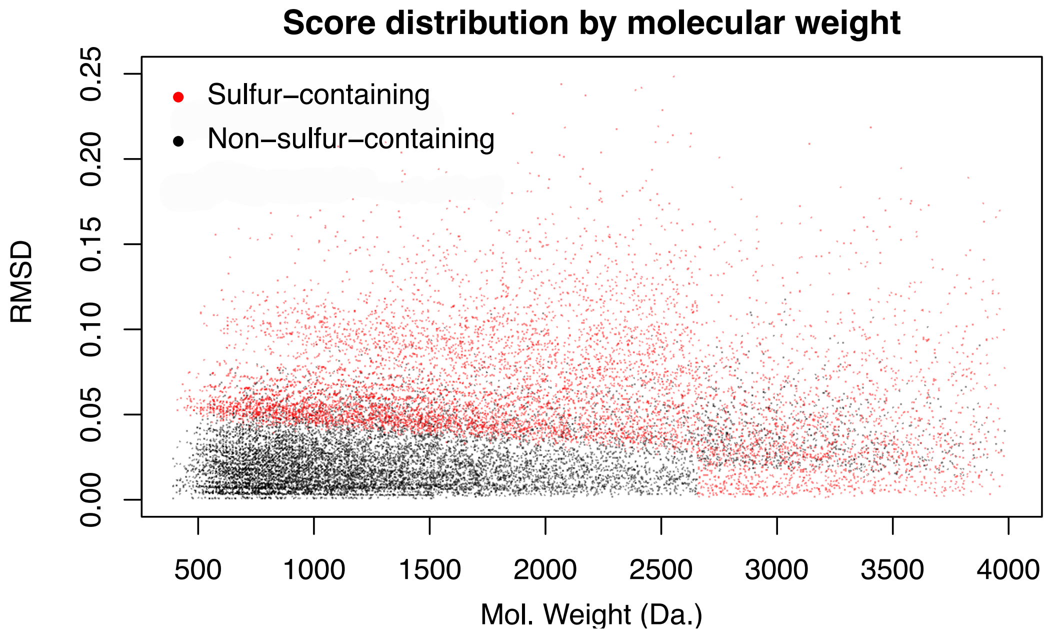


Figure 4-4: The score distribution of peptides by molecular weight. The distribution shows that sulfur is the primary source of error in the score. The sharp transition at molecular weight 2.6 kDa is due the averagine system predicting the presence of a single sulfur in the peptide. Below this molecular weight, no sulfur is predicted.

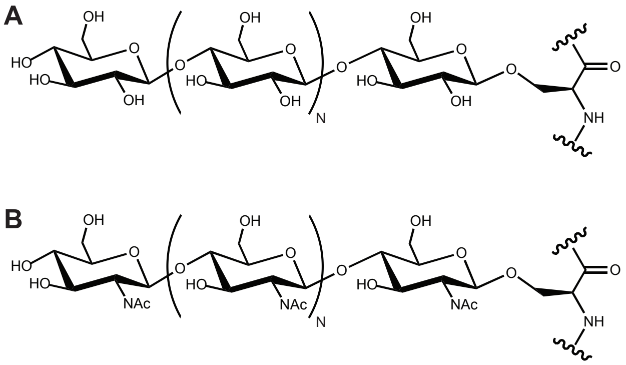


Figure 4-5: The structures of (**A**) Ser-*O*-polyhexose and (**B**) Ser-*O*-poly-*N*-acetylhexosamine used in the calculation of AveraHex and AveraHexNAc

(HexNAc). Due to the nature of glycosylation, it is possible to ignore the exact nature of glycan branching as the change in molecular weight in independent of the branch point or linkage. The first step in this direction is to determine the atom counts per unit molecular weight for large poly-hexoses and poly-*N*-acetylhexosamines, as shown in Figure 4-5. These counts will be called AveraHex and AveraHexNAc, respectively.

In determining the compositions of AveraHex and AveraHexNAc, the limit of an infinitely large glycan was considered to simplify the analysis. The net effect of this

Table 4-2: The composition of AveraHex

Element	Number per Da. Glycopeptide
Hydrogen	0.06171
Carbon	0.03702
Oxygen	0.03085

Table 4-3: The composition of AveraHexNAc

Element	Number per Da. Glycopeptide
Hydrogen	0.06401
Carbon	0.03939
Nitrogen	0.00492
Oxygen	0.02462

consideration is to ignore a single molecule of water for each site of glycosylation on the protein. The calculation of AveraHex was performed using limits of the form:

$$\text{AveraHex}_C = \lim_{N\to\infty} \frac{\text{\# of carbons in Hex}_N}{\text{mass of Hex}_N}$$

which leads to an average molecular formula of a singly-dehydrated hexose monomer, and the per unit mass composition shown in Table 4-2. Analogously, the composition of a large poly-HexNAc molecule is calculated using limits of the form:

$$\text{AveraHexNAc}_C = \lim_{N\to\infty} \frac{\text{\# of carbons in HexNAc}_N}{\text{mass of HexNAc}_N}$$

which similarly leads to an average molecular formula of a singly-dehydrated HexNAc monomer (the per unit mass composition shown in Table 4-3). Due to the manner in which AveraHex and AverHexNAc are calculated, there are no standard deviations associated with these estimates.

Using the compositions of averagine, AveraHex, and AveraHexNAc, it is then possible to analyze sets of mucin-like glycoproteins via linear combinations of these systems of the form:

$$\begin{aligned}\text{AveraMucin}_{p,q} &= p(\text{AveraHex}) + q(\text{AveraHexNAc}) \\ &\qquad + (1-p-q)\text{averagine}, \quad \begin{cases} p, q & \in [0,1]; \\ p+q & \leqslant 1; \end{cases}\end{aligned}$$

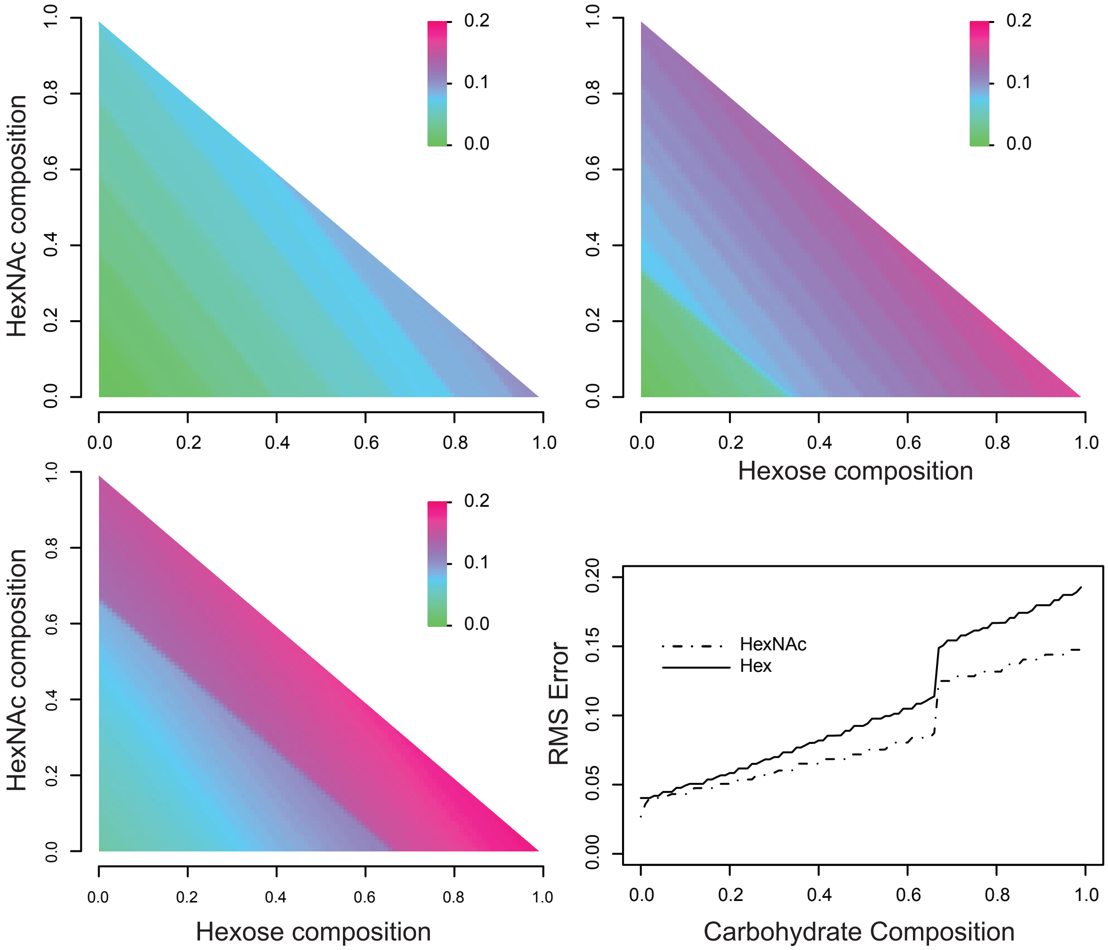


Figure 4-6: The performance of averagine in estimating isotopic envelopes of glycoproteins. The RMS difference between averagine and a modified averagine with the given glycan contribution on a 1000 Da (top left), 2000 Da (top right), and 4000 Da (bottom left) peptide. A cross-section of the plot at 4000 Da taken along the planes $x = 0$ (Hexose) and $y = 0$ (HexNAc).

The various linear combinations can then be analyzed by calculating the RMS Difference between the predicted averagine isotopic envelope and the AveraMucin predicted isotopic envelopes. The results of such an analysis are shown as a contour plot in Figure 4-6, analyzed at three molecular weights. This analysis shows that while there is a difference in prediction between the averagine estimate the AveraMucin estimates, these errors are still modest (maximum RMSD < 0.20) relative to the perturbation caused by the addition of two bromine atoms (mean RMSD > 0.80).

We are in no way required to disregard to possibility of carbohydrate contribution to the isotopic envelope. Depending on the sample, a hybrid approach may be appropriate where an AveraMucin intermediate between the most extreme experimental

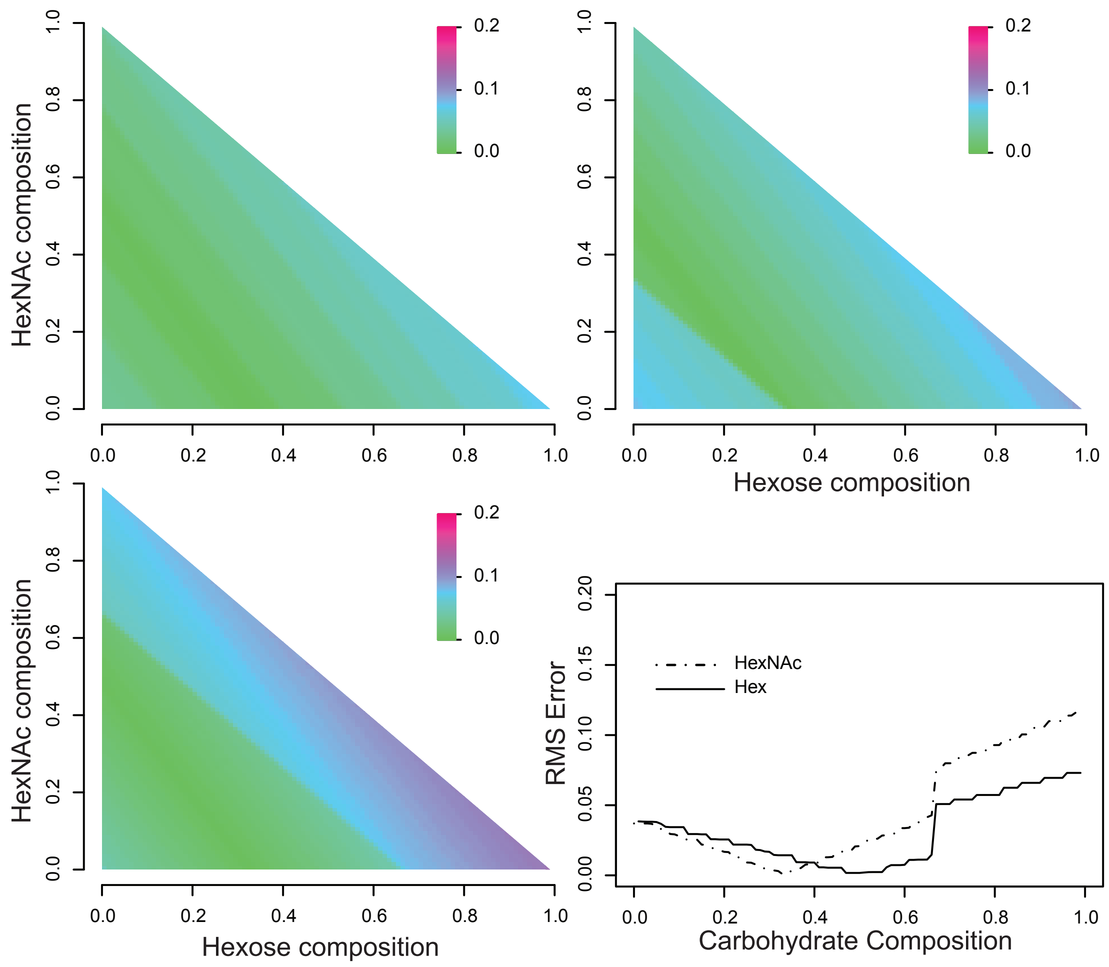


Figure 4-7: AveraMucin can be used to approximate isotopic envelopes of a wide range of glycopeptides. The same analysis as shown in Figure 4-6, using a modified 'averaglycogene' as a reference where 20% of the molecular weight is derived from hexose, 20% from HexNAc, and 60% from an averagine peptide. The analysis was performed at a number of molecular weights: 1000 Da (top left), 2000 Da (top right), and 4000 Da (bottom left). A cross-section of the plot at 4000 Da taken along the planes $x = 0$ (Hexose) and $y = 0$ (HexNAc). The use of a modified 'averagine' model reduces error at the extremes of the glycan content, and would be suitable for use in analyzing samples of unknown glycan content.

scenarios is chosen as a reference for all samples. For example, rather than using averagine, one could use AveraMucin$_{0.2,0.2}$ as a reference, thus reducing isotopic estimation errors at the extreme cases. An analysis of the performance of AveraMucin$_{0.2,0.2}$ is shown in Figure 4-7. From this, it can be seen that the maximum RMS Errors are greatly reduced using this model, with maximum RMSD values of approximately 10% in the mass ranges tested.

The analysis here suggests that chemically directed proteomics should be readily applicable to proteins containing a variety of post-translational modifications, including those where the PTM makes up a considerable percentage of the molecule's molecular weight. In many cases, the averagine estimate in isolation will be adequately robust to handle the presence of modifications, but in more complex scenarios it may be necessary to develop hybrid systems based on prior knowledge of the composition of the modification of interest.

4.4 A method for the rapid prediction of peptide isotopic distributions from molecular formulae

Essential to the use of the averagine system, and to isotopic pattern recognition as a whole, is the ability to rapidly calculate isotopic envelopes from a molecular formula. There are a number of methods by which to do this, for example, by utilizing Fourier transforms to compute high resolution envelopes for molecules that account for very slight mass differences between isotopes of a given element[10,11]. While these differences can be extremely important for some applicants, high-throughput peptide mass spectra are not typically of sufficient resolution to take advantage of the added detail of these computational techniques. For the purposes of analyzing peptide mass spectra, a reasonable approximation is that all isotopic peaks differ by exactly 1.0 mass units. This simplifies the calculation of isotopic envelopes immensely, allowing for greater speed while retaining a high degree of accuracy in the intensity dimension.

After assuming all isotopic peaks to be equally spaced, isotopic envelopes can be treated as vectors. Then, an operation $\circ$ can be defined that computes the isotopic envelope of a molecule given the isotopic envelopes of two simpler molecules. For example, the isotopic envelope of molecular bromine can be calculated from the natural abundance of ^{79}Br and ^{81}Br as:

$$\text{IE}(Br_2) = \text{IE}(Br) \circ \text{IE}(Br) \tag{4.3}$$

where $\text{IE}(X)$ denotes the set of intensities representing the isotopic envelope of a molecule X, normalized such that the summation of all peaks $a \in \text{IE}(X)$ satisfy the condition

$$\sum_{k=0}^{\infty} a_k = 1 \tag{4.4}$$

and where each $a_k \in \text{IE}(X)$ is the natural abundance of the molecule with a nominal

```
$c_k = 0 \; \forall \; c_k \; \in \; C, \quad \|C\| = \|A\| + \|B\| - 1$
for $i \in [0, \|A\|]$ do
    for $j \in [0, \|B\|]$ do
        $c_{i+j} = c_{i+j} + a_i b_j$
    od
od
return $C$
```

Algorithm 4-1: An algorithmic approach to calculating the convolution of two sets of isotopic intensities, $C = A \circ B$.

mass of k amu. As an example,

$$\begin{aligned} \mathrm{IE}(C) &= \{0,0,0,0,0,0,0,0,0,0,0,0,0.9893,0.0107,0,\cdots\} \\ \mathrm{IE}(Br) &= \{\underbrace{0,0,\cdots,0}_{79},0.5069,0,0.4931,0,\cdots\} \\ \mathrm{IE}(Br_2) &= \{\underbrace{0,0,\cdots,0}_{158},0.2569,0,0.4999,0,0.2431,\cdots\} \end{aligned} \tag{4.5}$$

represent the isotopic envelopes of carbon, atomic bromine, and molecular bromine, respectively. Using this notation, the convolution operator between two isotopic distributions can be defined by:

$$C = A \circ B \tag{4.6}$$

$$c_k = \sum_{i=0}^{k} a_i b_{k-i}, \quad k \in [0, \|A\| + \|B\| - 1], \quad \begin{cases} a_{j<0}, b_{j<0} & = 0; \\ a_{j>\|A\|} & = 0; \\ b_{j>\|B\|} & = 0; \end{cases} \tag{4.7}$$

a coputational implementation of which is given in Algorithm 4-1. This makes it possible to calculate the isotopic distribution for any molecule directly from the natural abundance of the isotopes of each element present in that molecule. However, for large biomolecules such as peptides, this would lead to the requirement of several thousand convolution operations required to predict the isotopic envelope of each molecule. To resolve this difficulty, it is important to note that due to the nature of the relevant chemistry, this operator is both commutative and associative:

$$\mathrm{IE}(CH_2Cl_2) = \mathrm{IE}(CH) \circ \mathrm{IE}(HCl_2) = \mathrm{IE}(Cl_2) \circ \mathrm{IE}(C) \circ \mathrm{IE}(H_2) \tag{4.8}$$

and so forth. For a more extreme (hypothetical) example,

$$\begin{aligned} \mathrm{IE}(C_{1108}) &= \underbrace{\mathrm{IE}(C) \circ \mathrm{IE}(C) \circ \cdots \circ \mathrm{IE}(C)}_{1108\ times} && (4.9) \\ &= \mathrm{IE}(C_{1024}) \circ \mathrm{IE}(C_{64}) \circ \mathrm{IE}(C_{16}) \circ \mathrm{IE}(C_4) && (4.10) \end{aligned}$$

which suggests the possibility of pre-calculating the isotopic distributions of large numbers of a given element. Using powers of two as such a basis set, we can define a recursive relationship that allows their rapid calculation:

$$\begin{aligned} \text{basis\_set}(Br, i) &= \mathrm{IE}(Br_{2^i}) \\ &= \mathrm{IE}(Br_{2^{i-1}}) \circ \mathrm{IE}(Br_{2^{i-1}}) \\ &= \text{basis\_set}(Br, i-1) \circ \text{basis\_set}(Br, i-1) \end{aligned} \qquad (4.11)$$

where basis_set$(X, 0)$ is defined to be the natural abundance of the isotopes of element X known from experimental or literature data. Then, the isotopic envelope of any molecule can be constructed from these basis sets, for example:

$$\begin{aligned} \mathrm{IE}(C_{1108}) = \text{basis\_set}(C, 10) \circ \text{basis\_set}(C, 6) \circ \\ \text{basis\_set}(C, 4) \circ \text{basis\_set}(C, 2) \end{aligned} \qquad (4.12)$$

or for C_5H_6:

$$\begin{aligned} \mathrm{IE}(C_6H_6) = \text{basis\_set}(C, 2) \circ \text{basis\_set}(C, 1) \circ \\ \text{basis\_set}(H, 2) \circ \text{basis\_set}(H, 1) \end{aligned} \qquad (4.13)$$

which allows a base set of isotopic distributions to be calculated once at a program's start, after which the isotopic envelopes of any molecule containing N atoms of a given element can be calculated through a at most $\lceil \log_2(N) + 1 \rceil$ convolution operations. The system as proposed still has two has two significant drawbacks: the basis sets grow in size exponentially, and Algorithm 4-1 is $O(n^2)$ in the size of the isotopic envelopes used; the former leads to challenges in storing the necessary basis sets in memory, while the latter suggests that this algorithm is intractable for large molecules. Both problems can be solved by an adjustment in the definition of the isotopic envelopes. Since the isotopic envelopes as defined in Equation 4.5 are sparse[†], the vast majority of the operations in calculating the convolution will contain a multiplication by zero. Redefining the reference frame so that the 0^{th} entry refers to the monoisotopic peak,

[†] *i.e.,* the majority of entries are zero values

the data in Equation 4.5 becomes:

$$\begin{aligned} \text{IE}(C) &= \{0.9893, 0.0107\} \\ \text{IE}(Br) &= \{0.5069, 0, 0.4931\} \\ \text{IE}(Br_2) &= \{0.2569, 0, 0.4999, 0, 0.2431\} \end{aligned} \tag{4.14}$$

solves both problems, with the only added requirement being that the monoisotopic mass of the molecule must be calculated independently from the exact monoisotopic masses of the contributing elements. An additional optimization includes trimming high molecular weight isotopic envelopes to remove very low intensity trailing values, *e.g.*, those contributing $< 0.01\%$ to the total distribution. In this case, most biological elements produce convoluted envelopes of roughly constant size, reducing the complexity of Algorithm 4-1 to $O(1)$, and drastically reducing the memory consumption of the basis sets.

Finally, the upper bound on number of convolutions required to calculate isotopic envelope of a molecule of the form $e^{(0)}_{n_0} e^{(1)}_{n_1} \cdots e^{(k)}_{n_k}$, where n_i is the number of atoms of element $e^{(i)}$ in the molecule, is given by

$$\sum_k \lceil \log_2(n_k) + 1 \rceil \tag{4.15}$$

each of which can be computed in linear time. The lower bound of the computational complexity is met when the numbers of each element, n_k, are powers of two. Then, the convolution requires one operation for each element considered, and the lower bound is given by

$$\sum_k 1 \tag{4.16}$$

and the average case is given by

$$\sum_k \left\lceil \left(\frac{\log_2(n_k)}{2} \right) + 1 \right\rceil \tag{4.17}$$

For example, applying these bounds to the analysis of the isotopic envelope of the BSA peptide MKWVTFISLLLLFSSAYSRGVFRRDTHK ($C_{157}H_{244}N_{42}O_{38}S_1$, MW=3.3 kDa), the estimated *upper bound* is 33 convolution operations, the *lower bound* is 5 convolutions, and the *average* complexity is 19 convolutions. In this particular case, the isotopic envelope can be calculated in 17 convolutions, slightly less than the expected average case.

4.5 Conclusions

The averagine system coupled with a system to rapidly calculate isotopic envelops of large molecules allows for the calculation of large numbers of these distributions in a very short period of time. Coupled with the searching algorithm described in Chapter 3, this allows for real-time searching of an isotopic pattern of interest while correcting for the isotopic contributions of other elements present. Furthermore, the fidelity of the averagine system shows that this approximation should be valid within searching tolerances for tryptic peptides of moderately large molecular weight, for example, up to 5000 Da. The extension of the traditional averagine system to account for the possible presence of large carbohydrate structures in Section 4.3 demonstrates that this approximation will also hold for such structures, and that a large range of protein co- and post-translational modifications can be handled using the averagine system with minimal modifications.

The primary limitation envisioned for the averagine system is in the analysis of proteins or modifications containing a large number of sulfur atoms. This is due to the large error in the estimation of sulfur by averagine coupled with the relatively high abundance of ^{34}S. The unpredictability of sulfur content is likely to be the primary limitation when using the averagine system to predict isotopic envelopes for large peptides and small proteins. Likewise, protein modifications containing large numbers of sulfur atoms, such as protein sulfation and the presence of sulfated glycans, will likely prove difficult to analyze. A possible solution to this difficulty would be to grow cells on media depleted of ^{34}S, in which case the error associated with sulfur prediction is inconsequential as it would no longer perturb the molecule's isotopic envelope.

C++ code for the isotopic envelope prediction algorithm presented in this chapter can be found in Appendix B.5.

References

[1] Senko, M. W, Beu, S. C, & McLafferty, F. W. (1995) Determination of monoisotopic masses and ion populations for large biomolecules from resolved isotopic distributions. *J. Am. Soc. Mass Spectrom.* **6**: 229–233.

[2] Valkenborg, D, Jansen, I, & Burzykowski, T. (2008) A model-based method for the prediction of the isotopic distribution of peptides. *J. Am. Soc. Mass Spectrom.* **19**: 703–712.

[3] Gasteiger, E et al. (2003) ExPASy: the proteomics server for in-depth protein knowledge and analysis. *Nucleic Acids Res.* **31**: 3784–3788.

[4] Cagney, G, Amiri, S, Premawaradena, T, Lindo, M, & Emili, A. (2003) In silico proteome analysis to facilitate proteomics experiments using mass spectrometry. *Proteome Science* **1**: 5–20.

[5] Jaynes, E. T. (2003) *Probability Theory: The Logic of Science*. (Carbridge University Press).

[6] Cohen, F. E & Sternberg, M. J. E. (1980) On the prediction of protein structure: the significance of the root-mean-square deviation. *J. Mol. Biol.* **138**: 321–333.

[7] Maiorov, V. N & Crippen, G. M. (1994) Significance of root-mean-square deviation in comparing three-dimensional structures of globular proteins. *J. Mol. Biol.* **234**: 625–634.

[8] Dixit, S. B, Ponomarev, S. Y, & Beveridge, D. L. (2006) Root mean square deviation probability analysis of molecular dynamics trajectories on dna. *J. Chem. Inf. Model.* **46**: 1084–1093.

[9] Bacon, D. J & Moult, J. (1992) Docking by least-squares fitting of molecular surface patterns. *J. Mol. Biol.* **225**: 849–858.

[10] Rockwood, A. L, Van Orden, S. L, & Smith, R. D. (1995) Rapid calculation of isotope distributions. *Anal. Chem.* **67**: 2699–2704.

[11] Rockwood, A. L & Van Orden, S. L. (1996) Ultrahigh-speed calculation of isotope distributions. *Anal. Chem.* **68**: 2027–2030.

chapter five

Targeting N-glycoproteins for chemically directed proteomics using a isotopic mixture (IsoMix) of a natural sugar

joint work with Mark A. Breidenbach and
Krishnan K. Palaniappan

5.1 Introduction

Protein N-linked glycosylation, a post-translational modification in which a glycan structure is linked to protein asparagine residues through a β-glycosylamide linkage, is an essential component of eukaryotic cell membranes[1–3]. N-glycosylation has been shown to be essential to a number of cellular functions, including cell surface organization[4], receptor recognition and function[5–7], cancer progression[8,9], and protein quality control in the endoplasmic reticulum[10–12]. Like O-linked glycans, the study of N-linked glycosylation is complicated by the size and heterogeneity of the carbohydrate structures. There are, however, important differences between N- and O-linked glycosylation. In contast to O-linked glycosylation in which no well-defined consensus sequence exists, N-linked glycosylation occurs primarily at sites with the primary sequence NXS/T, where X is any amino acid other than proline. While this greatly simplifies the prediction of N-glycosylation sites, protein secondary structure has been shown to play an important role in determining whether a given site is occupied[13,14], and there is recent evidence to suggest that N-linked glycosylation need not strictly follow this consensus sequence[15].

Rather than being completely heterogeneous, all N-glycans share a conserved core $Man_3GlcNAc_2$ pentasaccharide core[1]. The homogeneity of the N-linked core is complemented by the existence of endoglycosidases such as Peptide: N-Glycosidase F (PNGase F; removes the entire glycan) and endoglycosidase H (Endo H; cleaves after the the most proximal GlcNAc) which can be utilized to create a homogeneous population of glycan structures[16,17]. In particular, the use of EndoH can be used to create a glycoprotein consisting solely of an N-linked GlcNAc moiety, making the sole GlcNAc donor, UDP-GlcNAc, an attractive target for metabolic engineering.

Even with the availability of improve analytical techniques, the study of N-glycans has proven to be fraught with challenges. This is due primarily to the huge range in protein concentration and sub-stoichiometric occupancy of N-glycan sites, both of which the IsoStamp technology presented in Chapter 2 can be used to address. One strategy for incorporating and isotopic tag into sites of protein N-glycosylation would be through the metabolic incorporation of unnatural sugars functionalized with chemical handles and the use of a secondary reactive tag.

Recently, Breidenbach *et al.* demonstrated that it was indeed possible to incorporate the unnatural sugar GlcNAz into yeast N-glycans using a UDP-GlcNAc auxotrophic model[18]. Labeled glycans could then be captured with an azide-reactive dibromide tag in a method analogous to that of Hart and coworkers[19]. However, this methodology still requires the use of a secondary chemical labeling step, increasing the amount of sample handling and potentially decreasing the overall efficiency of sample labeling and increases the chances of sample contamination. Furthermore, UDP-GlcNAc auxotrophic yeast display a minor growth phenotype when grown with GlcNAz as the only GlcNAc source, suggesting that any N-glycosylation observed may not be physiological under these conditions.

An alternative strategy would be to use metabolic engineering to directly incorporate a dibromide-like isotopic pattern directly into a biomolecule of interest, forgoing the use of a secondary labeling step altogether. A simplistic methodology would be to incorporate an unnatural sugar containing two bromines directly into the cell, thus eliminating the necessity of a secondary labeling step. Such a method would still have the pitfall of using an unnatural substrate for the biosynthetic machinery, which may lead to lowered sensitivity sensitivity and may result in off-target hits. To eliminate these problems, a further simplification can be made: rather than include two bromines, the same isotopic pattern can be incorporated directly into the sugar moiety itself. To accomplish this, a mixture of isotopes of GlcNAc with the right molar ratios can be fed to cells, imparting a unique isotopic distribution to all proteins containing an *O*-GlcNAc modification. The position of the isotopic labels was chosen for convenience, as 1-^{13}C,^{15}N-glucosamine is commercially available, as is $^{13}C_2$-acetate, allowing the facile synthesis of both isotopically labeled GlcNAc molecules (see Section 5.4).

For the purposes of this chapter, the 1 : 2 : 1 molar mixture of the isotopes of GlcNAc will be referred to as IsoMix GlcNAc. By growing cells in media where IsoMix

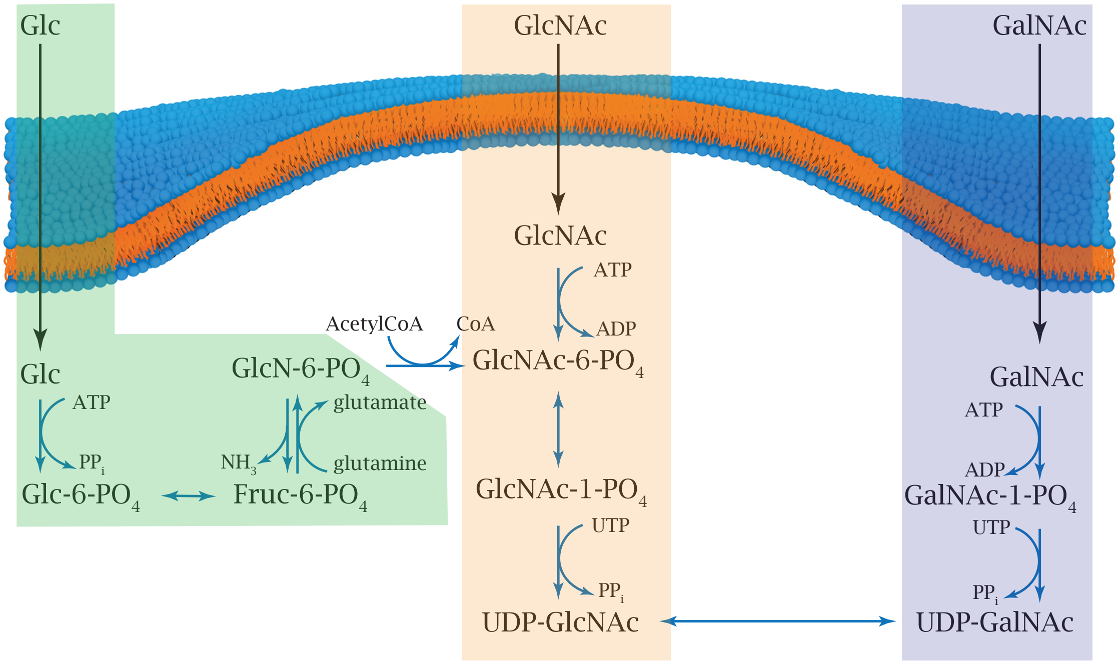


Figure 5-1: The biosynthetic pathways leading to UDP-GlcNAc in mammalian cells. The hexosamine bisynthetic pathway (*green*) converts cellular glucose to GlcNAc-6-phosphate, while the GlcNAc salvage pathway (*orange*) converts extracellular GlcNAc directly to GlcNAc-6-phosphate. Alternatively, the GalNAc salvage pathway (*blue*) converts extracellular GalNAc to UDP-GalNAc, which can then be epimerized at the 4 position by UDP-GalNAc/GlcNAc-4-epimerase to form UDP-GlcNAc.

GlcNAc is the only source of GlcNAc, all proteins containing this modification will incorporate the same isotopic pattern. Proteins can then be analyzed by LC-MS, and isotopically labeled species can then be determined using the algorithm described in Chapter 3.

A major hurdle in the described methodology is the presence of multiple metabolic pathways that feed into the UDP-GlcNAc pool (Figure 5-1). In particular, the majority of the UDP-GlcNAc pool in mammalian cells does not come from the GlcNAc salvage pathway exploited by most metabolic engineering techniques, but rather through the glucosamine/GFAT pathway[20]. This leads to two problems. First, that the desired isotopic pattern would need to be fed through more than one pathway or else risk dilution by competing pathways. Equally disastrous is that due to the reversibility of many of the steps, the incorporated isotopic pattern may end up in a multitude of other, non-GlcNAc containing biomolecules, rendering the labeling impossible to assign.

One solution to these problems is to use cell lines that lack the competing pathways. The BY4743-Δgna1 yeast line is one such system. In this yeast strain, the

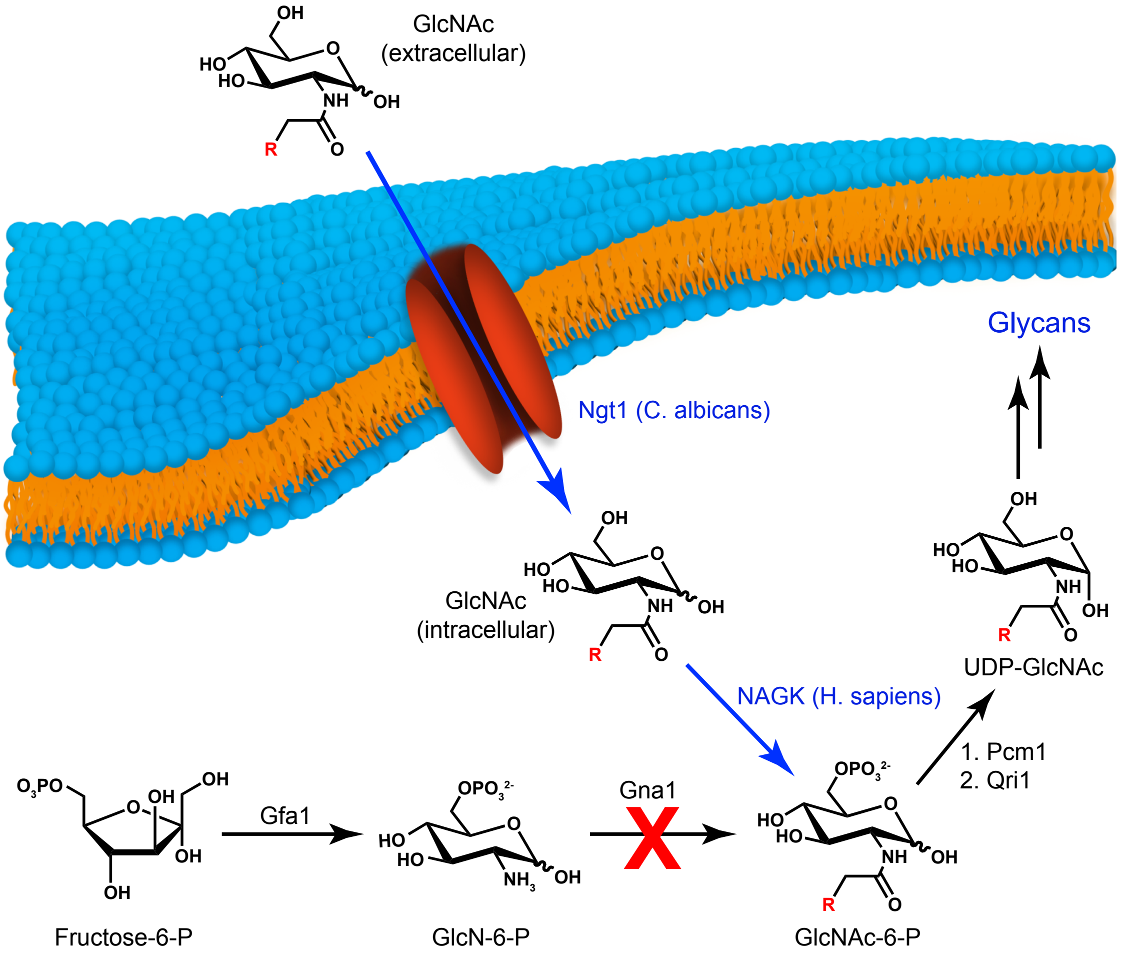


Figure 5-2: Δgna1 yeast rely entirely on extracellular GlcNAc. This strategy for bypassing *de novo* UDP-GlcNAc biosynthesis (*black arrows*) is shown. An exogenous salvage pathway (*blue arrows*) allows extracellular GlcNAc or analogs to be internalized by the transporter Ngt1 from *C. albicans*. The intracellular GlcNAc (or analog) is phosphorylated the at the 6 position via the activity of human GlcNAc kinase, NAGK[21]. The 6-phosphorylated product is subsequently converted into an activated nucleotide-sugar via the mutase and pyrophosphorylase activities of Pcm1 and Qri1 respectively. Figure adapted from Breidenbach *et al.*[18].

natural UDP-GlcNAc biosynthetic pathway has been knocked out by deletion of the transaminase Gna1, and a salvage pathway has been engineered in through the introduction of the transporter Ngt1 (*C. albicans*) and the human GlcNAc kinase NAGK (Figure 5-2). In the presence of isotopically labeled sugars, Δgna1 yeast show 100% incorporation of the labeled sugar (Figure 5-3)[18]. Employing this strain of yeast, cells can be grown in media containing the IsoMix, which will then be taken up by the GlcNAc transporter and will then enter the GlcNAc salvage pathway, where it is ultimately converted to the OGT substrate UDP-GlcNAc.

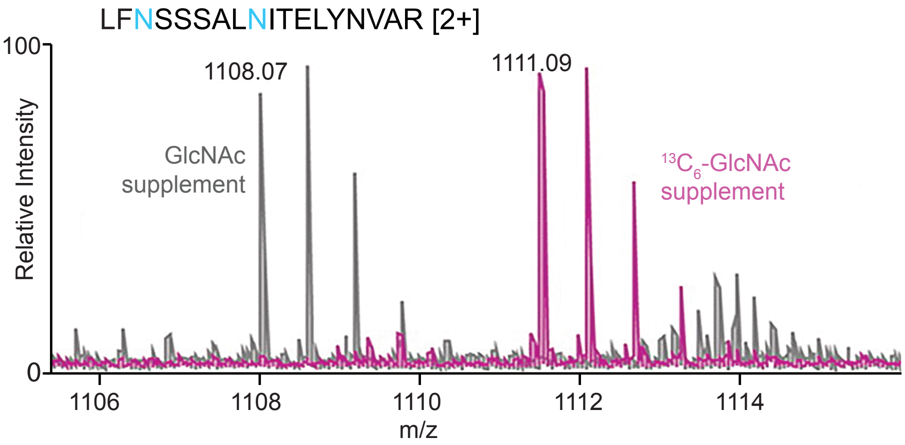


Figure 5-3: Δgna1 yeast show 100% incorporation of isotopically labeled GlcNAc. The protein Ygp1 was expressed in culture medium supplemented with either GlcNAc (*grey*) or $^{13}C_6$-GlcNAc. Ygp1 was treated with EndoH, trypsinized, and subjected to ESI-FTICR MS analysis. Masses for a representative glycopeptide, spanning Leu98–Arg115 and glycosylated at only one of the two potentital sites (indicated in *teal*) are shown. Relative intensities of the GlcNAc- and $^{13}C_6$-GlcNAc-modified peptides have been normalized to each other. Figure adapted from Breidenbach *et al.*[18]

In this chapter we present the use of the Δgna1 UDP-GlcNAc auxotrophic yeast in combination with a mixture isotopically labeled GlcNAc molecules to facilitate the chemically directed proteomic analysis of N-glycoproteins in *Saccharomyces cerevisiae*. To accomplish this, yeast are grown on media where the only source of GlcNAc is the isotopically labeled mixture, and their secreted or membrane proteins are isolated. We show that upon treatment with EndoH and analysis by LC-MS, glycopeptides display the characteristic 1 : 2 : 1 isotopic pattern, which can be used to generated an inclusion list of labeled species. Upon detection of labeled species, peptides are subjected to analysis by LC-MS/MS using an inclusion list, and resulting tandem spectra are compared against a yeast protein database using the SEQUEST algorithm to identify peptides along with sites of N-glycosylation.

5.2 Results

5.2.1 Isotopes of GlcNAc can be combined to recreate the dibromide motif

To recreate the isotopic pattern produced by a dibromide tag, three isotopically labeled GlcNAc derivatives separated by two mass unites each were required. Unlabeled GlcNAc, 1',2'-$^{13}C_2$-GlcNAc, and 1,1',2'-$^{13}C_3$,^{15}N-GlcNAc were chosen for these derivatives due to commercial availability and cost of reagents, where the sec-

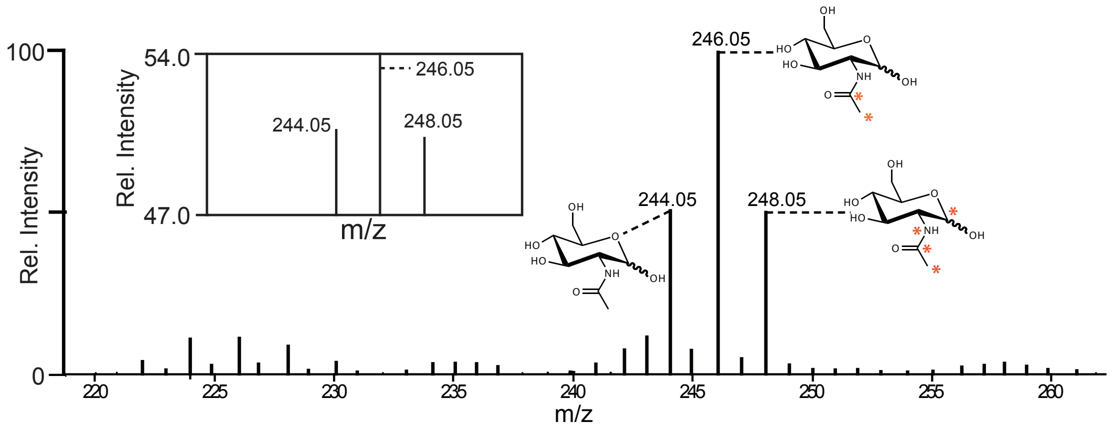


Figure 5-4: The mass spectrum of IsoMix GlcNAc shows a faithful reproduction of the 1 : 2 : 1 dibromide pattern. A blow up of the spectrum (insert) shows that the IsoMix pattern re-creates the dibromide pattern to within 1% accuracy. Data were obtained on a ThermoFisher LTQ in zoom scan mode, with 20 scans averaged to produce the final result. In these spectra, GlcNAc is ionized as the sodium salt, leading to the 23 amu mass increase. * indicates the presence of either ^{13}C or ^{15}N.

ond two derivatives were synthesized from glucosamine hydrochloride or 1-^{13}C,^{15}N-glucosamine hydrochloride and $^{13}C_2$ acetate, respectively. Stock solutions of each isotope were made, and combined in a 1 : 2 : 1 molar ratio to recreate the dibromide pattern (Figure 5-4).

5.2.2 The isotopic signature of IsoMix GlcNAc is retained in glycopeptides

The essential part of the isotopic labeling strategy is that the isotopic signature be recognizable on the target biomolecule, in this case on N-linked glycoproteins. To test this, Δgna1 yeast were grown in media containing 50 mM isomix GlcNAc, and the His-tagged protein YGP1 was purified from the media. The purified protein was then subjected to digestion with EndoH to remove all but the core GlcNAc moiety, followed by proteolytic digestion with trypsin followed by LC-MS analysis. The resulting data were then searched for peptides containing known sites of N-glycosylation to determine if the isotopic signature is retained in the glycoprotein. A representative MS spectrum shown in Figure 5-5 clearly shows a change in isotopic distribution upon glycosylation. Furthermore, the isotopic pattern searching software was able to recognize the resultant pattern, confirming that the pattern was adequately reproduced.

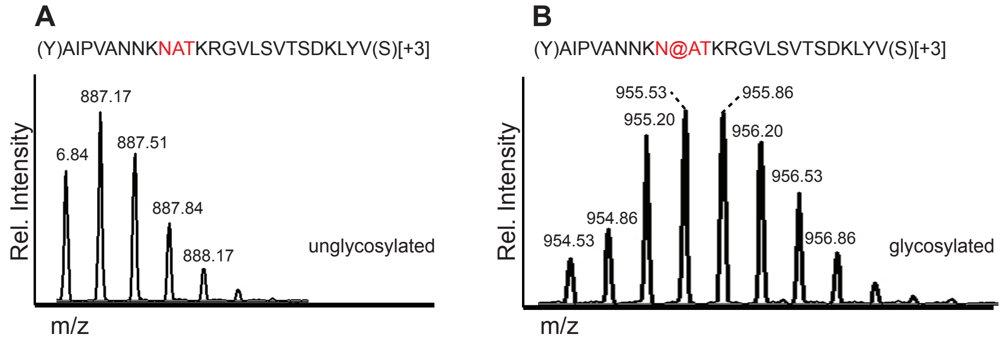


Figure 5-5: The isotopic signature of the IsoMix GlcNAc is retained in glycosylated peptides. The MS of the YGP1 peptide AIPVANNKNATKRGVLSVTSDKYLV (**A**) without and (**B**) with a GlcNAc residue at the indicated position (@ = N-GlcNAc). The modified peptide was confirmed by MS^2 (CID, xcorr = 1.68, Δcn = 0.36, 15 of 34 fragment ions assigned)

5.2.3 Chemically directed MS of glycoproteins facilitates the identification of sites of N-glycosylation

After confirming that the isotopic pattern was retained in the final glycan structure, we applied this labeling strategy to identify sites of N-glycosylation on secreted yeast proteins. To do this, media was filtered to remove cellular debris prior to concentration using tangential flow filtration. The concentrated protein fraction was then treated with EndoH to remove the bulk of the glycan structure prior to tryptic digestion. Peptides were then analyzed by LC-MS, and data was searched for the presence of the 1 : 2 : 1 isotopic pattern to generate an inclusion list of ions likely to be glycosylated, and additional MS experiments were performed using the inclusion list to direct MS/MS analysis of ions. MS/MS data was then searched against a yeast ORF database from the *Saccharomyces* genome database* to obtain peptide sequences and protein identifications. Data from multiple MS/MS experiments were then pooled and filtered for significance, and N-linked glycopeptides were extracted, producing 34 sites of N-glycosylation with high confidence in 15 proteins (Table 5-1).

This analysis also revealed a number of glycosylation sites with variable occupancy, such as the YGP1 peptide LFN@SSSALN@ITELYNVAR, where the doubly modified peptide, both singly modified peptides, and the unmodified peptide were all detected with high confidence.

*http://yeastgenome.org

Table 5-1: Identified peptides corresponding to N-linked glycoproteins in the secretome indicate multiple glycosylation sites of variable occupancy. Peptides were filtered for $p < 0.001$ and XCorr > 1.5. M# = OxoMet; C* = alkylated cysteine; N@ = N-GlcNAc.

Protein	Sequence	P-value	XCorr	Δcn
APE3	LAN@YSTPDYGHPTR	1.3e-20	2.21	0.49
APE3	SKEGLHGTLGEPTK	2e-04	2.21	0.353
CDC20	GNAAISGN@RSVLSIASPTK	5.4e-07	2.06	0.207
CDC20	HIHFWNSITGAR	1.1e-46	2.6	0.236
CDC20	CTGRSREDGLMDGM#LGLIGK	0.00062	1.65	0.238
CDC20	EDGLM#DGM#LGLIGK	2e-07	1.59	0.179
ECM33	AAFSN@LTTVGGGFIIAN@NTQLK	<1e-50	4.85	0.406
ECM33	VQTVGGAIEVTGN@FSTLDLSSLK	<1e-50	3.28	0.422
ECM33	DVNSISFGSLQTVN@ASLGFIN@NTLPSLN@LTQLSK	<1e-50	5.66	0.12
ECM33	VQTVGGAIEVTGN@FSTLDLSSLK	<1e-50	3.28	0.422
ECM33	VGQSLSIVSNDELSK	1.4e-22	3.54	0.631
GAS1	VYAIN@TTLDHSEC*MK	<1e-50	3.02	0.387
GAS1	TVVDTFAN@YTNVLGFFAGNEVTNN@YTNTDASAFVK	<1e-50	5.87	0.111
GAS1	FFYSNN@GSQFYIR	<1e-50	2.86	0.289
GAS1	ALNDADIYVIADLAAPATSINR	<1e-50	4.55	0.603
GAS1	ADFYGINMYEWC*GK	8.9e-10	3.08	0.47
HSP60	EDTVILN@GSGPKEAIQER	2.9e-20	1.86	0.166
HSP60	QIIENAGEEGSVIIGK	<1e-50	4.35	0.586
HSP60	TNEAAGDGTTSATVLGR	2.2e-35	3.11	0.548
HSP60	DRYDDALNATRAAVEEGILPGGGTALVK	5e-11	1.7	0.174
PDC1	MSAN@ISETTAM#ITDIATAPAEIDR	1.4e-28	1.58	0.503
PDC1	KLIDLTQFPAFVTPMGK	<1e-50	3.54	0.479
PDC1	KLIDLTQFPAFVTPM#GK	3.1e-15	3.08	0.49
PDC1	MIEIM#LPVFDAPQNLVEQAK	<1e-50	4.16	0.53
PDC1	M#IEIMLPVFDAPQNLVEQAK	<1e-50	4.38	0.513
PDC1	MIEIMLPVFDAPQNLVEQAK	<1e-50	5.4	0.582
PDC1	LIDLTQFPAFVTPMGK	<1e-50	4.46	0.634
PDC1	LKQVNVNTVFGLPGDFNLSLLDK	<1e-50	5.01	0.647
PDC1	QVNVNTVFGLPGDFNLSLLDK	<1e-50	4.57	0.64
PDC1	LLTTIADAAK	3.1e-15	2.21	0.315
PDC1	LLQTPIDMSLKPNDAESEKEVIDTILALVK	<1e-50	5.56	0.615
PDC1	TPANAAVPASTPLK	1.4e-28	3.28	0.589
PDC1	EVIDTILALVK	1.4e-28	2.91	0.406
PDC1	AQYNEIQGWDHLSLLPTFGAK	3.2e-10	3.04	0.317
PDI1	QSQPAVAVVADLPAYLAN@ETFVTPVIVQSGK	<1e-50	4.54	0.602
PDI1	LAPTYQELADTYAN@ATSDVLIAK	3e-42	3.41	0.43
PDI1	NSDVN@NSIDYEGPR	1.5e-46	2.74	0.133
PDI1	IDADFN@ATFYSMANK	3.6e-13	1.76	0.025
PDI1	ALYEEAQEK	5.5e-33	1.52	0.149
PDI1	SQEIFENQDSSVFQLVGK	2.4e-12	4.31	0.555
PDI1	GVVIEGYPTIVLYPGGK	<1e-50	4.37	0.644
PDI1	YGLPQLSEEAFDELSDK	<1e-50	4.13	0.522
PDI1	GLM#NFVSIDARK	<1e-50	1.75	0.506
PDI1	GVVIEGYPTIVLYPGGKK	1.1e-12	1.93	0.143
PDI1	TAEAIVQFMIK	8.8e-36	3.05	0.378
PHO3	SVGANLFN@ATLK	<1e-50	1.55	0.321
PHO3	GYSDVC*DIFTEDELVR	<1e-50	3.84	0.536
PHO3	IGTQEDIFPFLGGAGPYFSFPGDYGISR	1.5e-27	4.82	0.62
PRC1	VRN@WTASITDEVAGEVK	<1e-50	3.59	0.433
PRC1	N@WTASITDEVAGEVK	5.2e-32	2.51	0.49
PRC1	KDWDFVVKNDAIENYQLR	4.2e-07	1.53	0.024
PRC1	M#KAFTSLLC*GLGLSTTLAK	3.2e-10	1.5	0.237

Protein	Sequence	P-value	XCorr	∆cn
PRC1	NFLFAGDWM#K	7.5e-08	1.5	0.167
PST1	C*DTLVGN@LTIGGGLK	<1e-50	3.5	0.544
PST1	C*DTLVGNLTIGGGLK	3.7e-50	3.48	0.392
PST1	IGGLDN@LTTIGGTLEVVGN@FTSLNLDSLK	<1e-50	4.65	0.525
PST1	NLSFSN@LSTIGGALVVAN@NTGLQK	4.9e-23	1.77	0.268
PST1	SANNIYISDTSLQSVDGFSALKK	5.6e-16	1.91	0.298
PST1	LNTIGQTFSIVSNDYLK	4.9e-23	4	0.565
PST1	ITFDDLVWANNISLTDVHSVSFANLQK	7.3e-09	3.24	0.432
TAL1	ASGTVVVADTGDFGSIAKFQPQDSTTN@PSLILAAAK	1e-20	1.52	0.233
TAL1	ASGTVVVADTGDFGSIAK	<1e-50	4.92	0.556
TAL1	NLAGVDYLTISPALLDK	<1e-50	4.37	0.571
TAL1	VANNSLEQLK	1.4e-09	2.35	0.383
TAL1	LSFDTQATIEK	3.7e-25	2.76	0.521
TAL1	FDLNEDAMATEK	1.4e-09	2.66	0.262
TDH2	LKGVLGYTEDAVVSSDFLGDSN@SSIFDAAAGIQLSPKFVK	2e-10	2.19	0.103
TDH2	NVEVVALNDPFISNDYSAYMFK	8.5e-09	3.36	0.443
TDH2	LKGVLGYTEDAVVSSDFLGDSNSSIFDAAAGIQLSPK	8.5e-09	1.63	0.083
TDH2	YAGEVSHDDKHIIVDGHK	1.6e-07	2.18	0.508
TFP1	AVANGAN@WSKLADSTGDVK	7.8e-06	1.58	0.255
TFP1	NNLNTENPLWDAIVGLGFLK	<1e-50	3.83	0.607
TFP1	LNLCAEYKDR	4.3e-42	1.61	0.624
TFP1	GRETM#YSVVQKSQHR	5.5e-06	1.99	0.446
VMA4	DLVSGGVVVSNASDKIEIN@NTLEER	1.8e-06	1.99	0.026
VMA4	DLVSGGVVVSNASDK	1.8e-06	3.43	0.462
YGP1	LFN@SSSALN@ITELYNVAR	<1e-50	4.31	0.554
YGP1	LFNSSSALN@ITELYNVAR	<1e-50	4.38	0.584
YGP1	LFN@SSSALNITELYNVAR	<1e-50	3.71	0.62
YGP1	LFNSSSALNITELYNVAR	<1e-50	4.72	0.519
YGP1	VVN@ETIQDK	<1e-50	1.78	0.317
YGP1	RGVLSVTSDK	<1e-50	2.46	0.537
YGP1	PTLISSDSIIR	<1e-50	2.88	0.548
YGP1	LVYSGVFTPPTAC*SYGAGLPVAIVDDQDEVK	<1e-50	5.46	0.555
YGP1	GVLSVTSDKLVYSGVFTPPTAC*SYGAGLPVAIVDDQDEVK	<1e-50	5.63	0.693
YGP1	NAVGAGYLSPIKAQILLSIAAVNGVTSK	<1e-50	3.07	0.442
YGP1	SSAGAVVVANAK	<1e-50	3.53	0.552
YGP1	AQILLSIAAVNGVTSK	<1e-50	5.18	0.696

5.3 Discussion

This work demonstrates a new metabolic labeling approach where an isotopically labeled natural substrate is used to impart a detectable isotopic pattern on on a subset of ions based on their biological properties. They key advantages of this approach over traditional metabolic labeling strategies are efficiency and a lack of perturbation to the biological system. Since a natural substrate is utilized without the need for a secondary labeling step, labeling of glycan structures approaches 100% efficiency. In addition, since the modified substrate does not contain any altered functionality, its flux through biochemical pathways is assured to be reflective of the natural state of the system.

Initially there was some concern about incorporating an isotopic label at the anomeric position as the C-O bond at that position is broken multiple times during the biosynthesis of the N-glycan structure, and may introduce a skewing effect due to kinetic isotope effects. Our results indicate that this is not a major concern in this particular pathway. However, if this does prove to be problematic in other systems, the site of labeling can be moved to a less precocious location.

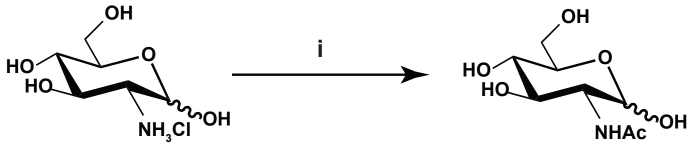
The IsoMix methodology is by no means limited to the study of N-glycosylation. We anticipate that this strategy is generally applicable to any systems where the metabolic precursor is synthetically tractable, and the biochemical pathways involved are relatively linear, such that a single precursor can be used to label the majority (*e.g.* >90%) of the cytosolic pool. In cases where the latter condition is not met, it may be possible to incorporate an labeling strategy with all masses being offset from the monoisotopic mass. For example, instead of incorporating a substrate with peaks at +0, +2, and +4 amu of the monoisotopic mass, a substrate with peaks at +4, +6, and +8 could be utilized.

The IsoMix system also introduces the possibility of using isotopic patterns other than the dibromide motif. One possible use of such designer patterns might be in the use of orthogonal isotopic tags, such that the presence of two different tags could be determined independently based solely on the isotopic distribution of the resultant ion. This may have applications in areas such as glycan sequencing or in the simultaneous analysis of multiple types of post-translational modifications.

5.4 Materials and methods

Synthesis of isotopically labeled N-acetylglucosamine

The synthesis of *N*-acetyl-D-glucosamine from D-glucosamine hydrochloride was performed in a single synthetic step according to the procedure described by Zhu *et al.*[22] Glucosamine hydrochloride (150 mg, 0.6 mmol) was dissolved in a minimal volume of H_2O, with the pH then adjusted to 7.5 by the addition of Dowex 200-400 mesh (OH-) anion-exchange resin. The resin was then washed 3x with H_2O, and the fractions were combined (6mL total). 1.1 eq $^{13}C_2$-sodium acetate (Cambridge Isotope Laboratories, Andover, MA; dissolved in minimal H_2O) was then added to the reaction, followed by 1.1 eq EEDQ (dissolved in 10 mL EtOH). The total reaction volume was then brough to 40 mL by addition of EtOH. The reaction was covered in foil and let to stir for 36h at room temperature. The reaction was repeated using 1-^{13}C,^{15}N-D-glucosamine hydrochloride (ISOTEC, Miamisburg, OH) as a starting



1 $^{12}C_6$,^{14}N-GlcN

4 1-$^{13}C_6$,^{15}N-GlcN

2 $^{12}C_8$,^{14}N-GlcNAc

3 1',2'-$^{13}C_2$,^{14}N-GlcNAc

5 1,1',2'-$^{13}C_3$,^{15}N-GlcNAc

Figure 5-6: The synthesis of *N*-acetyl glucosamine. **i)** pH GlcNH$_3$Cl to 7.5 using dowex 200-400 mesh $^-$OH-exchange resin, then 1.1eq NaOAc, 1.1eq EEDQ in H_2O/EtOH, r.t. 36hrs.

material.

The crude products were then purified by flash column chromatography (solvent: EtOAc : pyridine : H_2O, 10 : 4 : 3), dried, filtered, and lyophilized.

Combination and analysis of isotopes

Stock solutions of each sugar were made of approximately equal concentrations of 10 mM in H_2O, prior to making a solution containing all three isotope at a 1 : 2 : 1 molar ratio. This sample was then analyzed by direct infusion on an Thermo-Finnigan LTQ-XL mass spectrometer set to zoom scan, with the signal averaged over 20 scans. The isotopic ratios were then adjusted by the iterative addition of the desired isotopes, resulting in the isotopic signature shown in Figure 5-4. The IsoMix sample was then lyophylized and stored at -20°C where it is stable indefinitely.

Yeast growth and protein fractionation

A culture of Δgna1 *S. cerevisiae*[18] was grown in CSM (complete supplement medium, MP Biomedicals) with 2% dextrose and a 100uM 'IsoMix' GlcNAc (or regular GlcNAc) supplement at 30°C until reaching saturation.

Cells were pelleted by centrifugation and flash-frozen in LN2 and stored at -8°C. Fully-secreted proteins in the supernatant were 0.2 micron-filtered to remove debris and concentrated via tangential flow filtration with a 10 KDa NMWCO filter (Pall) and buffer-exchanged into PBS, flash-frozen in LN2, and stored at -80°C.

Total protein concentrations of secretome samples were measured colorometrically with the Biorad Dc assay.

Protein purification and preparation

Milligram-scale protein samples were prepared for MS analysis by denaturation, disulfide reduction/alkylation, EndoH deglycosylation, and digestion with trypsin. Detergent (Rapigest, 0.1% w/v) and 2.5mM DTT were added to secretome samples before heat denaturation by boiling 3 minutes followed by incubation at 56°C for 30 mins. Rapigest-solublized membrane samples were diluted to 0.2% (w/v) detergent and supplemented with 2.5 mM DTT prior to heat-denatuation and alkylation. Reduced cys residues were alkylated by adding 25 mM iodoacetamide followed by 1 hr incubation at RT in the dark before quenching by addition of 12mM DTT. N-glycans were removed by lowering pH to $\sim$5.5 with 50mM sodium citrate buffer and adding EndoH or EndoHf (NEB) and incubating 3-4 hrs at 37°C . Following deglycosylation, sample pH was adjusted to $\sim$7.5 with NaOH before adding proteomics-grade trypsin (Promega) at a ratio of 1:100 and O/N incubation at 37°C in siliconized eppendorf tubes.

Following tryptic digestion, samples were acidified with TFA to a final pH of $\sim$2 and incubated at 37°C 30 min to hydrolyze the detergent. Insoluble debris was removed, and the samples were desalted with Sep-Pak C18 cartridges (Waters). Peptides were eluted in 80% MeCN w/ 0.1% Formic acid; solvent was removed until near-dryness and samples were stored at -20°C until use. Peptides were resuspended in water immediately prior to MS analysis.

After fractionation, protein samples were reduced by adding 2.5 mM DTT and incubated at 56 °C for 30 min, and cysteine residues alkylated with 10 mM iodoacetamide in the dark at room temperature for 1 h.

LC-MS analysis

All samples were subjected to reversed-phase capillary chromatography with an Agilent 1200 LC system using 100-μm × 1-cm (5-μm, 200 Magic C18AQ resin; Micrhom Bioresources, Auburn, CA) fritted capillary pre-column and a 100-μm × 10-cm self-packed C18 column (5-μm, 100 Magic C18AQ resin; Michrom Bioresources, Auburn, CA). A binary solvent system consisting of buffer A (0.1% formic acid) and buffer B (0.1% formic acid in acetonitrile) was employed. After a 10 min loading step in 2% buffer B, a gradient was employed from 10% to 40% buffer B for 62 min, followed by a washing step in 99% buffer B for 10 min. A solvent split was used to maintain a flow rate of 400 nL/min at the column tip. Data were collected on a Thermo-Finnigan LTQ Orbitrap XL mass spectrometer set at 60,000 resolution in full scan mode with an m/z scan range of 400-2000.

Data analysis

Raw data for full-scan LC-MS experiments were converted to the mzXML format through the use of the program ReAdW. Peak detection was then performed using a continuous wavelet transform (Appendix C), and searched for singly- and doubly-labeled peptides (corresponding to dibromide-like and tetrabromide-like patterns, respectively) using the IsoStamp algorithm, and results were output as an inclusion list of m/z values.

Biological samples were then analyzed by LC-MS/MS using the generated inclusion lists as described above. MS/MS data was searched against an annotated yeast OFR database using SEQUEST (Bioworks, Thermo-Finnigan) to determine peptide identifications. Allowed modifications included cysteine alkylation, methionine oxidation, and N-linked GlcNAc. Data from multiple samples were combined based on peptide sequence, and combined significance levels were calculated using Stouffer's Z-score method[23].

Peptides were then filtered for to include only identified species with P-Values < 10^{-3} and XCorr scores > 1.5. Peptides with assigned non-canonical sites were removed if moving the glycosylation site by a single residue produced a canonical site (*e.g.* NN@SG → N@NSG) as this type of misassignment can be a common artifact of the SEQUEST algorithm.

Acknowledgements

The work in this chapters was done in close collaboration with Dr. Mark A. Breidenbach and Kanna K. Palaniappan. Mark is responsible for growing yeast cultures, isolating and purifying proteins, and for help in analyzing data. Kanna is responsible for the MS experiments as well as for help in analyzing data.

References

[1] Burda, P & Aebi, M. (1999) The dolichol pathway of N-linked glycosylation. *Biochim. Biophys. Acta-Gen. Subj.* **1426**: 239–257.

[2] Jones, J, Krag, S. S, & Betenbaugh, M. J. (2005) Controlling N-linked glycan site occupancy. *Biochim. Biophys. Acta.* **1726**: 121–137.

[3] Weerapana, E & Imperiali, B. (2006) Asparagine-linked protein glycosylation: from eukaryotic to prokaryotic systems. *Glycobiology* **16**: 91R–101R.

[4] Dennis, J. W, Nabi, I. R, & Demetriou, M. (2009) Metabolism, cell surface organization, and disease. *Cell* **139**: 1229–1241.

[5] Zhao, Y et al. (2008) Branched N-glycans regulate the biological functions of integrins and cadherins. *FEBS J.* **275**: 1939–1948.

[6] Rudd, P. M et al. (1999) Roles of glycosylation of cell surface receptors involved in immune recognition. *J. Mol. Biol.* **293**: 351–366.

[7] Dennis, J. W, Lau, K. S, Demetriou, M, & Nabi, I. R. (2009) Adaptive regulation at the cell surface by N-glycosylation. *Traffic* **10**: 1569–1578.

[8] Zhao, Y. Y et al. (2008) Functional roles of N-glycans in cell signaling and cell adhesion in cancer. *Cancer Sci.* **99**: 1304–1310.

[9] Lau, K. S & Dennis, J. W. (2008) N-glycans in cancer progression. *Glycobiology* **18**: 750–760.

[10] Helenius, A & Aebi, M. (2004) Roles of N-linked glycans in the endoplasmic reticulum. *Annual Rev. Biochem.* **73**: 1019–1049.

[11] Ruddock, L. W & Molinari, M. (2006) N-glycan processing in ER quality control. *J. Cell Sci.* **119**: 4373–4380.

[12] Molinari, M. (2007) N-glycan structure dictates extension of protein folding or onset of disposal. *Nat. Chem. Biol.* **3**: 313–320.

[13] O'Connor, S. E & Imperiali, B. (1997) Conformational switching by asparagine-linked glycosylation. *J. Am. Chem. Soc.* **119**: 2295–2296.

[14] O'Connor, S. E & Imperiali, B. (1998) A molecular basis for glycosylation-induced conformational switching. *Chem. Biol.* **5**: 427–437.

[15] Zielinska, D. F, Gnad, F, Wiśniewski, J. R, & Mann, M. (2010) Precision mapping of an in vivo N-glycoproteome reveals rigid topological and sequence constraints. *Cell* **141**: 897–907.

[16] Plummer, Jr, T. H & Tarentino, A. L. (1991) Purification of the oligosaccharide-cleaving enzymes of *flavobacterium menigosepticum*. *Glycobiology* **1**: 257–263.

[17] Robbins, P. W et al. (1984) Primary structure of the *streptomyces endo-beta-n-acetylglucosaminidase h*. *J. Biol. Chem.* **259**: 7577–7583.

[18] Breidenbach, M. A et al. (2010) Targeted metabolic labeling of yeast N-glycans with unnatural sugars. *Proc. Natl. Acad. Sci.* **107**: 3988–3993.

[19] Wang, Z et al. (2010) Extensive crosstalk between O-GlcNAcylation and phosphory-

lation regulates cytokinesis. *Science Signaling* **3**: ra2.

[20] Neufeld, E. F & Ginsburg, V. (1965) Carbohydrate metabolism. *Annu. Rev. Biochem.* **34**: 297–312.

[21] Vocadlo, D. J, Hang, H. C, Kim, E. J, Hanover, J. A, & Bertozzi, C. R. (2003) A chemical approach for identifying O-GlcNAc-modified proteins in cells. *Proc. Natl. Acad. Sci.* **100**: 116–9121.

[22] Zhu, Y et al. (2006) [^{13}C,^{15}N]2-acetamido-2-deoxy-Daldohexoses and the methyl glycosides: synthesis and NMR investigations of *j*-couplings invlolving ^{1}H, ^{13}C and ^{15}N. *J. Org. Chem.* **71**: 466–479.

[23] Whitlock, M. C. (2005) Combining probability from indepndent tests: the weighted Z-method is superior to Fisher's approach. *J. Evol. Biol.* **18**: 1368–1373.

chapter six

The future of isotopic labeling in mass spectrometry

The previous chapters have outlined a strategy by which isotopic fingerprinting labels can be used to demarcate a subset of ions from a biological sample as being information rich. The key to this strategy is that the isotopic tag can be incorporated into peptides or other biomolecules which are most likely to provide information that is interesting in the context of a biological question. One method of accomplishing this task is through the use of binary tagging strategies, such as the cysteine-alkylation strategy used in Chapters 2. This technique is essentially limited by the efficiency and selectivity of the chemistry involved in the labeling step.

An alternative strategy is to incorporate the isotopic tag directly into the biomolecule of interest, as exemplified in the previous chapter. This strategy has the advantage that there is no loss of efficiency due to a chemical labeling step. However, for this strategy to be tractable, the isotopic signature must reach its biological target essentially unchanged. This means that there must be limited competition for the isotopically tagged species among a variety of biological pathways, and that there must be limited *de novo* biosynthesis of the tagged molecule to prevent dilution of the signal. In the previous chapter, this was ensured through the use of a yeast strain lacking many of the competing pathways, such that the isotopic tag was incorporated unobscured and at 100% efficiency.

A logical extension of the IsoStamp method is to expand the scope of the tagging strategy to include a number of the common bioorthogonal reactions outlined in Chapter 1. Section 6.1 discusses some initial work in this direction, along with application in glycoproteomics through the metabolic incorporation of unnatural sugars. Extending applications even further, section 6.2 suggests a number of biological systems of the IsoStamp technique, and Section 6.3 explores the idea of multifunctional tags incorporating isotopic fingerprinting labeling with other extant tagging strategy currently used in MS. Section 6.4 examines what the future of chemically directed proteomics may look like.

A relatively recent arrival on the mass spectral scene is the idea of using mass spec-

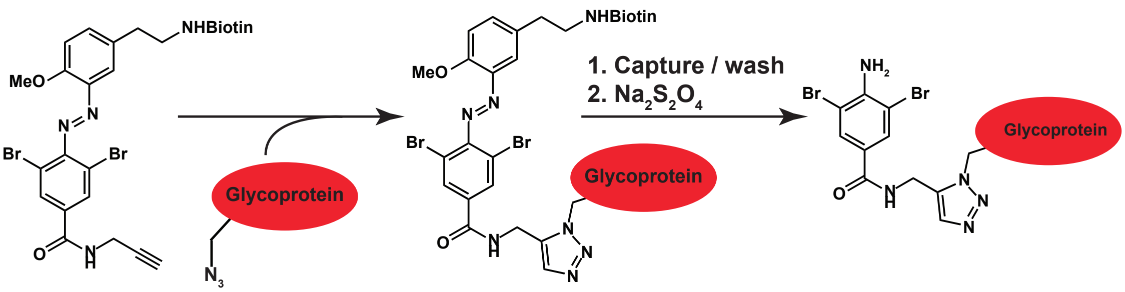


Figure 6-1: The dibromide motif can be incorporated through bioorthogonal chemistry. An azide-reactive alkynyl dibromide tag can be reacted with glycoproteins bearing unnatural sugars with azide functionality. The protein mixture can then be purified using biotin-avadin affinity chromatography, and glycoproteins can be eluted upon treatment with sodium bisulfite and analyzed by directed MS.

trometry to generate images of biological samples, a technique termed "mass spectrometry imaging" (MS imaging for short)[1]. In this technique, a biological sample is fixed to a solid support, and mass spectra are obtained at predetermined xy-corrdinates, generating a multidimensional image of the sample. Section 6.5 introduces the idea of isotopic tagging in the context of MS imaging, and proposes a method in which the graph theoretic approach introduced in Chapter 3 might be applied to the analysis of MS imaging data.

Finally, Section 6.6 highlights the possibility of using multiple isotopic tagging strategies in combination to help elucidate glycan composition. Such a system would be useful to discriminate between monosaccharides of identical molecular weight when analyzing large glycans of unknown composition by mass spectrometry.

6.1 Using metabolic engineering to enable chemically directed proteomics

As discussed in Chapter 1, unnatural sugars can be incorporated into glycoproteins through the hijacking of cells' biosynthetic machinery, allowing the introduction of bioorthogonal chemical groups into these biomolecules. Some unnatural sugars and their biological targets are given in Table 1-1. Initial work towards applying the IsoStamp technology to metabolically engineered glycoproteins has been carried out by Brian Smart and Krishnan Palaniappan, which has resulted in the synthesis of a multi-functional azide reactive dibromide tag, as can be seen in Figure 6-1.

An attractive first target for combining metabolic engineering with the IsoStamp technology is in the analysis of cytosolic O-GlcNAcylation. Cytosolic O-GlcNAcylation is a dynamic form of glycosylation, in which a single GlcNAc moiety is ap-

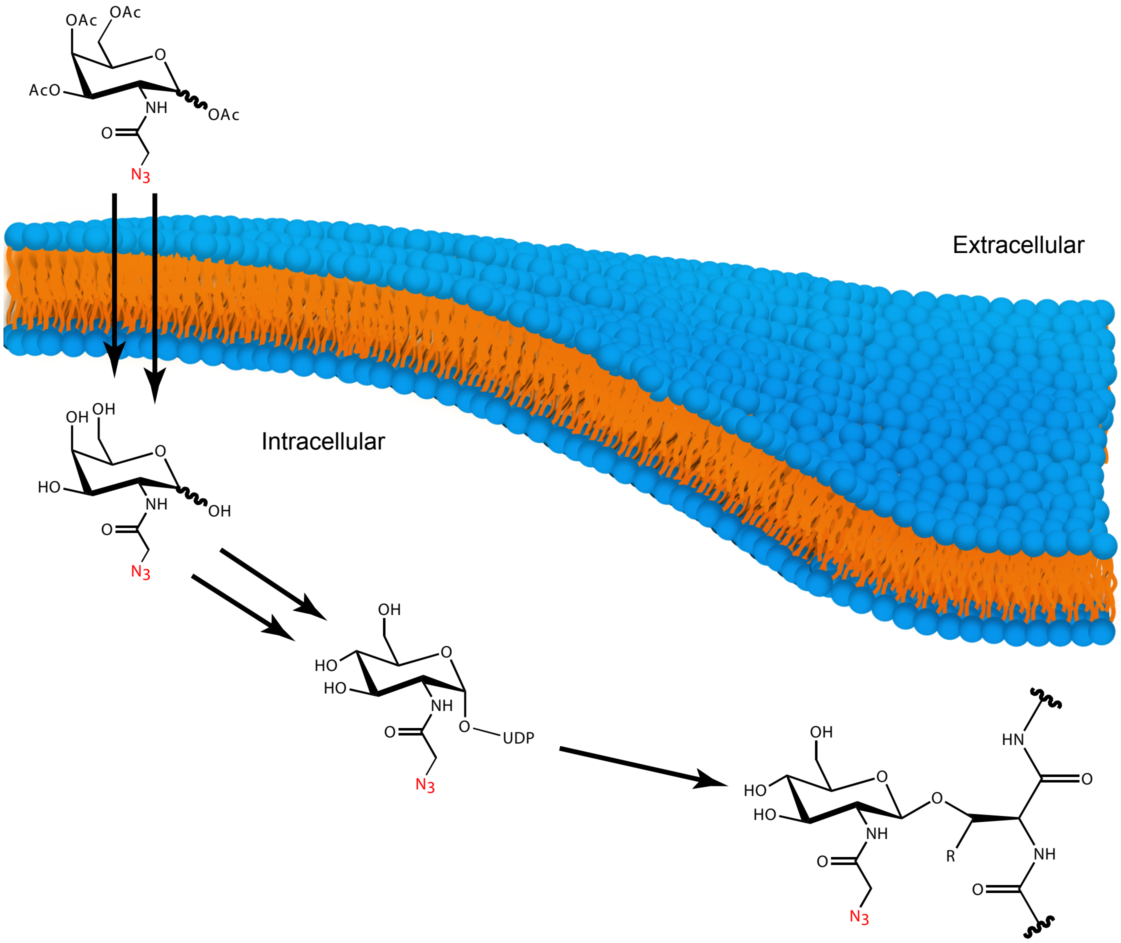


Figure 6-2: The metabolic incorporation of GlcNAz into O-GlcNAcylated proteins. Peracetalated *N*-acetylgalactosamine (GalNAz) can be added to culture media where it is then taken up by cells through the GalNAc salvage pathway, where it is ultimately converted to UDP-GalNAz and epimerized at the 4 position to produce UDP-GlcNAz (see Figure 5-1 for details). UDP-GlcNAz is then a substrate for *O*-linked *N*-acetylglucosamine transferase (OGT), the sole enzyme responsible for cytosolic GlcNAcylation.

pended to protein Ser/Thr residues by the enzyme O-GlcNAc Transferase (OGT) or removed by the enzyme O-GlcNAcase[2,3]. Compared to cell surface glycosylation, the study of cytosolic O-GlcNAcylation is simplified by the fact that only a single sugar is attached to proteins, removing the difficulties of dealing with glycan heterogeneity. Furthermore, O-GlcNAc is interesting from a biological point of view: it has been implicated as been an essential component of a number of disease states including diabetes[4–7] and a number of signal transduction pathways[8,9], and has been proposed to be complementary to the action of myriad kinases within mammalian cells[10].

The O-GlcNAc modification can be targeted through metabolically labeling cells with peracetylated GalNAz, as shown in Figure 6-2. Briefly, after entering the cell, the sugar is deacetylated and ultimately converted to the donor sugar UDP-GalNAz, which is then epimerized at the 4 position to produce the OGT substrate UDP-GlcNAz. The reason that GalNAz is used rather than GlcNAz is due to recent work showing an increase in labeling efficiency and specificity with the use of GalNAz (work by Dr. Mike Boyce, manuscript in preparation). After labeling, proteins from the cytosolic fraction would be purified prior to labeling with an azide-reactive dibromide tag. Tag-labeled proteins could then be purified by affinity chromatography and analyzed by directed LC-MS/MS. This technique would give a platform by which one could analyze dynamic changes in cytosolic O-GlcNAcylation of low-abundance proteins. Furthermore, under ideal conditions, the dibromide label would be detectable in individual peptides' tandem MS, which could be used to improve confidence in database searching and assignment of modification sites.

6.2 Beyond glycosylation

Though glycosylation is an attractive application for directed proteomics, the IsoStamp technology is only limited by the ability to incorporate the isotopic signature into the desired biomolecule. Thus, this approach could be used to study a number of other types of posttranslational modifications, including lipidation[11–13] and phosphorylation[14], as well as for the analysis of cysteine oxidation[15] and active sites of enzymes[16]. Other biochemical tagging strategies, such as the labeling of protein N-termini, are also amenable to this technique[17,18].

Furthermore, the IsoMix approach used in the previous chapter could be applied to a number of biochemical targets. One enticing possibility would be the isotopic tagging of GPI anchors, in particular because GPI anchor attachment does not follow a clear consensus sequence, and the anchor itself would be accessible through a scheme very similar to that used to label yeast N-glycans. Though GPI anchor structure can vary between organisms, there is a conserved glucosamine core that is derived from the UDP-GlcNAc pool[19–21]. The relevant steps of GPI anchor biosynthesis are shown in Figure 6-3[22]. The major change from the labeling protocol previously used would be the location of the heavy isotopes within the GlcNAc: since the acetate is removed during the construction of the GPI anchor, only isotopes within the glucosamine core would be retained. However, this would be easily achieved with commercially available isotopes of GlcNAc.

Using the same UPD-GlcNAc auxotrophic yeast described earlier, competition for the UDP-GlcNAc pool from non-labeled sugars would be removed, permitting

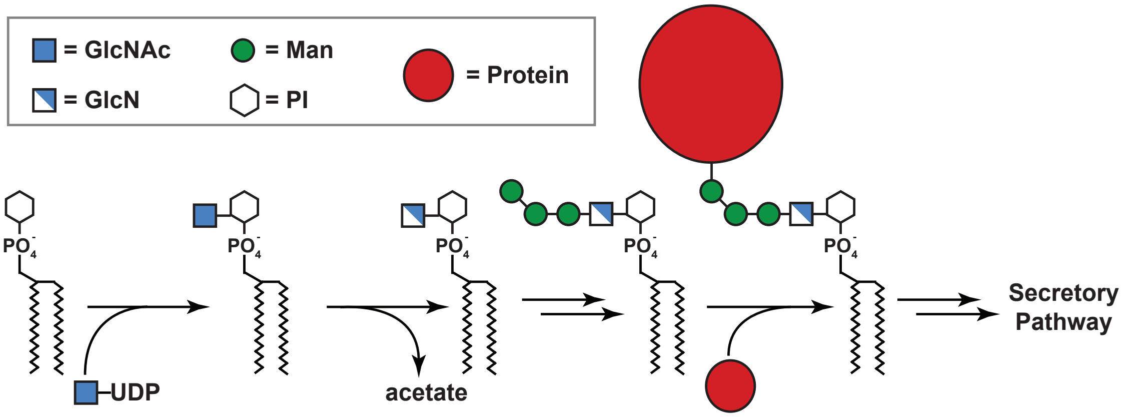


Figure 6-3: The biosynthesis of GPI anchors in mammalian cells permits incorporation of IsoMix GlcNAc through the metabolic targeting of UDP-GlcNAc.

efficient incorporation of the isotopic label into the GPI anchored pool of proteins. Proteins could then be isolated using standard protocols, and directed MS could be used to analyze modified proteins, including the determination of sites of modification.

Another potential target for the IsoMix strategy is the labeling of fucosylated proteins. Fucosylation of O- and N-linked glycans has been shown to have a number of essential biological functions[23], and has been shown to be an important biomarker for a number of disease states including liver disease and cancer[24,25]. Furthermore, O-fucose—the addition of a single fucose moiety to protein Ser/Thr residues—has been shown to be essential to development as in the case of Notch fucosylation[26]. However, attempts to target fucose through metabolic engineering have proven to be challenging, in part due toxicity cause by unnatural fucose analogs[27].

Fucose is also an attractive target because the fucose salvage pathway has been shown to be highly efficient compared to other salvage pathways, suggesting that it may be possible to target the metabolic precursor, GDP-fucose, simply by growing cells in media containing IsoMix fucose. Alternatively, should the salvage pathway prove to be inadequate, produce of a GDP-fucose auxotrophic cell line would only require the knockout of one of two genes in the *de novo* fucose biosynthetic pathway, as both GDP-mannose 6-dehydrogenase (GDH) and a dual-function epimerase-reductase, termed the FX Protein, are required for the production of GDP-fucose from non-fucose pools[28].

6.3 Multifunctional tagging strategies

One of the attractions of the IsoStamp system is that it need not be used in isolation. The extremely complex nature of biological samples as they relate to mass spectrometry poses a problem that will likely not be solved by a single strategy, but will likely necessitate the combination of a number of functionalities. Fortunately, many technologies can be combined in a single small molecule tag, such as that proposed by Hart and coworkers, combining a reactive group, an affinity label, a cleavable linker, and an ionizable group to improve MS signal[29]. Other tagging strategies used in mass spectrometry include isotope labels for relative quantitation between samples, such as ICAT[30] and iTRAQ[31]. Each of these tagging strategies could be modified to include an isotopic fingerprinting label such as the dibromide tag. Such a combination of functionalities in a single tag would allow for multiple methods of enrichment (e.g., affinity purification along with ionization enrichment), quantification, and directed MS to be achieved using a single tagging step, with no loss of efficiency of any of the component technologies.

Initial work on integrating the IsoStamp methodology with isotopic quantitation techniques is underway. A library functions to facilitate peak and ion integration that can be interfaced with the pattern searching software can be found in Appendix B.6.

6.4 The future of chemically directed proteomics

At present, the largest limitation to the IsoStamp technology is the requirement of two separate LC-MS experiments: the first to determine the location of labeled species, and the second to obtain fragmentation data on the same sample. Ideally, the technology could be used without the necessitating two separate experiments, saving both time and samples. In order to achieve this, the pattern searching software would need to be integrated with the MS instrument's operational software such that full scan data is searched in real time, and matched patterns are fragmented as soon as they are detected.

In addition to streamlining the process, such an on-line analysis strategy would offer an additional benefit: since the instrument would fragment ions as soon as they were detected, it increases the likelihood that the ion came from a tagged species and rather than an ion that had the same m/z value as a previously detected species. This would serve to increase the efficiency of fragmenting labeled species. As the current searching software can processes an entire LC-MS dataset in under five minutes, the technical difficulties to achieve such an integrated system should be within reason.

6.5 Applications of isotopic labeling to MS imaging

MS Imaging is a relatively young technology in which a fixed two-dimensional structure, typically a tissue sample, is moved in both dimensions and a mass spectrum is obtained at fixed xy-coordinates. At present, there are primarily two methods for performing MS imaging: MALDI imaging and SIMS imaging, which differ primarily in the mechanism of ionization[1]. In MALDI imaging, a sample is fixed to a solid support prior to the application of a MALDI matrix (e.g., α-cyano-4-hydroxycinnamic acid). A laser is then rasterized across the surface, volatilizing the sample at discrete x,y coordinates, after which a mass spectrum is obtained[32]. At present, the spatial resolution of MALDI imaging is limited by the size of crystals in the matrix, though there have been recent reports of spatial resolution as high as 15 μm on peptide samples[33].

In contrast, SIMS (secondary ion mass spectrometry) imaging uses a focused ion beam, such as that of C_{60} fullerene, to ionize a solid sample at discrete coordinates, again obtaining a mass spectrum at each point[34]. Since SIMS does not require a crystal matrix for ionization, the spatial resolution is not limited by crystal size but rather on the ability to focus an ion beam, with current technology providing resolution as high as 3 μm[35]. While these resolutions are large relative to the size of subcellular structures, resolution for MS imaging techniques are expected to improve drastically with the technological advances in ionization techniques, making MS imaging of cellular structures a tractable task.

Already, MS imaging has been used to analyze a diverse set of biological problems, from analyzing tissue structure and organization[33,36,37], as well as for analyzing the location of action of natural products[38] and for drug discovery[39]. In general, the current use of MS imaging is to produce an image based on the signal intensity at one m/z value, and thus the technique is limited to cases in which there is prior knowledge about the molecular species present at a given m/z value. The IsoStamp technology lends itself perfectly to this situation.

The IsoStamp technique could be incorporated into MS imaging through two straightforward steps. First, the isotopic label would have to be incorporated into the biological structure of interest. This could be done through a secondary labeling strategy as in Chapter 2, through direct incorporation of the isotopic label as in Chapter 5, or through any number of hybrid methods as discussed above. Secondly, the MS data would have to be searched for the dibromide pattern prior to creation of two-dimensional images. This could be accomplished using the same general algorithm put forth in Chapter 3, requiring only modification of the structure of the graph used for false positive reduction. Rather than treating two nodes as potential neighbors if they elute from the LC column at similar times, nodes would be treated as potential neighbors if they were located in adjacent pixels in the two dimensional grid. This would allow the generation of images based specifically on the signal generated by a specific set of biomolecules based on their chemical or biological properties, rather than based on a single biomolecule. Furthermore, the use of a graph construct here easily lends itself to further image analysis such as segmentation and edge detection[40–44]. This suggests that structures within cells or tissues could be identified based solely on the signal of a specified set of biomolecules. Such a system has the potential to be a powerful tool for analyzing physiological structures at the molecular level.

6.6 Orthogonal isotopic labels

The IsoMix system described in the previous chapter introduces the possibility of "designer" isotopic signature tags representing a broad range of tunable isotopic signatures, limited only by the availability or synthetic tractability of isotopically labeled building blocks. For, example, applications in glycobiology, a large number of ^{13}C and ^{15}N sugars are commercially available making it possible to create mixtures of isotopes covering a broad range of patterns. In theory, two isotopic tags could be developed such that the number of each tag on a molecule could be determined from the full-scan mass spectrum alone. If these orthogonal isotopic tags were then attached to two different chemical functionalities (e.g., azide- and carbonyl-reactive tags) or to two different biological subunits (e.g., GlcNAc and GalNAc), the composition of the biomolecule with respect to those two functionalities or subunits could be determined from the full-scan mass spectrum alone. Furthermore, in the case of glycomics, such a system could improve the elucidation of glycan structures and may be able to add sufficient information to improve existing software for glycan analysis such as CartoonistTwo[45].

While in theory such a technique could be used to determine the composition of large polymeric structures, in practice such an approach will likely be limited by isotopic dilution effects, as the signal for the entire ion is divided amongst a large number of peaks. Increased instrumental resolution and sensitivity combined with improved methods of sample preparation will increase the scope of such an application, but with current technology it is estimated that such a labeling strategy would work well for molecules with up to four isotopic labels, after which isotopic dilution and decreased fidelity in recreating the isotopic pattern will be limiting.

References

[1] McDonnell, L. A & Heeren, R. M. (2007) Imaging mass spectrometry. *Mass. Spectrom. Rev.* **26**: 606–643.

[2] Torres, C & Hart, G. W. (1984) Topography and polypeptide distribution of terminal N-acetylglucosamine residues on the surfaces of intact lymphocytes: evidence for O-linked glcnac. *J. Biol. Chem.* **259**: 3308–3315.

[3] Wells, L, Vosseller, K, & Hart, G. W. (2001) Glycosylation of nucleocytoplasmic proteins: signal transduction and o-glcnac. *Science* **291**: 2376–2378.

[4] Dentin, R, Hedrick, S, Xie, J, Yates III, J, & Montminy, M. (2008) Hepatic glucose sensing via the CREB coactivator CRTC2. *Science* **319**: 1402–1405.

[5] Yang, X et al. (2008) Phosphoinositide signalling links *O*-GlcNAc transferase to insulin resistance. *Nature* **451**: 961–967.

[6] Goldberg, H. J, Whiteside, C. I, Hart, G. W, & Fantus, I. G. (2006) Posttranslational, reversible *o*-glycosylation is stimulated by high glucose and mediates plasminogen activator inhibitor-1 gene expression and Sp1 transcriptional activity in glomerular mesangial cells. *Endocrinology* **147**: 222–231.

[7] Cooksey, R. C, Pusuluri, S, Hazel, M, & McClain, D. A. (2005) Hexosamines regulate sensitivity of glucose-stimulated insulin secretion in β-cells. *Am. J. Physiol. Endocrinol. Metab.* **290**: E334–E340.

[8] Jackson, S. P & Tjian, R. (1988) O-Glycosylation of eukaryotic transcription factors - implications for mechanisms of transcriptional regulation. *Cell* **55**: 125–133.

[9] Zachara, N. E & Hart, G. W. (2002) The emerging significance of O-GlcNAc in cellular regulation. *Chem. Rev.* **102**: 431–438.

[10] Wang, Z et al. (2010) Extensive crosstalk between O-GlcNAcylation and phosphorylation regulates cytokinesis. *Science Signaling* **3**: ra2.

[11] Heal, W. P et al. (2008) Site-specific N-terminal labelling of proteins *in vitro* and *in vivo* using N-myristoyl transferase and bioorthogonal ligation chemistry. *Chem. Commun.* pp. 480–482.

[12] Heal, W. P, Wickramasinghe, S. R, Leatherbarrow, R. J, & Tate, E. W. (2008) N-myristoyl transferase-mediated protein labeling *in vivo*. *Org. Biomol. Chem.* **6**: 2308–2315.

[13] Roth, A. F et al. (2006) Global analysis of protein palmitoylation in yeast. *Cell* **125**: 1003–1013.

[14] Blethrow, J. D, Glavy, J. S, Morgan, D. O, & Shokat, K. M. (2008) Covalent capture of kinase-specific phosphopeptides reveals Cdk-1-cyclin B substrates. *Proc. Natl. Acad. Sci.* **105**: 1442–1447.

[15] Reddie, K. G, Seo, Y. H, Muse III, W. B, Leonard, S. E, & Carroll, K. S. (2008) A chemical approach for detecting sulfenic acid-modified proteins in living cells. *Molecular Biosystems* **4**: 521–531.

[16] Meier, J. L et al. (2009) An orthogonal active site identification system (OASIS) for proteomic profiling of natural product biosynthesis. *ACS Chem. Biol.* **4**: 948–957.

[17] Agard, N. J, Maltby, D, & Wells, J. A. (2010) Inflammatory stimuli regulate caspace substrate profiles. *Mol. Cell. Proteomics* **9**: 880–893.

[18] Heal, W. P & Tate, E. W. (2010) Getting a chemical handle on protein post-translational modification. *Org. Biomol. Chem.* **8**: 731–738.

[19] Nosjean, O, Briolay, A, & Roux, B. (1997) Mammalian GPI proteins: sorting, membrane residence and functions. *Biochim. Biophys. Acta.* **1331**: 153–86.

[20] Paulick, M. G & Bertozzi, C. R. (2008) The glycosylphosphatidylinositol anchor: a complex membrane-anchoring structure for proteins. *Biochemistry* **47**: 6991–7000.

[21] Hoessli, D & Ilangumaran, S, eds. (1999) *GPI-anchored membrane proteins and carbohydrates*. (Landes Bioscience).

[22] Kinoshita, T, Ohishi, K, & unji Takeda, J. (1997) GPI-anchor synthesis in mammalian cells: genes, their products, and a deficiency. *J. Biochem.* **122**: 251–257.

[23] Staudacher, E, Altmann, F, Wilson, I. B, & März, L. (1999) Fucose in N-glycans: from plant to man. *Biochim. Biophys. Acta.* **1473**: 216–236.

[24] Mehta, A & Block, T. M. (2008) Fucosylated glycoproteins as markers of liver disease. *Dis. Markers* **25**: 259–265.

[25] Miyoshi, E, Moriwaki, K, & Nakagawa, T. (2008) Biological function of fucosylation i n cancer biology. *J. Biochem.* **143**: 725–729.

[26] Lei, L, Xu, A, Panin, V. M, & Irvine, K. D. (2003) An O-fucose site in the ligand binding domain inhibits Notch activation. *Development* **130**: 6411–6421.

[27] Rabuka, D, Hubbard, S. C, Laughlin, S. T, Argade, S. P, & Bertozzi, C. R. (2006) A chemical reporter strategy to probe glycoprotein fucosylation. *J. Am. Chem. Soc.* **128**: 12078–12079.

[28] Becker, D. J & Lowe, J. B. (2003) Fucose: biosynthesis and biological function in mammals. *Glycobiology* **13**: 41R–53R.

[29] Wang, Z et al. (2010) Enrichment and site mapping of O-linked N-acetylglucosamine by a combination of chemical/enzymatic tagging, photochemical cleavage, and electron transfer dissociation mass spectrometry. *Mol. Cel. Proteomics* **9**: 153–160.

[30] Gygi, S. P et al. (1999) Quantitative analysis of complex protein mixtures using isotope-coded affinity tags. *Nature Biotechnology* **17**: 994–999.

[31] Ross, P. L et al. (2004) Multiplexed protein quantitation in *saccharomyces cerevisiae* using amine-reactive isobaric tagging reagents. *Mol. Cell. Proteomics* **3**: 1154–1169.

[32] Chaurand, P, Norris, J. L, Comett, D. S, Mobley, J. A, & Caprioli, R. M. (2006) New developments in profiling and imaging of proteins from tissue sections by MALDI mass spectrometry. *J. Prot. Res.* **5**: 2889–2900.

[33] Snel, M. F & Fuller, M. (2010) High-spatial resolution matrix-assisted laser desorption ionization imaging analysis of glucosylceramide in spleen section from a mouse model of gaucher disease. *Anal. Chem.* **82**: 3664–3670.

[34] Nygren, H & Malmberg, P. (2007) High resolution imaging by organic secondary ion

mass spectrometry. *Trends Biotechnol.* **25**: 499–504.

[35] Nygren, H & Malmberg, P. (2010) High-resolution imaging and proteomics of peptide fragments by TOF-SIMS. *Proteomics* **10**: 1694–1698.

[36] Rohner, T. C, Staab, D, & Stoeckli, M. (2005) MALDI mass spectrometric imaging of biological tissue sections. *Mech. Ageing. Dev.* **126**: 177–185.

[37] Fletcher, J. S. (2009) Cellular imaging with secondary ion mass spectrometry. *Analyst.* **134**: 2204–2215.

[38] Esquenazi, E, Yang, Y. L, Watrous, J, Gerwick, W. H, & Dorrestein, P. C. (2009) Imaging mass spectrometry of natural products. *Nat. Prod. Rep.* **26**: 1521–1534.

[39] Solon, E. G, Schweitzer, A, Stoeckli, M, & Prideaux, B. (2010) Autoradiography, MALDI-MS, and SIMS-MS imaging in pharmaceutical discovery and development. *AAPS J.* **12**: 11–26.

[40] Shi, J & Malik, J. (2000) Normalized cuts and image segmentation. *IEEE T-PAMI* **22**: 888–905.

[41] Grady, L. (2006) Random walks for image segmentation. *IEE T-PAMI* **28**: 1768–1783.

[42] Wu, Z & Leahy, R. (1993) An optimal graph theoretic approach to data clustering: theory and its application to image segmentation. *IEEE T-PAMI* **15**: 1101–1113.

[43] Grady, L & Schwartz, E. L. (2006) Isoperimetric graph partitioning for image segmentation. *IEE T-PAMI* **28**: 469–475.

[44] Zahn, C. T. (1971) Graph-theoretical methods for detecting and describing gestalt clusters. *IEE Trans. Comp.* **20**: 68–86.

[45] Goldberg, D, Bern, M, Li, B, & Lebrilla, C. (2006) Automatic determination of O-glycan structure from fragmentation spectra. *J. Prot. Res.* **5**: 1429–1434.

the mighty appedix

appendix a

Halogen tag sensitivity

This appendix contains the analysis of the raw MS data files used to to generate Figure 2-4A, examining the relative sensitivity of each of the halogenated tags. In each MS file, data were searched for the presence of ions at the appropriate m/z value for charge states ranging from +1 to +5. Ions found were then compared against a reference dataset ("Ideal" in each table) that contained no added lysate as a control for retention time and ionizability.

Numbers in each column indicate the number of ions found for each peptide listed, while values of "—" indicate that no ions corresponding to that peptide were found in any of the samples analyzed, including the sample of pure BSA. Data was obtained on an LTQ Orbitrap XL (resolution=60,000) for the dibromide (Table A-1), dichloride (Table A-2), and monobromide (Table A-3) tags, while data obtained on a Waters Q-ToF Premier (resolution=10,000) was only obtained for dibromide-tagged BSA (Table A-4). Original file names for each sample are provided in the table notes.

Table A-1: Detection rate of dibromide-tagged BSA peptides in MS experiments where the indicated amount of tagged BSA was spiked into 10 μg of Jurkat cell lysate before tryptic digestion and LC-MS analysis on an LTQ-Orbitrap XL (resolution=60,000) mass spectrometer.

						BSA Concentration*							
Residues	**Sequence**	**Missed Cleavages**	**Mods.**[†]	**M.I. Mass**[‡]	**Ideal**[§]	**30 ng**	**3 ng**	**1.5 ng**	**0.8 ng**	**0.3 ng**	**0.15 ng**	**0.08 ng**	**0.03 ng**
223-228	(R)CASIQK(F)	0		649.3338	2	2	2	2	2	2	0	0	0
198-204	(K)GACLLPK(I)	0		701.4015	2	2	2	2	2	2	1	0	0
483-489	(R)LCVLHEK(T)	0		841.46	3	3	2	2	2	2	0	0	0
221-228	(R)LRCASIQK(F)	1		918.5189	—	—	—	—	—	—	—	—	—
123-130	(R)NECFLSHK(D)	0		977.4509	3	3	2	2	2	0	0	0	0
413-420	(K)QNCDQFEK(L)	0	pyroGlu	994.3935	2	2	2	2	1	0	0	0	0
413-420	(K)QNCDQFEK(L)	0		1011.42	2	3	2	1	0	1	0	0	0
310-318	(K)SHCIAEVEK(D)	0		1015.4877	3	3	2	2	2	2	0	0	0
588-597	(K)EACFAVEGPK(L)	0		1050.4925	2	2	1	0	0	0	0	0	0
223-232	(R)CASIQKFGER(A)	1		1138.5674	—	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2	pyroGlu	1185.6521	—	—	—	—	—	—	—	—	—
337-346	(K)DVCKNYQEAK(D)	1		1197.5568	—	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2		1202.6786	—	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1		1331.7174	—	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1	1Met-ox	1347.7123	—	—	—	—	—	—	—	—	—
89-100	(K)SLHTLFGDELCK(V)	0		1362.6722	2	3	2	2	2	2	2	0	2
286-297	(K)YICDNQDTISSK(L)	0		1386.6206	1	2	1	0	0	1	0	1	0
221-232	(R)LRCASIQKFGER(A)	2		1407.7525	—	—	—	—	—	—	—	—	—
223-235	(R)CASIQKFGERALK(A)	2		1450.7835	—	—	—	—	—	—	—	—	—
483-495	(R)LCVLHEKTPVSEK(V)	1		1482.7985	—	—	—	—	—	—	—	—	—
387-399	(K)DDPHACYSTVFDK(L)	0		1497.6315	2	2	2	2	1	0	2	0	0
139-151	(K)LKPDPNTLCDEFK(A)	0		1519.7461	2	2	2	2	2	2	2	0	0
198-211	(K)GACLLPKIETMREK(V)	2		1588.8549	—	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1	pyroGlu	1599.722	1	—	0	—	—	0	0	0	0
198-211	(K)GACLLPKIETMREK(V)	2	1Met-ox	1604.8499	—	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1		1616.7486	3	0	0	0	0	0	0	0	0
286-299	(K)YICDNQDTISSKLK(E)	1		1627.7996	—	—	—	—	—	—	—	—	—
469-482	(R)MPCTEDYLSLILNR(L)	0		1667.8131	2	2	2	2	1	0	0	0	0
469-482	(R)MPCTEDYLSLILNR(L)	0	1Met-ox	1683.8081	2	2	2	2	2	2	0	0	0
387-401	(K)DDPHACYSTVFDKLK(H)	1		1738.8105	—	—	—	—	—	—	—	—	—
483-498	(R)LCVLHEKTPVSEKVTK(C)	2		1811.0095	—	—	—	—	—	—	—	—	—
508-523	(R)RPCFSALTPDETYVPK(A)	0		1823.8996	3	4	2	2	2	1	2	0	0
123-138	(R)NECFLSHKDDSPDLPK(L)	1		1844.8483	3	3	3	3	3	0	3	0	0
529-544	(K)LFTFHADICTLPDTEK(Q)	0		1850.8993	2	3	2	1	0	0	0	0	0
281-297	(R)ADLAKYICDNQDTISSK(L)	1		1884.9008	3	3	0	0	0	0	0	0	0
89-105	(K)SLHTLFGDELCKVASLR(E)	1		1888.9949	0	2	—	—	—	—	—	—	—
139-155	(K)LKPDPNTLCDEFKADEK(K)	1		1962.9477	2	3	3	3	3	3	3	0	0
588-607	(K)EACFAVEGPKLVVSTQTALA(-)	1		2034.0576	—	—	—	—	—	—	—	—	—
139-156	(K)LKPDPNTLCDEFKADEKK(F)	2		2091.0427	3	4	0	0	0	0	0	0	0
281-299	(R)ADLAKYICDNQDTISSKLK(E)	2		2126.0798	—	—	—	—	—	—	—	—	—
529-547	(K)LFTFHADICTLPDTEKQIK(K)	1		2220.1369	—	—	—	—	—	—	—	—	—
402-420	(K)HLVDEPQNLIKQNCDQFEK(L)	1		2298.1183	—	—	—	—	—	—	—	—	—
529-548	(K)LFTFHADICTLPDTEKQIKK(Q)	2		2348.2319	—	—	—	—	—	—	—	—	—
131-151	(K)DDSPDLPKLKPDPNTLCDEFK(A)	1		2387.1435	—	—	—	—	—	—	—	—	—
319-340	(K)DAIPENLPPLTADFAEDKDVCK(N)	1		2401.1592	2	3	3	3	2	2	2	0	0
508-528	(R)RPCFSALTPDETYVPKAFDEK(L)	1		2414.1697	—	—	—	—	—	—	—	—	—
45-65	(K)GLVLIAFSQYLQQCPFDEHVK(L)	0		2435.2428	—	—	—	—	—	—	—	—	—

Residues	Sequence	Missed Cleavages	Mods.[†]	M.I. Mass[‡]	Ideal[§]	BSA Concentration[*] 30 ng	3 ng	1.5 ng	0.8 ng	0.3 ng	0.15 ng	0.08 ng	0.03 ng
524-544	(K)AFDEKLFTFHADICTLPDTEK(Q)	1		2441.1693	—	—	—	—	—	—	—	—	—
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1	pyroGlu	2455.1711	—	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2	pyroGlu	2467.1194	3	3	1	0	0	0	0	0	0
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1		2472.1976	—	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2		2484.146	—	—	—	—	—	—	—	—	—
400-420	(K)LKHLVDEPQNLIKQNCDQFEK(L)	2		2539.2973	—	—	—	—	—	—	—	—	—
337-359	(K)DVCKNYQEAKDAFLGSFLYEYSR(R)	2		2746.2817	—	—	—	—	—	—	—	—	—
524-547	(K)AFDEKLFTFHADICTLPDTEKQIK(K)	2		2810.4069	—	—	—	—	—	—	—	—	—
131-155	(K)DDSPDLPKLKPDPNTLCDEFKADEK(K)	2		2830.3451	—	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2	pyroGlu	2875.3832	—	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2		2892.4097	—	—	—	—	—	—	—	—	—
310-336	(K)SHCIAEVEKDAIPENLPPLTADFAEDK(D)	1		2952.4295	—	—	—	—	—	—	—	—	—
387-412	(K)DDPHACYSTVFDKLKHLVDEPQNLIK(Q)	2		3025.5088	—	—	—	—	—	—	—	—	—
319-346	(K)DAIPENLPPLTADFAEDKDVCKNYQEAK(D)	2		3134.4987	3	3	2	2	2	0	0	0	0
37-65	(K)DLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	1		3390.6827	—	—	—	—	—	—	—	—	—
45-75	(K)GLVLIAFSQYLQQCPFDEHVKLVNELTEFAK(T)	1		3579.8556	—	—	—	—	—	—	—	—	—
35-65	(R)FKDLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	2		3665.8461	—	—	—	—	—	—	—	—	—
402-433	(K)HLVDEPQNLIKQNCDQFEKLGEYGFQNALIVR(Y)	2		3758.8959	—	—	—	—	—	—	—	—	—
				Total ions:	58	64[¶]	42	37	31	22	17	1	2
				Total Peptides:	25	24	21	18	16	12	8	1	1
				Proportion of Ideal:		0.96	0.84	0.72	0.64	0.48	0.32	0.04	0.04

[*] Numbers indicate the number of ions found for the corresponding peptide. File names (.mzXML): 30ng BSA (ORB4704), 30ng BSA + lysate (ORB4797), 3ng BSA + lysate (ORB4709), 1.5ng BSA + lysate (ORB4796), 0.8ng BSA + lysate (ORB4795), 0.3ng BSA + lysate (ORB4708), 0.15ng BSA + lysate (ORB4794), 0.08ng BSA + lysate (ORB4793), 0.03ng BSA + lysate(ORB4707). BSA indicates dibromide-tagged BSA, lysate indicates the presence of 10μg Jurkat cell lysate.

[†] Modifications

[‡] Monoisotopic mass

[§] The number of ions found in a sample containing no background

[¶] The total number of ions found can be sensitive to specific files, and may include false positives in the sense that a non-real match could have been correlated to real matches, increasing the total number of ions found for a given peptide. The total number of peptides found is less sensitive to this sort of variation due to the graph theoretic construct used to reduce false positives, and for this reason, the number of peptides found is a more reliable measure

Table A-2: Detection rate of dichloride-tagged BSA peptides in MS experiments where the indicated amount of tagged BSA was spiked into 10 μg of Jurkat cell lysate before tryptic digestion and LC-MS analysis on an LTQ-Orbitrap XL (resolution=60,000) mass spectrometer.

Residues	Sequence	Missed Cleavages	Mods.[†]	M.I. Mass[‡]	Ideal[§]	BSA Concentration[*] 3 ng	1.5 ng	0.8 ng	0.3 ng	0.15 ng	0.08 ng	0.03 ng
223-228	(R)CASIQK(F)	0		649.3338	2	0	0	0	0	0	0	0
198-204	(K)GACLLPK(I)	0		701.4015	2	2	0	0	0	0	0	0
483-489	(R)LCVLHEK(T)	0		841.46	2	2	0	0	0	0	0	0

Residues	Sequence	Missed Cleavages	Mods.[†]	M.I. Mass[‡]	Ideal[§]	BSA Concentration[*] 3 ng	1.5 ng	0.8 ng	0.3 ng	0.15 ng	0.08 ng	0.03 ng
221-228	(R)LRCASIQK(F)	1		918.5189	—	—	—	—	—	—	—	—
123-130	(R)NECFLSHK(D)	0		977.4509	3	2	0	0	0	0	0	0
413-420	(K)QNCDQFEK(L)	0	pyroGlu	994.3935	2	0	0	0	0	0	0	0
413-420	(K)QNCDQFEK(L)	0		1011.42	2	0	0	0	0	0	0	0
310-318	(K)SHCIAEVEK(D)	0		1015.4877	3	3	0	0	0	0	0	0
588-597	(K)EACFAVEGPK(L)	0		1050.4925	2	0	0	0	0	0	0	0
223-232	(R)CASIQKFGER(A)	1		1138.5674	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2	pyroGlu	1185.6521	—	—	—	—	—	—	—	—
337-346	(K)DVCKNYQEAK(D)	1		1197.5568	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2		1202.6786	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1		1331.7174	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1	1Met-ox	1347.7123	—	—	—	—	—	—	—	—
89-100	(K)SLHTLFGDELCK(V)	0		1362.6722	2	2	3	2	2	0	0	0
286-297	(K)YICDNQDTISSK(L)	0		1386.6206	1	0	0	0	0	0	0	0
221-232	(R)LRCASIQKFGER(A)	2		1407.7525	—	—	—	—	—	—	—	—
223-235	(R)CASIQKFGERALK(A)	2		1450.7835	—	—	—	—	—	—	—	—
483-495	(R)LCVLHEKTPVSEK(V)	1		1482.7985	—	—	—	—	—	—	—	—
387-399	(K)DDPHACYSTVFDK(L)	0		1497.6315	2	2	2	2	0	0	0	0
139-151	(K)LKPDPNTLCDEFK(A)	0		1519.7461	2	2	2	2	2	2	0	0
198-211	(K)GACLLPKIETMREK(V)	2		1588.8549	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1	pyroGlu	1599.722	—	—	—	—	—	—	—	—
198-211	(K)GACLLPKIETMREK(V)	2	1Met-ox	1604.8499	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1		1616.7486	—	—	—	—	—	—	—	—
286-299	(K)YICDNQDTISSKLK(E)	1		1627.7996	—	—	—	—	—	—	—	—
469-482	(R)MPCTEDYLSLILNR(L)	0		1667.8131	—	—	—	—	—	—	—	—
469-482	(R)MPCTEDYLSLILNR(L)	0	1Met-ox	1683.8081	1	0	0	0	0	0	0	0
387-401	(K)DDPHACYSTVFDKLK(H)	1		1738.8105	—	—	—	—	—	—	—	—
483-498	(R)LCVLHEKTPVSEKVTK(C)	2		1811.0095	—	—	—	—	—	—	—	—
508-523	(R)RPCFSALTPDETYVPK(A)	0		1823.8996	2	0	0	0	0	0	0	0
123-138	(R)NECFLSHKDDSPDLPK(L)	1		1844.8483	3	3	3	3	3	3	2	0
529-544	(K)LFTFHADICTLPDTEK(Q)	0		1850.8993	2	0	0	0	0	0	0	0
281-297	(R)ADLAKYICDNQDTISSK(L)	1		1884.9008	—	—	—	—	—	—	—	—
89-105	(K)SLHTLFGDELCKVASLR(E)	1		1888.9949	—	—	—	—	—	—	—	—
139-155	(K)LKPDPNTLCDEFKADEK(K)	1		1962.9477	3	3	3	3	3	3	2	0
588-607	(K)EACFAVEGPKLVVSTQTALA(-)	1		2034.0576	—	—	—	—	—	—	—	—
139-156	(K)LKPDPNTLCDEFKADEKK(F)	2		2091.0427	—	—	—	—	—	—	—	—
281-299	(R)ADLAKYICDNQDTISSKLK(E)	2		2126.0798	—	—	—	—	—	—	—	—
529-547	(K)LFTFHADICTLPDTEKQIK(K)	1		2220.1369	—	—	—	—	—	—	—	—
402-420	(K)HLVDEPQNLIKQNCDQFEK(L)	1		2298.1183	—	—	—	—	—	—	—	—
529-548	(K)LFTFHADICTLPDTEKQIKK(Q)	2		2348.2319	—	—	—	—	—	—	—	—
131-151	(K)DDSPDLPKLKPDPNTLCDEFK(A)	1		2387.1435	—	—	—	—	—	—	—	—
319-340	(K)DAIPENLPPLTADFAEDKDVCK(N)	1		2401.1592	2	0	3	0	0	0	0	0
508-528	(R)RPCFSALTPDETYVPKAFDEK(L)	1		2414.1697	—	—	—	—	—	—	—	—
45-65	(K)GLVLIAFSQYLQQCPFDEHVK(L)	0		2435.2428	—	—	—	—	—	—	—	—
524-544	(K)AFDEKLFTFHADICTLPDTEK(Q)	1		2441.1693	2	0	0	0	0	0	0	0
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1	pyroGlu	2455.1711	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2	pyroGlu	2467.1194	—	—	—	—	—	—	—	—
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1		2472.1976	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2		2484.146	—	—	—	—	—	—	—	—
400-420	(K)LKHLVDEPQNLIKQNCDQFEK(L)	2		2539.2973	—	—	—	—	—	—	—	—
337-359	(K)DVCKNYQEAKDAFLGSFLYEYSR(R)	2		2746.2817	—	—	—	—	—	—	—	—
524-547	(K)AFDEKLFTFHADICTLPDTEKQIK(K)	2		2810.4069	—	—	—	—	—	—	—	—

						BSA Concentration*						
Residues	**Sequence**	**Missed Cleavages**	**Mods.**[†]	**M.I. Mass**[‡]	**Ideal**[§]	**3 ng**	**1.5 ng**	**0.8 ng**	**0.3 ng**	**0.15 ng**	**0.08 ng**	**0.03 ng**
131-155	(K)DDSPDLPKLKPDPNTLCDEFKADEK(K)	2		2830.3451	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2	pyroGlu	2875.3832	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2		2892.4097	—	—	—	—	—	—	—	—
310-336	(K)SHCIAEVEKDAIPENLPPLTADFAEDK(D)	1		2952.4295	—	—	—	—	—	—	—	—
387-412	(K)DDPHACYSTVFDKLKHLVDEPQNLIK(Q)	2		3025.5088	—	—	—	—	—	—	—	—
319-346	(K)DAIPENLPPLTADFAEDKDVCKNYQEAK(D)	2		3134.4987	—	—	—	—	—	—	—	—
37-65	(K)DLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	1		3390.6827	—	—	—	—	—	—	—	—
45-75	(K)GLVLIAFSQYLQQCPFDEHVKLVNELTEFAK(T)	1		3579.8556	—	—	—	—	—	—	—	—

Residues	Sequence	Missed Cleavages	Mods.†	M.I. Mass‡	Ideal§	BSA Concentration* 3 ng	1.5 ng	0.8 ng	0.3 ng	0.15 ng	0.08 ng	0.03 ng
35-65	(R)FKDLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	2		3665.8461	—	—	—	—	—	—	—	—
402-433	(K)HLVDEPQNLIKQNCDQFEKLGEYGFQNALIVR(Y)	2		3758.8959	—	—	—	—	—	—	—	—
				Total ions:	40	21	16	12	10	8	4	0
				Total Peptides:	19	9	6	5	4	3	2	0
				Proportion of Ideal:		0.47	0.31	0.26	0.21	0.16	0.11	0.0

* Numbers indicate the number of ions found for the corresponding peptide. File names (.mzXML): 30ng BSA (ORB4710), 3ng BSA + lysate (ORB4714), 1.5ng BSA + lysate (ORB4802), 0.8ng BSA + lysate (ORB4801), 0.3ng BSA + lysate (ORB4879), 0.15ng BSA + lysate (ORB4800), 0.08ng BSA + lysate (ORB4799), 0.03ng BSA + lysate(ORB4798). BSA indicates dichloride-tagged BSA, lysate indicates the presence of 10μg Jurkat cell lysate.

† Modifications

‡ Monoisotopic mass

§ The number of ions found in a sample containing no background

Table A-3: Detection rate of monobromide-tagged BSA peptides in MS experiments where the indicated amount of tagged BSA was spiked into 10 μg of Jurkat cell lysate before tryptic digestion and LC-MS analysis on an LTQ-Orbitrap XL (resolution=60,000) mass spectrometer.

						BSA Concentration*						
Residues	**Sequence**	**Missed Cleavages**	**Mods.†**	**M.I. Mass‡**	**Ideal§**	**3 ng**	**1.5 ng**	**0.8 ng**	**0.3 ng**	**0.15 ng**	**0.08 ng**	**0.03 ng**
223-228	(R)CASIQK(F)	0		649.3338	2	2	0	0	0	0	0	0
198-204	(K)GACLLPK(I)	0		701.4015	2	2	2	2	2	0	0	0
483-489	(R)LCVLHEK(T)	0		841.46	2	2	0	0	0	0	0	0
221-228	(R)LRCASIQK(F)	1		918.5189	2	0	0	0	0	0	0	0
123-130	(R)NECFLSHK(D)	0		977.4509	3	0	0	0	0	0	0	0
413-420	(K)QNCDQFEK(L)	0	pyroGlu	994.3935	3	0	2	0	0	0	0	0
413-420	(K)QNCDQFEK(L)	0		1011.42	2	2	0	0	0	0	0	0
310-318	(K)SHCIAEVEK(D)	0		1015.4877	3	3	0	0	0	0	0	0
588-597	(K)EACFAVEGPK(L)	0		1050.4925	—	—	—	—	—	—	—	—
223-232	(R)CASIQKFGER(A)	1		1138.5674	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2	pyroGlu	1185.6521	—	—	—	—	—	—	—	—
337-346	(K)DVCKNYQEAK(D)	1		1197.5568	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2		1202.6786	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1		1331.7174	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1	1Met-ox	1347.7123	—	—	—	—	—	—	—	—
89-100	(K)SLHTLFGDELCK(V)	0		1362.6722	2	0	0	0	0	0	0	0
286-297	(K)YICDNQDTISSK(L)	0		1386.6206	2	0	0	0	0	0	0	0
221-232	(R)LRCASIQKFGER(A)	2		1407.7525	—	—	—	—	—	—	—	—
223-235	(R)CASIQKFGERALK(A)	2		1450.7835	—	—	—	—	—	—	—	—
483-495	(R)LCVLHEKTPVSEK(V)	1		1482.7985	—	—	—	—	—	—	—	—
387-399	(K)DDPHACYSTVFDK(L)	0		1497.6315	2	0	0	0	0	0	0	0

						BSA Concentration*						
Residues	**Sequence**	**Missed Cleavages**	**Mods.**[†]	**M.I. Mass**[‡]	**Ideal**[§]	**3 ng**	**1.5 ng**	**0.8 ng**	**0.3 ng**	**0.15 ng**	**0.08 ng**	**0.03 ng**
139-151	(K)LKPDPNTLCDEFK(A)	0		1519.7461	2	0	0	0	0	0	0	0
198-211	(K)GACLLPKIETMREK(V)	2		1588.8549	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1	pyroGlu	1599.722	—	—	—	—	—	—	—	—
198-211	(K)GACLLPKIETMREK(V)	2	1Met-ox	1604.8499	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1		1616.7486	3	0	0	0	0	0	0	0
286-299	(K)YICDNQDTISSKLK(E)	1		1627.7996	—	—	—	—	—	—	—	—
469-482	(R)MPCTEDYLSLILNR(L)	0		1667.8131	2	0	0	0	0	0	0	0
469-482	(R)MPCTEDYLSLILNR(L)	0	1Met-ox	1683.8081	2	0	0	0	0	0	0	0
387-401	(K)DDPHACYSTVFDKLK(H)	1		1738.8105	—	—	—	—	—	—	—	—
483-498	(R)LCVLHEKTPVSEKVTK(C)	2		1811.0095	—	—	—	—	—	—	—	—
508-523	(R)RPCFSALTPDETYVPK(A)	0		1823.8996	3	0	0	0	0	0	0	0
123-138	(R)NECFLSHKDDSPDLPK(L)	1		1844.8483	3	0	0	0	0	0	0	0
529-544	(K)LFTFHADICTLPDTEK(Q)	0		1850.8993	2	0	0	0	0	0	0	0
281-297	(R)ADLAKYICDNQDTISSK(L)	1		1884.9008	—	—	—	—	—	—	—	—
89-105	(K)SLHTLFGDELCKVASLR(E)	1		1888.9949	—	—	—	—	—	—	—	—
139-155	(K)LKPDPNTLCDEFKADEK(K)	1		1962.9477	3	0	0	0	0	0	0	0
588-607	(K)EACFAVEGPKLVVSTQTALA(-)	1		2034.0576	—	—	—	—	—	—	—	—
139-156	(K)LKPDPNTLCDEFKADEKK(F)	2		2091.0427	—	—	—	—	—	—	—	—
281-299	(R)ADLAKYICDNQDTISSKLK(E)	2		2126.0798	—	—	—	—	—	—	—	—
529-547	(K)LFTFHADICTLPDTEKQIK(K)	1		2220.1369	—	—	—	—	—	—	—	—
402-420	(K)HLVDEPQNLIKQNCDQFEK(L)	1		2298.1183	—	—	—	—	—	—	—	—
529-548	(K)LFTFHADICTLPDTEKQIKK(Q)	2		2348.2319	—	—	—	—	—	—	—	—
131-151	(K)DDSPDLPKLKPDPNTLCDEFK(A)	1		2387.1435	—	—	—	—	—	—	—	—
319-340	(K)DAIPENLPPLTADFAEDKDVCK(N)	1		2401.1592	3	0	0	0	0	0	0	0
508-528	(R)RPCFSALTPDETYVPKAFDEK(L)	1		2414.1697	1	0	0	0	0	0	0	0
45-65	(K)GLVLIAFSQYLQQCPFDEHVK(L)	0		2435.2428	3	3	3	3	2	3	0	0
524-544	(K)AFDEKLFTFHADICTLPDTEK(Q)	1		2441.1693	—	—	—	—	—	—	—	—
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1	pyroGlu	2455.1711	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2	pyroGlu	2467.1194	2	0	0	0	0	0	0	0
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1		2472.1976	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2		2484.146	2	2	0	3	0	0	0	0
400-420	(K)LKHLVDEPQNLIKQNCDQFEK(L)	2		2539.2973	—	—	—	—	—	—	—	—
337-359	(K)DVCKNYQEAKDAFLGSFLYEYSR(R)	2		2746.2817	—	—	—	—	—	—	—	—
524-547	(K)AFDEKLFTFHADICTLPDTEKQIK(K)	2		2810.4069	—	—	—	—	—	—	—	—
131-155	(K)DDSPDLPKLKPDPNTLCDEFKADEK(K)	2		2830.3451	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2	pyroGlu	2875.3832	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2		2892.4097	—	—	—	—	—	—	—	—
310-336	(K)SHCIAEVEKDAIPENLPPLTADFAEDK(D)	1		2952.4295	—	—	—	—	—	—	—	—
387-412	(K)DDPHACYSTVFDKLKHLVDEPQNLIK(Q)	2		3025.5088	—	—	—	—	—	—	—	—
319-346	(K)DAIPENLPPLTADFAEDKDVCKNYQEAK(D)	2		3134.4987	—	—	—	—	—	—	—	—
37-65	(K)DLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	1		3390.6827	—	—	—	—	—	—	—	—
45-75	(K)GLVLIAFSQYLQQCPFDEHVKLVNELTEFAK(T)	1		3579.8556	—	—	—	—	—	—	—	—
35-65	(R)FKDLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	2		3665.8461	—	—	—	—	—	—	—	—
402-433	(K)HLVDEPQNLIKQNCDQFEKLGEYGFQNALIVR(Y)	2		3758.8959	—	—	—	—	—	—	—	—
				Total ions:	56	16	7	8	4	3	0	0
				Total Peptides:	24	7	3	3	2	1	0	0
				Proportion of Ideal:		0.28	0.13	0.13	0.08	0.04	0.0	0.0

						BSA Concentration*					
Residues	Sequence	Missed Cleavages	Mods.†	M.I. Mass‡	Ideal§	3 ng	1.5 ng	0.8 ng	0.3 ng	0.15 ng	0.08 ng	0.03 ng

[*] Numbers indicate the number of ions found for the corresponding peptide. File names (.mzXML): 30ng BSA (ORB4803), 3ng BSA + lysate (ORB4878), 1.5ng BSA + lysate (ORB4809), 0.8ng BSA + lysate (ORB4808), 0.3ng BSA + lysate (ORB4807), 0.15ng BSA + lysate (ORB4806), 0.08ng BSA + lysate (ORB4805), 0.03ng BSA + lysate(ORB4804). BSA indicates monobromide-tagged BSA, lysate indicates the presence of 10μg Jurkat cell lysate.

[†] Modifications

[‡] Monoisotopic mass

[§] The number of ions found in a sample containing no background

Table A-4: Detection rate of dibromide-tagged BSA peptides in MS experiments where the indicated amount of tagged BSA was spiked into 15mg of Jurkat cell lysate before tryptic digestion and LC-MS analysis using a quadrupole/time-of-life (Waters Q-ToF Premier; resolution=10,000) mass spectrometer.

						BSA Concentration*						
Residues	Sequence	Missed Cleavages	Mods.†	M.I. Mass‡	Ideal§	3 ng	1.5 ng	0.8 ng	0.3 ng	0.15 ng	0.08 ng	0.03 ng
223-228	(R)CASIQK(F)	0		649.3338	2	2	2	2	2	2	0	2
198-204	(K)GACLLPK(I)	0		701.4015	2	2	2	2	2	2	0	0
483-489	(R)LCVLHEK(T)	0		841.46	3	3	3	3	3	2	0	0
221-228	(R)LRCASIQK(F)	1		918.5189	—	—	—	—	—	—	—	—
123-130	(R)NECFLSHK(D)	0		977.4509	3	3	3	2	0	0	0	0
413-420	(K)QNCDQFEK(L)	0	pyroGlu	994.3935	2	2	2	2	0	0	0	0
413-420	(K)QNCDQFEK(L)	0		1011.42	2	0	2	0	0	0	0	0
310-318	(K)SHCIAEVEK(D)	0		1015.4877	3	3	3	3	3	2	0	0
588-597	(K)EACFAVEGPK(L)	0		1050.4925	2	2	2	2	2	2	2	2
223-232	(R)CASIQKFGER(A)	1		1138.5674	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2	pyroGlu	1185.6521	—	—	—	—	—	—	—	—
337-346	(K)DVCKNYQEAK(D)	1		1197.5568	—	—	—	—	—	—	—	—
219-228	(R)QRLRCASIQK(F)	2		1202.6786	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1		1331.7174	—	—	—	—	—	—	—	—
198-209	(K)GACLLPKIETMR(E)	1	1Met-ox	1347.7123	—	—	—	—	—	—	—	—
89-100	(K)SLHTLFGDELCK(V)	0		1362.6722	2	2	2	2	2	2	0	0
286-297	(K)YICDNQDTISSK(L)	0		1386.6206	1	1	1	1	1	1	1	0
221-232	(R)LRCASIQKFGER(A)	2		1407.7525	—	—	—	—	—	—	—	—
223-235	(R)CASIQKFGERALK(A)	2		1450.7835	—	—	—	—	—	—	—	—
483-495	(R)LCVLHEKTPVSEK(V)	1		1482.7985	—	—	—	—	—	—	—	—
387-399	(K)DDPHACYSTVFDK(L)	0		1497.6315	2	2	2	0	2	2	0	0
139-151	(K)LKPDPNTLCDEFK(A)	0		1519.7461	2	2	2	2	2	0	0	0
198-211	(K)GACLLPKIETMREK(V)	2		1588.8549	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1	pyroGlu	1599.722	1	0	0	0	0	0	0	0
198-211	(K)GACLLPKIETMREK(V)	2	1Met-ox	1604.8499	—	—	—	—	—	—	—	—
118-130	(K)QEPERNECFLSHK(D)	1		1616.7486	3	3	0	0	0	0	0	0

						BSA Concentration*						
Residues	**Sequence**	**Missed Cleavages**	**Mods.**†	**M.I. Mass**‡	**Ideal**§	**3 ng**	**1.5 ng**	**0.8 ng**	**0.3 ng**	**0.15 ng**	**0.08 ng**	**0.03 ng**
286-299	(K)YICDNQDTISSKLK(E)	1		1627.7996	—	—	—	—	—	—	—	—
469-482	(R)MPCTEDYLSLILNR(L)	0		1667.8131	2	2	2	0	2	0	0	0
469-482	(R)MPCTEDYLSLILNR(L)	0	1Met-ox	1683.8081	2	2	0	0	0	0	0	0
387-401	(K)DDPHACYSTVFDKLK(H)	1		1738.8105	—	—	—	—	—	—	—	—
483-498	(R)LCVLHEKTPVSEKVTK(C)	2		1811.0095	—	—	—	—	—	—	—	—
508-523	(R)RPCFSALTPDETYVPK(A)	0		1823.8996	3	2	2	2	2	2	0	0
123-138	(R)NECFLSHKDDSPDLPK(L)	1		1844.8483	3	3	3	2	0	0	0	0
529-544	(K)LFTFHADICTLPDTEK(Q)	0		1850.8993	2	2	2	0	0	0	0	0
281-297	(R)ADLAKYICDNQDTISSK(L)	1		1884.9008	3	0	0	0	0	0	0	0
89-105	(K)SLHTLFGDELCKVASLR(E)	1		1888.9949	0	—	—	—	—	—	—	—
139-155	(K)LKPDPNTLCDEFKADEK(K)	1		1962.9477	2	3	3	3	2	2	0	0
588-607	(K)EACFAVEGPKLVVSTQTALA(-)	1		2034.0576	—	—	—	—	—	—	—	—
139-156	(K)LKPDPNTLCDEFKADEKK(F)	2		2091.0427	3	0	0	0	0	0	0	0
281-299	(R)ADLAKYICDNQDTISSKLK(E)	2		2126.0798	—	—	—	—	—	—	—	—
529-547	(K)LFTFHADICTLPDTEKQIK(K)	1		2220.1369	—	—	—	—	—	—	—	—
402-420	(K)HLVDEPQNLIKQNCDQFEK(L)	1		2298.1183	—	—	—	—	—	—	—	—
529-548	(K)LFTFHADICTLPDTEKQIKK(Q)	2		2348.2319	—	—	—	—	—	—	—	—
131-151	(K)DDSPDLPKLKPDPNTLCDEFK(A)	1		2387.1435	—	—	—	—	—	—	—	—
319-340	(K)DAIPENLPPLTADFAEDKDVCK(N)	1		2401.1592	2	2	2	0	0	2	0	0
508-528	(R)RPCFSALTPDETYVPKAFDEK(L)	1		2414.1697	—	—	—	—	—	—	—	—
45-65	(K)GLVLIAFSQYLQQCPFDEHVK(L)	0		2435.2428	—	—	—	—	—	—	—	—
524-544	(K)AFDEKLFTFHADICTLPDTEK(Q)	1		2441.1693	—	—	—	—	—	—	—	—
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1	pyroGlu	2455.1711	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2	pyroGlu	2467.1194	3	0	0	0	0	0	0	0
413-433	(K)QNCDQFEKLGEYGFQNALIVR(Y)	1		2472.1976	—	—	—	—	—	—	—	—
118-138	(K)QEPERNECFLSHKDDSPDLPK(L)	2		2484.146	—	—	—	—	—	—	—	—
400-420	(K)LKHLVDEPQNLIKQNCDQFEK(L)	2		2539.2973	—	—	—	—	—	—	—	—
337-359	(K)DVCKNYQEAKDAFLGSFLYEYSR(R)	2		2746.2817	—	—	—	—	—	—	—	—
524-547	(K)AFDEKLFTFHADICTLPDTEKQIK(K)	2		2810.4069	—	—	—	—	—	—	—	—
131-155	(K)DDSPDLPKLKPDPNTLCDEFKADEK(K)	2		2830.3451	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2	pyroGlu	2875.3832	—	—	—	—	—	—	—	—
413-436	(K)QNCDQFEKLGEYGFQNALIVRYTR(K)	2		2892.4097	—	—	—	—	—	—	—	—
310-336	(K)SHCIAEVEKDAIPENLPPLTADFAEDK(D)	1		2952.4295	—	—	—	—	—	—	—	—
387-412	(K)DDPHACYSTVFDKLKHLVDEPQNLIK(Q)	2		3025.5088	—	—	—	—	—	—	—	—
319-346	(K)DAIPENLPPLTADFAEDKDVCKNYQEAK(D)	2		3134.4987	3	2	2	0	0	0	0	0
37-65	(K)DLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	1		3390.6827	—	—	—	—	—	—	—	—
45-75	(K)GLVLIAFSQYLQQCPFDEHVKLVNELTEFAK(T)	1		3579.8556	—	—	—	—	—	—	—	—
35-65	(R)FKDLGEEHFKGLVLIAFSQYLQQCPFDEHVK(L)	2		3665.8461	—	—	—	—	—	—	—	—
402-433	(K)HLVDEPQNLIKQNCDQFEKLGEYGFQNALIVR(Y)	2		3758.8959	—	—	—	—	—	—	—	—
				Total ions:	58	45	42	27	25	21	3	4
				Total Peptides:	25	20	19	13	12	11	2	2
				Proportion of Ideal:		0.80	0.76	0.52	0.48	0.44	0.08	0.08

						BSA Concentration*						
Residues	Sequence	Missed Cleavages	Mods.†	M.I. Mass‡	Ideal§	3 ng	1.5 ng	0.8 ng	0.3 ng	0.15 ng	0.08 ng	0.03 ng

* Numbers indicate the number of ions found for the corresponding peptide. File names (.mzXML): 30ng BSA (QT12002), 3ng BSA + lysate (QT12001), 1.5ng BSA + lysate (QT12000), 0.8ng BSA + lysate (QR11999), 0.3ng BSA + lysate (QT11998), 0.15ng BSA + lysate (QT11997), 0.08ng BSA + lysate (QT11996), 0.03ng BSA + lysate(QT11995). BSA indicates dibromide-tagged BSA, lysate indicates the presence of 10μg Jurkat cell lysate.

† Modifications

‡ Monoisotopic mass

§ The number of ions found in a sample containing no background

appendix b

Computer source code

B.1 Overview and conventions

One of the primary frustrations in starting this project was the lack of cross-platform compatibility of existing code, making it difficult or impossible to reuse much of the code existing in the literature. As such, one of the primary considerations for software development was cross-comaptibility and code reusability. Given the computation complexity of some of the problems undertaken, execution speed was also a great concern. Finally, as many of the problems encountered require sophisticated data structures for optimal analysis, a language with robust constructs was required.

With the speed and features requirements in mind, I chose to write the majority of the software in C++ as it remains one of the fastest languages to do. The libraries included here make extensive use of the C++ STL, and a number of the classes themselves are derivatives of these libraries. Care was also taken to avoid platform-specific implementations, so third party libraries were avoided whenever possible. The single exception to this rule is the use of the Boost regular expression library used in the mzXML I/O library (subsection B.2) and in the handling of some user input. For libraries including the boost regular expression libraries, code must be compiled with the flag -lboost_regex. All code included here has been compiled under Linux (Ubuntu 8-10) on a number of hardware platforms and under OS 10.4 and 10.5. To date, only portions of the code have been compiled under Windows, though there should be minimal challenges if such a task were to be undertaken.

B.2 mzXML reading and writing

B.2.1 Description

A major hurdle in working with mass spectral data is storing the information in a way that can be easily shared and accessed. As the majority of instrument manufacturers do not release the specification of their proprietary file types, the community has adopted the mzXML filetype as a standard. In brief, the mzXML file type stores header information about the type of experiment (instrument used, ionization potentials, etc). Each scan is then stored as a separate XML block that includes information such as retention time, base peak m/z and intensity, MS level, and the raw data. The raw data is stored as a base64 encoded text block, with the precision specified in the scan header.

Finally, the file is closed with an index containing scan numbers and their respective offsets within the file. After this has been written, an indexOffset tag is written specifying the file offset of the beginning of the index tag. Together these two tags allow the file to be treated as a random access file—by reading the index first, scans can be read without having to read any other data within the file. This is essential as these files can be quite massive[*].

Finally, the file is closed with a tag including a sha-1 hash of the entire file (up to the end of the opening ⟨sha1⟩ tag). This is to ensure that the data has not been tampered with.

The C++ classes below provide a rapid and simple interface to working with the mzXML filetype. The main classes necessary or mzxml_reader.h with reading data, and mzxml_writer.h when writing data. The other classes are support classes that are invoked when necessary. These classes make use of the Boost regular expression (regex) library[†], which must be linked to the project at compile time, typically via the flag '-lboost_regex'.

B.2.2 C++ code

mzxml_lib.h

```
#ifndef MZXML_LIB_H_INCLUDED
#define MZXML_LIB_H_INCLUDED

#include <fstream>
#include <string>
#include <sstream>
#include <boost/regex.hpp>

#include "mzxml_types.h"

// for debugging:
#include <iostream>
```

[*] A typical mzXML file in profile mode from an Orbitrap is 1-3 GB

[†] Documentation and downloads can be found at www.boost.org

```
#ifndef __uint64
#ifdef  __int64
#define __uint64 unsigned __int64
#else
#define __uint64 unsigned long long int
#endif
#endif

namespace mzxml
{

  static const std::string base64_chars = "";
  base64_chars += "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  base64_chars += "abcdefghijklmnopqrstuvwxyz0123456789+/";
  static const int nullptr = 0;

  /**
   * @brief Converts any type to its string representation
   *
   * @param __x   The value
   * @return A string representation of the supplied value
   */
  template <typename _Tp>
  std::string stringify(_Tp &__x) {
    std::ostringstream __o;
    __o << __x;
    return __o.str();
  }

  /**
   * @brief Converts a string to a number
   *
   * @param  __s The string representation of the number
   * @return The value encoded in the supplied string
   */
  template <typename _Tp>
  _Tp val_from_string(const std::string __s) {
    std::istringstream __i(__s);
    _Tp __r;
    __i >> __r;
    return __r;
  }

  /**
   * @brief Obtains an offset from a string value
```

```
 *
 * This function is necessary since some mzxml conversion
 * programs use improper data types (I'm looking at you,
 * ReAdW), and thus large files may cause negative offset
 * values to be written due to integer overflows. This
 * function works backwards and converts negative values
 * back into the correct offset.
 *
 * @param  s A string representation of the offset
 * @return A size_t value of the corrected offset
 */
template <int i>
size_t offset_from_string(std::string s) {
  long long int __o;
  __o = val_from_string<long long int>(s);
  if( __o >= 0 )
    return (size_t)__o;

  // we'll assume they used long datatypes.
  // if they used normal integers, they're ”challenged.”
  // max(long): 2147483647, min(long): -2147483647

  return  2147483647 + (__o + 2147483649) - 20;
}

/**
 * @brief Converts a number into its hexadecimal
     representation
 *
 * @param  __x The value to be converted
 * @return A string of the hexadecimal representation of __x.
 */
template <typename _Tp>
std::string to_hex(_Tp &__x) {
  std::ostringstream __o;
  __o << std::hex << __x;
  return __o.str();
}

std::string pad_left(const std::string __s, size_t length,
    char __p = '␣');
std::string pad_left(std::string &__s, size_t length, char
    __p = '␣');

bool isBigEndian();
bool isLittleEndian();
```

```
    bool is_base64(unsigned char c);
    bool is_base64(char &c);

    std::string pack(float *pData, size_t size);
    std::string pack(float *pData);
    float * unpack(std::string input);
    float * unpack(std::string input, size_t &length);

    std::string base64_encode(unsigned char const* , unsigned int
        len);
    std::string base64_decode(std::string &s);
}
#endif                              // MZXML_LIB_H_INCLUDED
```

mzxml_lib.cpp

```
#include "mzxml_lib.h"

using namespace mzxml;

#define swapBytes(x) x = ((x & 0x000000ff) << 24) | ((x &
    0x0000ff00) << 8) | ((x & 0x00ff0000) >> 8) | ((x &
    0xff000000) >> 24);

/* you must redefine __MS_BIG_ENDIAN if you do not
 * have a big endian machine.. you can use the
 * test function given below, isBigEndian(), to
 * find out what you're running.
 */

#define __MS_BIG_ENDIAN false

/**
 * @brief Pads the supplied string the a given
 *        character until it is the desired length.
 *
 * @param __s      The string that is being padded
 * @param length   The desired string length
 * @param __p      The character the string is
 *                 padded with, defaults to spaces.
 * @return A padded string
 */
```

```
std::string mzxml::pad_left(const std::string __s, size_t
   length, char __p)
{
  std::string pad = "";
  while(__s.length() + pad.length() < length) {
    pad.push_back(__p);
  }

  return pad + __s;
}





/**
 * @brief Pads the supplied string the a given character until
 *        it is the desired length.
 *
 * @param __s      The string that is being padded
 * @param length   The desired string length
 * @param __p      The character the string is padded
 *                 with, defaults to spaces.
 * @return A padded string
 */
std::string mzxml::pad_left(std::string &__s, size_t length,
   char __p)
{
  std::string pad = "";
  while(__s.length() + pad.length() < length) {
    pad.push_back(__p);
  }

  return pad + __s;
}




/**
 * @brief Returns true if the current machine uses big
 *        endian byte ordering
 */
bool mzxml::isBigEndian()
{
  short int word = 0x0001;
  char *byte = (char *) &word;
  return (byte[0] ? false : true);
}



/**
 * @brief Returns true if the current machine
 *        uses little endian byte ordering
 */
bool mzxml::isLittleEndian()
{
  short int word = 0x0001;
  char *byte = (char *) &word;
  return (byte[0] ? true : false);
}




/**
 * @brief Returns true if the supplied character is a
 *        valid base64 character, false otherwise.
 */
bool mzxml::is_base64(unsigned char c)
{
  return (isalnum(c) || (c == '+') || (c == '/'));
}




/**
 * @brief Returns true if the supplied character
 *        is a valid base64 character, false otherwise.
 */
bool mzxml::is_base64(char &c)
{
  return (isalnum(c) || (c == '+') || (c == '/'));
}




/**
 * @brief Returns a base64 encoded string of the supplied data
 *
 * @param bytes_to_encode   The raw data that we would like to
 *                          encode
 * @param in_len            The length of the block of data
 * @return A base-64 encoded string of the data
 */
std::string mzxml::base64_encode(unsigned char const*
   bytes_to_encode, unsigned int in_len)
{
  std::string ret;
  int i = 0;
  int j = 0;
  unsigned char char_array_3[3];
  unsigned char char_array_4[4];

  while (in_len--) {
    char_array_3[i++] = *(bytes_to_encode++);
    if (i == 3) {
      char_array_4[0] = (char_array_3[0] & 0xfc) >> 2;
      char_array_4[1] = ((char_array_3[0] & 0x03) << 4)
                      + ((char_array_3[1] & 0xf0) >> 4);
      char_array_4[2] = ((char_array_3[1] & 0x0f) << 2)
                      + ((char_array_3[2] & 0xc0) >> 6);
      char_array_4[3] = char_array_3[2] & 0x3f;

      for(i = 0; (i <4) ; i++)
        ret += base64_chars[char_array_4[i]];
      i = 0;
    }
  }

  if (i) {
    for(j = i; j < 3; j++)
      char_array_3[j] = '\0';

    char_array_4[0] = (char_array_3[0] & 0xfc) >> 2;
    char_array_4[1] = ((char_array_3[0] & 0x03) << 4)
                    + ((char_array_3[1] & 0xf0) >> 4);
    char_array_4[2] = ((char_array_3[1] & 0x0f) << 2)
                    + ((char_array_3[2] & 0xc0) >> 6);
    char_array_4[3] = char_array_3[2] & 0x3f;

    for (j = 0; (j < i + 1); j++)
      ret += base64_chars[char_array_4[j]];

    while((i++ < 3))
```

```
      ret += '=';

  }
  return ret;
}




/**
 * @brief Returns a string of characters from a base-64
 *        encoded string
 *
 * @param encoded_string     The base-64 encoded data
 * @return A string of decoded characters
 */
std::string mzxml::base64_decode(std::string &encoded_string)
{
  int in_len = encoded_string.size();
  int i = 0;
  int j = 0;
  int in_ = 0;
  unsigned char char_array_4[4], char_array_3[3];
  std::string ret;

  while (in_len-- && ( encoded_string[in_] != '=')
              && is_base64(encoded_string[in_]))
  {
    char_array_4[i++] = encoded_string[in_]; in_++;
    if (i ==4) {
      for (i = 0; i <4; i++)
        char_array_4[i] = base64_chars.find(char_array_4[i]);

      char_array_3[0] = (char_array_4[0] << 2)
                      + ((char_array_4[1] & 0x30) >> 4);
      char_array_3[1] = ((char_array_4[1] & 0xf) << 4)
                      + ((char_array_4[2] & 0x3c) >> 2);
      char_array_3[2] = ((char_array_4[2] & 0x3) << 6)
                      + char_array_4[3];

      for (i = 0; (i < 3); i++)
        ret += char_array_3[i];
      i = 0;
    }
  }
  if (i) {
    for (j = i; j <4; j++)
      char_array_4[j] = 0;

    for (j = 0; j <4; j++)
      char_array_4[j] = base64_chars.find(char_array_4[j]);

    char_array_3[0] = (char_array_4[0] << 2)
                    + ((char_array_4[1] & 0x30) >> 4);
    char_array_3[1] = ((char_array_4[1] & 0xf) << 4)
                    + ((char_array_4[2] & 0x3c) >> 2);
    char_array_3[2] = ((char_array_4[2] & 0x3) << 6)
                    + char_array_4[3];

    for (j = 0; (j < i - 1); j++) ret += char_array_3[j];
  }

  return ret;
}




/**
 * @brief Packs an array of floating point values into a
 *        base64 encoded string
 *
 * @param pData   An array of floating point values
 * @param size    The number of elements in pData
 * @return A base64 encoded string of the packed data
 */
std::string mzxml::pack(float *pData, size_t size)
{
  std::string ret = "";

  unsigned int tmp = 0;
  short block_size = sizeof(*pData);
  unsigned char block[sizeof(*pData)];

  // load a string with bytes...s
  for(size_t i = 0; i < size; i++) {
    if(!__MS_BIG_ENDIAN) {
      memcpy(&tmp, &pData[i], sizeof(*pData));
      swapBytes(tmp);
      memcpy(&block, &tmp, block_size);
    }
    else {
      memcpy(&block, &pData[i], block_size);
    }
    ret.append((const char*)block, 4);
  }

  return base64_encode((const unsigned char*)ret.c_str(),
     ret.size());
}




/**
 * @brief Transforms an array of floating point values
 *        into a base64 encoded string
 *
 * @param pData An array of floating point values
 *        that is terminated by a -1 value
 * @return A base64 encoded string of the packed data
 */
std::string mzxml::pack(float *pData)
{
  std::string ret = "";

  unsigned int tmp = 0;
  short block_size = sizeof(*pData);
  unsigned char block[sizeof(*pData)];

  // load a string with bytes...
  int index = 0;
  while(pData[index] > -1) {
    if(!__MS_BIG_ENDIAN) {
      memcpy(&tmp, &pData[index++], sizeof(*pData));
      swapBytes(tmp);
      memcpy(&block, &tmp, block_size);
    }
    else {
      memcpy(&block, &pData[index++], block_size);
    }
    ret.append((const char*)block, 4);
  }

  return base64_encode((unsigned const char*)ret.c_str(),
     ret.size());
```

```
}




/**
 * @brief Unpacks a base64 encoded string into an
 *        array of floating point values
 *
 * @param  input   A base-64 encoded string
 * @return An array of floating point values terminated
 *         by two sequential -1 values
 */
float * mzxml::unpack(std::string input)
{
  float *pData = NULL;
  input = base64_decode(input);
  char BLANK = (char)0;

  // pad the input so we get enough characters...
  while(input.length() % 4 != 0) {
    input.append(BLANK, 1);
  }

  int index = 0;
  int size = input.length();
  pData = (float *)realloc(pData, size + 8 );
  unsigned char block[sizeof(*pData)];
  float fTmp = 0.0;

  for(size_t i = 0; i < input.length(); ) {
    // grab each character...
    if(!__MS_BIG_ENDIAN) {
      // swap the bytes as we pull them out.
      block[3] = (unsigned char)input.at(i++);
      block[2] = (unsigned char)input.at(i++);
      block[1] = (unsigned char)input.at(i++);
      block[0] = (unsigned char)input.at(i++);
    }
    else {
      block[0] = (unsigned char)input.at(i++);
      block[1] = (unsigned char)input.at(i++);
      block[2] = (unsigned char)input.at(i++);
      block[3] = (unsigned char)input.at(i++);
    }
    memcpy(&fTmp, &block , 4);
    pData[index++] = fTmp;
  }
  pData[index++] = -1.0;
  pData[index] = -1.0;
  // we add two 'kill' values because
  // these data are typically read in pairs, and bad
  // data can result in segfaults if not carefully
  // controlled for in other algorithms. This way,
  // we waste an extra 4 bytes but prevent this problem.
  return pData;
}




/**
 * @brief Unpacks a base64 encoded string into an array
 *        of floating point values
 *
 * @param input   A base-64 encoded string
 * @param length  A size type reference that will store
 *        the length of the array
 * @return An array of floating point values
 */
float * mzxml::unpack(std::string input, size_t &length)
{
  float *pData = NULL;
  input = base64_decode(input);
  char BLANK = (char)0;

  // pad the input so we get enough characters...
  while(input.length() % 4 != 0) {
    input.append(BLANK, 1);
  }

  int index = 0;
  int size = input.length();
  pData = (float *)realloc(pData, size + 8 );
  unsigned char block[sizeof(*pData)];
  float fTmp = 0.0;

  for(size_t i = 0; i < input.length(); ) {
    // grab each character...
    if(!__MS_BIG_ENDIAN) {
      // swap the bytes as we pull them out.
      block[3] = (unsigned char)input.at(i++);
      block[2] = (unsigned char)input.at(i++);
      block[1] = (unsigned char)input.at(i++);
      block[0] = (unsigned char)input.at(i++);
    }
    else {
      block[0] = (unsigned char)input.at(i++);
      block[1] = (unsigned char)input.at(i++);
      block[2] = (unsigned char)input.at(i++);
      block[3] = (unsigned char)input.at(i++);
    }
    memcpy(&fTmp, &block, 4);
    pData[index++] = fTmp;
  }

  length = index;
  return pData;
}
```

mzxml_reader.h

```
#ifndef MZXML_READER_H_INCLUDED
#define MZXML_READER_H_INCLUDED

#include <boost/regex.hpp>
#include <fstream>
#include <string>
#include <map>

#include ”mzxml_types.h”
#include ”mzxml_lib.h”

namespace mzxml
{

  using namespace std;

  /** @brief A basic mzXML reader
   *
   * At this point, this class assumes that
   * all data is stored as 32-bit precision (float) in
   * base64 data. Header information contains LCSM settings,
   * Rf times, etc, but does not yet include MALDI spot
```

```
 * coordinates, etc.
 */
class Reader
{
  public:
    Reader();
    Reader(char const* __file);
    Reader(std::string &__file);
    ~Reader();

    /* File opening and closing */
    bool open(std::string &f_name);
    bool open(const char *__file);
    void close();

    /* File reading operations  */
    bool read_index_offset();
    bool read_index();
    bool read_instrument_header();
    bool read_run_header();
    bool read_scan_header(unsigned int scanNum);
    bool scan_exists(unsigned int scanNum);
    bool read_parent_file_header();

    float *  read_scan(unsigned int scan_num);
    float *  read_scan(unsigned int scan_num, size_t &count);

    /* Access to header information */

    //!< Returns the curent instrument header
    inline InstrumentStruct get_instrument_header() {
      return __instrument;
    }

    //!< Returns the current run header
    inline RunHeaderStruct get_run_header() {
      return __run_header;
    }

    //!< Returns the current scan header
    inline ScanHeaderStruct get_scan_header() {
      return __scan_header;
    }

     //!< Returns the index offset of the current file.
    inline size_t get_index_offset() {
      return __index_offset;
    }

    //!< Returns the parent file information that has been
       read
    inline vector<ParentFileStruct> get_parent_file_header() {
      return __parent_files;
    }

    //!< Returns the number of scan offsets that have been
       read
    inline int get_scan_count() {
      return __index.size();
    }

    //!< Returns the saved checksum of the file
    inline string get_SHA1() {
      return checksum;
    }

    inline size_t file_size();

    //!< Returns the open status of the file
    inline bool is_open() {
      return __in.is_open();
    }

    bool validate();

  private:
    ifstream                 __in;
    size_t                   __index_offset;
    size_t                   __scan_count;
    std::map<unsigned int,size_t>        __index;
    InstrumentStruct         __instrument;
    RunHeaderStruct          __run_header;
    ScanHeaderStruct         __scan_header;
    ParentFileStruct         __parent_file;
    vector<ParentFileStruct>      __parent_files;
    string                   __f_name;
    string                   __checksum;
    void stringSplit(string s, vector<string> delims,
       vector<string> &ret);
    void stringSplit(string s, string delim, vector<string>
       &ret);
    void stripWhitespace(string &s);
```

```
      void stripChars(string &s, char * tokens, int tokSize);
      void stripChars(string &s, char token);
      void resetParams();
      string checksum;

      inline float time_modifier(std::string __s);

  };

}
#endif                    // MZXML_READER_H_INCLUDED
```

mzxml_reader.cpp

```
#include "mzxml_reader.h"

using namespace mzxml;
using namespace boost;
using namespace std;

#define LINE_SIZE 512    //!< The maximum size of a line read
#define BLOCK_SIZE 1024 //!< The size of character blocks we read

mzxml::Reader::Reader() : __index_offset(0), __scan_count(0) {}


mzxml::Reader::Reader(string &__file) : __index_offset(0), __scan_count(0)
{
  std::cout << "Initiating reader." << std::endl;
  open(__file);
}


mzxml::Reader::Reader(char const* __file) : __index_offset(0), __scan_count(0)
{
  open(__file);
}

```

```
mzxml::Reader::~Reader()
{
  __in.close();
}


void mzxml::Reader::close()
{
  __in.close();
}


/**
 * @brief Returns true if we have an index offset
 *        recorded for the supplied index
 */
bool mzxml::Reader::scan_exists(unsigned int ind)
{
  return (__index[ind]);
}




/**
 * @brief returns the size of the current file in bytes
 */
inline size_t Reader::file_size()
{
  if(__in.is_open()) {
    __in.seekg(0, ios::end);
    return __in.tellg();
  }
  else {
    return 0;
  }
}




/**
 * @brief Open an mzxml file. No checking is done to
 *        validate the format at this point
 *
 * @param __file    The name of the mzXML file to be opened
 * @return True if the file was able to be opened
 */
bool mzxml::Reader::open(string &__file)
{
  __f_name = __file;
  __in.open(__file.c_str(), ios::in);

  // do some basic file checking
  return __in.is_open();
}




/**
 * @brief Open an mzxml file. No checking is done to
 *        validate the format at this point
 *
 * @param __file    The name of the mzXML file to be opened
 * @return True if the file was able to be opened
 */
bool mzxml::Reader::open(const char *__file)
{
  __f_name = __file;
  __in.open(__file, ios::in);

  return __in.is_open();
}




/**
 * @brief Determine the index offset from the end of the file
 *
 * @return True if the index offset was able to be determined
 */
bool mzxml::Reader::read_index_offset()
{
  regex re("<\\s?indexOffset\\s?>(-?[0-9]+)</indexOffset");
  cmatch what;
  char memblock[BLOCK_SIZE]; // just long enough for one line.
  __in.seekg(-BLOCK_SIZE, ios::end);

  while(__in.getline(memblock, BLOCK_SIZE-1)) {
    if(regex_search(memblock, what, re)) {
      // we've found our match...
      __index_offset =
         offset_from_string<0>(string(what[1].first,
         what[1].second));
      return true;
    }
  }

  // we weren't able to find it...
  if(__in.fail()) {
    __in.clear();
  }
  return false;
}


/**
 * @brief Reads the index table from the end of the mzXML file
 *
 * read_index() finds the index located at the
 *              position indicated by the index offset, and then
 *              proceeds to read the index into the __index map,
 *              storing each offset indexed by the scan number
 *
 * @return True if the index was able to be read in completion,
 *              false otherwise
 */
bool mzxml::Reader::read_index()
{
  /* the index looks like:
     <index>
       <offset id=”#####”>#######</offset>
       <offset id=”#####”>#########</offset>
       .
       .
       .
     </index>
  */

  if(!__index_offset)
    read_index_offset();

  __in.seekg(__index_offset, ios::beg);
  char memblock[LINE_SIZE];

  // read until we find the line <index>
  regex _re_ind_start("<index");
  regex _re_ind_end("</index>");
  regex _re_offset("<offset\\s+id=\"(-?[0-9]+)\"\
\\s*>([0-9]+)\"?</offset");
  cmatch what;

  while(__in.getline(memblock, LINE_SIZE)) {
    if(regex_search(memblock, what, _re_ind_start))
      break;
  }

  if(__in.eof()) return false;
  unsigned int __s_no;
  size_t __offset;

  do {
    if(regex_search(memblock, what, _re_offset)) {
      // we have an offset...
      __s_no = val_from_string<unsigned
         int>(string(what[1].first, what[1].second));
      __offset = offset_from_string<0>(string(what[2].first,
                                              what[2].second));
      __index[__s_no] = __offset;
    }

    if(regex_search(memblock, what, _re_ind_end))
      break;

  } while (__in.getline(memblock, LINE_SIZE));

  // in the case of small files, the block size can be too big
  // and we can cause the file to fail even if we read the data
  // successfully. Reset the flags in this case so that we don't
  // have any problems later.
  if(__in.fail()) __in.clear();

  // if the </index> tag wasn't the last thing we read,
     something isn't right
  if(regex_search(memblock, what, _re_ind_end)) return true;

  return false;
}
```

```cpp
/**
 * @brief Reads the run header information from
 *        the current file. Reads in the scan count and
 *        the starting and ending times (in seconds)
 *
 * @return True if the header was read successfully,
 *         false otherwise.
 */
bool mzxml::Reader::read_run_header()
{
  // we need to get scanCount, startTime, and endTime
  if(!__in.is_open()) return false;

  regex __msr_start("<msRun\\s+");
  regex __msr_end("<msInstrument");

  cmatch what;
  __in.seekg(0, ios::beg);
  char memblock[LINE_SIZE];
  size_t count = 0;
  while(__in.getline(memblock, LINE_SIZE)) {
    if(regex_search(memblock, what, __msr_start))
      break;
    // a little bit smoother of a failure. We won't brick the
    // computer trying to read several gigs.
    if(++count > 10000) return false;
  }

  if(__in.eof()) return false;

  do {
    if(regex_search(memblock, what,
       regex("scanCount=\"(\\d+)\""))) {
      // we have the scan count...
      __run_header.scan_count = val_from_string<unsigned
         int>(string(what[1].first, what[1].second));
    }

    if(regex_search(memblock, what,
       regex("startTime=\"[a-zA-Z]*([0-9\\.]+)([sShHmM])\""))) {
      // we have the start time...
      __run_header.start_time =
         val_from_string<double>(string(what[1].first,
         what[1].second));
      __run_header.start_time *=
         time_modifier(string(what[2].first, what[2].second));
    }

    if(regex_search(memblock, what,
       regex("endTime=\"[a-zA-Z]*([0-9\\.]+)([sShHmM])\""))) {
      // we have the start time...
      __run_header.end_time =
         val_from_string<double>(string(what[1].first,
         what[1].second)) ;
      __run_header.end_time *=
         time_modifier(string(what[2].first, what[2].second));
    }

    if(regex_search(memblock, what, __msr_end))
      break;

  } while(__in.getline(memblock, LINE_SIZE));

  if(regex_search(memblock, what, __msr_end)) return true;
  return false;
}



/**
 * @brief Reads the instrument header information
 *        contained in the current file
 *
 * @return True if the header was able to be read,
 *         false otherwise
 */
bool mzxml::Reader::read_instrument_header()
{
  if(!__in.is_open()) return false;

  regex __ih_start("<msInstrument");
  regex __ih_end("</msInstrument");
  regex __ih("<ms\\S*\\s+category=\"(.*)\"\\s+\
value=\"(.*)\"\\s*/>");
  cmatch what;

  __in.seekg(0, ios::beg);
  char memblock[LINE_SIZE];
  while(__in.getline(memblock, LINE_SIZE)) {
    if(regex_search(memblock, what, __ih_start))
      break;
  }

  if(__in.eof())
    return false;

  do {
    if(regex_search(memblock, what, __ih)) {
      __instrument.data[string(what[1].first, what[1].second)]
          = string(what[2].first, what[2].second);
    }

    if(regex_search(memblock, what, __ih_end))
      break;
  } while (__in.getline(memblock, LINE_SIZE));

  if(regex_search(memblock, what, __ih_end)) return true;

  return false;
}




/**
 * @brief Reads the scan header for the opened file
 *
 * @param scan_num     The number of the scan header to be read
 * @return true if the header is read successfully, false if
 *         the header was not read or if the scan does not exist
 */
bool mzxml::Reader::read_scan_header(unsigned int scan_num)
{
  if(!__in.is_open()) return false;
  if(!scan_exists(scan_num)) return false;

  regex __scan_start(”<scan\\s+”);

  // stop at the beginning of the ’peaks’ list.
  regex __scan_end(”<peaks\\s+”);
  cmatch what;

  __in.seekg(__index[scan_num], ios::beg);
  char memblock[LINE_SIZE];
  while(__in.getline(memblock, LINE_SIZE)) {
    if(regex_search(memblock, what, __scan_start))
      break;
  }

  if(__in.eof()) return false;

  map<string, regex> __keys;

  do {
    // test key value pairs...
    if(regex_search(memblock, what, regex("num=\"([0-9]+)\"")))
      __scan_header.num = val_from_string<unsigned
         int>(string(what[1].first, what[1].second));

    if(regex_search(memblock, what,
       regex("msLevel=\"([0-9])\"")))
      __scan_header.ms_level = val_from_string<unsigned
         int>(string(what[1].first, what[1].second));

    if(regex_search(memblock, what, regex("polarity=\"(.*)\"")))
      __scan_header.polarity = string(what[1].first,
         what[1].second);

    if(regex_search(memblock, what,
       regex("scanType=\"(\\S+)\"")))
      __scan_header.scan_type = string(what[1].first,
         what[1].second);

    if(regex_search(memblock, what,
       regex("retentionTime=\"[a-zA-Z]\
*([0-9\\.]+)([sShHmM])?\""))) {
      __scan_header.retention_time =
         val_from_string<double>(string(what[1].first,
         what[1].second));
      __scan_header.retention_time *=
         time_modifier(string(what[2].first, what[2].second));
    }

    if(regex_search(memblock, what,
       regex("lowMz=\"([0-9\\.]+)\"")))
      __scan_header.low_mz =
         val_from_string<double>(string(what[1].first,
         what[1].second));

    if(regex_search(memblock, what,
       regex("highMz=\"([0-9\\.]+)\"")))
      __scan_header.high_mz =
         val_from_string<double>(string(what[1].first,
         what[1].second));

    if(regex_search(memblock,what,regex("basePeakMz=\\
"([0-9\\.]+)\"")))
      __scan_header.base_peak_mz =
         val_from_string<double>(string(what[1].first,
         what[1].second));

    if(regex_search(memblock, what,
       regex("basePeakIntensity=\"([0-9\\.]\
+([eE][+\\-]\\d+))\"")))
      __scan_header.base_peak_intensity =
         val_from_string<double>(string(what[1].first,
         what[1].second));

    if(regex_search(memblock, what,
       regex("totIonCurrent=\"([0-9\\.]+)\"")))
      __scan_header.total_ion_current =
         val_from_string<double>(string(what[1].first,
         what[1].second));

    if(regex_search(memblock, what,
       regex("peaksCount=\"(\\d+)\"")))
      __scan_header.peaks_count = val_from_string<unsigned
         int>(string(what[1].first, what[1].second));

    if(regex_search( memblock, what,
       regex("ionisationEnergy=\"(\\d+)\"")))
      __scan_header.ionization_energy =
         val_from_string<double>(string(what[1].first,
         what[1].second));

    if(regex_search(memblock, what, __scan_end))
      break;

  } while(__in.getline(memblock, LINE_SIZE));

  do {
    if(regex_search(memblock, what,
       regex("<precursorMz.*>([0-9\\.]+)</precursor")))
      __scan_header.precursor_mz =
         val_from_string<double>(string(what[1].first,
         what[1].second));

    if(regex_search(memblock, what, regex("<peaks")))
      break;
  } while(__in.getline(memblock, LINE_SIZE));

  // we can look for the optional precursor m/z
  // tag here, formatted as:
  // <precursorMz precursorIntensity="##">##.##</precursorMz>

  if(regex_search(memblock, what, regex("<peaks"))) return true;

  return false;
}



/**
 * @brief Returns a floating point modifier to convert
 *        minutes and hours to seconds
 *
 * @param __s A string match for the units portion of
 *          a number. Only the first character is read.
 * @return A floating point value to convert minutes
 *          or hours to seconds
 */
inline float mzxml::Reader::time_modifier(string __s)
{
  if(__s.length() < 1) return 1.0;
  float __r = 1.0;
  switch (__s[0]) {
    case 'h': __r = 3600.0;
    case 'H': __r = 3600.0;
    case 'm': __r = 60.0;
    case 'M': __r = 60.0;
    default: __r = 1.0;
  }
  return __r;
}



/**
 * @brief Reads the peak list from the current
 *        file at the given scan number
 *
 * @param scan_num The desired scan number
 *        to fetch peaks from
 * @return An array of floating point values in
 *         mz/intensity pairs terminated by -1.0 values
 */
float * mzxml::Reader::read_scan(unsigned int scan_num)
{
  if( (!__in.is_open()) || (!scan_exists(scan_num)) ) {
    float * __r = new float[2];
    __r[0] = -1.0;
    __r[1] = -1.0;
    return __r;
  }

  // sometimes the file can fail when reading the index
  // or index offset in the case of small files.
  if( __in.fail() ) __in.clear();

  __in.seekg(__index[scan_num], ios::beg);
  char memblock[BLOCK_SIZE];
  cmatch what;

  string __data = "";
  while(__in.read(memblock, BLOCK_SIZE-1)) {
    __data.append(memblock, BLOCK_SIZE-1);
    if(regex_search(__data.c_str(), what, regex("</peaks")))
      break;
  }

  if(regex_search(__data.c_str(), what,
     regex("<peaks.*>([a-zA-Z0-9+=\\/]*)</peaks"))) {
    __data = string(what[1].first, what[1].second);
    return unpack(__data);
  }
  else {
    // no data??
    float * __r = new float [2];
    __r[0] = -1.0;
    __r[1] = -1.0;
    return __r;
  }
}



/**
 * @brief Reads the peak list from the current
 *         file at the given scan number
 *
 * @param scan_num The desired scan number from
 *         which to retrieve the data
 * @param peaks_count A reference to a size type
 *         that will give number of values returned.
 * @return An array of floating point values in
 *         mz/intensity pairs
 */
float * mzxml::Reader::read_scan(unsigned int scan_num, size_t
   &peaks_count)
{
  if( (!__in.is_open()) || (!scan_exists(scan_num)) ) {
    peaks_count = 0;
    return (float *)nullptr;
  }

  // sometimes the file can fail when reading the index
  // or index offset in the case of small files.
  if(__in.fail() ) __in.clear();

  __in.seekg(__index[scan_num], ios::beg);

  char memblock[BLOCK_SIZE];
  cmatch what;

  string __data = "";
  string msg = "";

  while(__in.read(memblock, BLOCK_SIZE-1)) {
    __data.append(memblock, BLOCK_SIZE-1);
    //std::cout << memblock << "\n\n";

    // in case we've read only until after </
    if(regex_search(memblock, what, regex("</"))) {
      msg = "stopped after </";
      break;
    }       // if we stopped just before /peaks
    else if(regex_search(memblock, what, regex("/peaks>"))) {
      msg = "stopped after /peaks>";
      break;
    }
  }

  if(msg.length() < 1) {
    __data.append(memblock,__in.gcount());
  }

  if(regex_search(__data.c_str(), what,
     regex(">([a-zA-Z0-9+=\\/]*)<\\s*\\/"))) {
    __data = string(what[1].first, what[1].second);
    return unpack(__data, peaks_count);
  }
  else {
    peaks_count = 0;
    return (float*)nullptr;
  }
}
```

```
bool mzxml::Reader::read_parent_file_header()
{
  // This function was never needed, but code
  // was kept to meet the standard.
  return false;
}
```

mzxml_writer.h

```
#ifndef MZXML_WRITER_H_INCLUDED
#define MZXML_WRITER_H_INCLUDED

#include <boost/regex.hpp>
#include <fstream>
#include <string>
#include <map>
#include <vector>

#include "mzxml_types.h"
#include "mzxml_lib.h"
#include "mzxml_sha1.h"

namespace mzxml
{
  using namespace std;
  class Writer
  {
    public:
      Writer(){ }
      Writer(const char * f_name);
      Writer(string f_name);
      ~Writer();

      // file opening and closing
      bool open(const char * f_name);
      bool open(string f_name);
      bool close();
      bool is_open() {  //!< Returns true if the file is open
        return __o.is_open();
      }

      bool write_header(RunHeaderStruct &runHeader,
        InstrumentStruct &instrument,
        vector<ParentFileStruct>  &parentFiles);

      bool write_scan(ScanHeaderStruct &scanheader, float
         *pData);
      bool write_scan(ScanHeaderStruct &scanheader, float
         *pData, size_t &data_size);
      bool write_scan(ScanHeaderStruct &scanHeader,
         vector<float> vData);

    private:
      ofstream __o;
      vector<size_t> __index;
      string __f_name;

      bool __index_written;
      size_t __index_offset;
      unsigned short __current_scan;

      void init();
      bool write_index();
      bool write_checksum();

  };
} // end namespace
#endif                       // MZXML_WRITER_H_INCLUDED
```

mzxml_writer.cpp

```
#include "mzxml_writer.h"

using namespace mzxml;
using namespace std;

/**
 * @brief A basic constructor that opens the file for output
 *
 * @param f_name  The path to the file to be opened
 */
Writer::Writer(const char * f_name)
{
  open(f_name);
}


/**
 * @brief A basic constructor that opens the file for output
 *
 * @param f_name  The path to the file to be opened
 */
Writer::Writer(string f_name)
{
  open(f_name.c_str() );
}


/**
 * @brief A basic destructor that closes the file
 *        and ensures that the index is written
 */
Writer::~Writer()
{
  close();
}


/**
 * @brief Opens a file for output
 *
 * @param f_name  The path to the file to be opened.
 * @return true if a new file was opened, false if
 *         there were problems
 */
```

```
bool Writer::open(const char * f_name)
{
  if(__o.is_open())
    close();

  __o.open(f_name, ios::out);
  __f_name = f_name;
  init();
  return __o.is_open();
}


/**
 * @brief Opens a file for output
 *
 * @param f_name  The path to the file to be opened.
 * @return true if a new file was opened,
 *         false if there were problems
 */
bool Writer::open(string f_name)
{
  return open(f_name.c_str());
}


/**
 * @brief Closes the file (if open) and ensures
 *        that the index has been written.
 */
bool Writer::close()
{
  // we need to ensure that the index has been written, etc.
  write_index();
  __o.close();
  return true;
}


/**
 * @brief Initializes some state flags for the writing object
 */
void Writer::init()
{
  __index_offset = 0;
  __current_scan = 0;
  __index_written = false;
```

```
}

/**
 * @brief Writes the main scan header information
 *
 * @return Returns true if the writing occurred successfully,
 *         false if the writer is in a fail state
 */
bool Writer::write_header(RunHeaderStruct &run_header,
InstrumentStruct &instrument_header,
vector<ParentFileStruct> &parent_files)
{
   string s;
   /** the standard mzXML header. **/
   s = "<?xml␣version=\"1.0\"␣encoding=\"ISO-8859-1\"?>";
   s += "\n<mzXML";
   s += "\n␣xmlns=\"http://sashimi.sourceforge.net/";
   s += "schema_revision/mzXML_2.0\"";
   s += "\n␣xmlns:xsi=\"http://www.w3.org/2001";
   s += "/XMLSchema-instance\"";
   s += "\n␣
        xsi:schemaLocation=\"http://sashimi.sourceforge.net/";
   s += "schema_revision/mzXML_2.0␣";
   s += "http://sashimi.sourceforge.net/schema_revision/
␣␣s␣+=␣"mzXML_2.0/mzXML_idx_2.0.xsd\">";
   __o.write(s.c_str(), s.length());

   s = "\n␣<msRun␣scanCount=\"";
   s += stringify(run_header.scan_count);
   s += "\"";
   s += "\n\tstartTime=\"PT";
   s += stringify(run_header.start_time);
   s += "S\"";
   __o.write(s.c_str(), s.length());

   s = "\n\tendTime=\"PT";
   s += stringify(run_header.end_time);
   s += "S\">";
   __o.write(s.c_str(), s.length());

   // parent file information should go here..
   for(size_t i = 0; i < parent_files.size(); ++i) {
      s = "\n␣␣<parentFile␣fileName=\"";
      s += parent_files[i].file_name;
      s += "\"\n␣␣␣␣fileType=\"";
```

```
    s += parent_files[i].file_type;
    s += "\"\n    fileSha1=\"";
    s += parent_files[i].file_SHA1;
    s += "\"/>";
    __o.write(s.c_str(), s.length());
  }

  // write the instrument specifications
  // This is a set of key value pairs stored in a map,
  // so each tag should be <key category="key" value="value"/>
  map<string,string>::iterator _start, _end =
     instrument_header.data.end();
  s = "\n  <msInstrument>";
  __o.write(s.c_str(), s.length());
  for(_start = instrument_header.data.begin(); _start != _end;
     ++_start) {
    s = "\n    <";
    s += _start->first;
    s += " category=\"";
    s += _start->first;
    s += "\" value=\"";
    s += _start->second;
    s += "\"/>";
    __o.write(s.c_str(), s.length());
  }

  for(size_t i = 0; i < instrument_header.software.size(); ++i)
     {
    s = "\n    <software type=\"";
    s += instrument_header.software[i].type;
    s += "\"\n    name=\"";
    s += instrument_header.software[i].name;
    s += "\"\n    version=\"";
    s += instrument_header.software[i].version;
    s += "\"/>";
    __o.write(s.c_str(), s.length());
  }

  // aquisition software stuff should go here.
  s = "\n  </msInstrument>";
  __o.write(s.c_str(), s.length());

  return !__o.fail();
}
```

```
/**
 * @brief Writes a new scan to the open file
 *
 * @param scan_header A scan header structure
 *        with information on the current scan
 * @param pData  An array of floating point
 *        values terminated by a -1 value
 * @return true if the data was written successfully,
 *        false if there was a problem
 */
bool Writer::write_scan(ScanHeaderStruct &scan_header, float
   *pData)
{
  // the file isn't open yet.
  if(!__o.is_open())  return false;

  // we don't want to write any more scans
  // after the index has been written
  if(__index_written) return false;

  __o.seekp(0, ios::end);
  size_t __c_offset = __o.tellp();
  __index.push_back(__c_offset);

  // write the scan properties...
  float _bp_intensity = -1.0;
  float _bp_mz = 0.0;

  // locate the base peak:
  size_t index = 0;
  while(pData[index] > -1.0) {
    if(pData[index + 1] > _bp_intensity) {
      _bp_intensity = pData[index + 1];
      _bp_mz = pData[index];
    }
    index += 2;
  }

  // index now stores the number of mz/intensity pairs
  index /= 2;

  string s;

  s = "\n  <scan num=\"";
```

```
    s += stringify(++__current_scan);
    s += "\"\n␣␣msLevel=\"";
    s += stringify(scan_header.ms_level);
    s += "\"\n␣␣peaksCount=\"";
    s += stringify(index);
    s += "\"\n␣␣polarity=\"";
    s += scan_header.polarity;
    s += "\"\n␣␣scanType=\"";
    s += scan_header.scan_type;
    s += "\"\n␣␣retentionTime=\"PT";
    s += stringify(scan_header.retention_time);
    s += "\"\n␣␣lowMz=\"";
    s += stringify(scan_header.low_mz);
    s += "\"\n␣␣highMz=\"";
    s += stringify(scan_header.high_mz);
    s += "\"\n␣␣basePeakMz=\"";
    s += stringify(_bp_mz);
    s += "\"\n␣␣basePeakIntensity=\"";
    s += stringify(_bp_intensity);
    s += "\"\n␣␣totIonCurrent=\"";
    s += stringify(scan_header.total_ion_current);
    s += "\">";

    __o.write(s.c_str(), s.length());

    int precision = 8 * sizeof(pData[0]);

    // start of <peaks tag
    s = "\n␣␣<peaks␣precision=\"";
    s += stringify(precision);
    s += "\"\n␣␣byteOrder=\"network\"";
    s +=   "\n␣␣pairOrder=\"m/z-int\">";
    __o.write(s.c_str(), s.length());

    // get a base-64 encoded version of the data..
    s = pack(pData);
    __o.write(s.c_str(), s.length());

    s = "</peaks>\n␣␣</scan>";
    __o.write(s.c_str(), s.length());

    return !__o.fail();
}
```

```
/**
 * @brief Writes a new scan to the open file
 *
 * @param scan_header   A scan header structure with
 *        information on the current scan
 * @param pData         An array of floating point values
 * @param size          The number of elements in
 *        pData to be written
 * @return true if the data was written successfully,
 *         false if there was a problem
 */
bool Writer::write_scan(ScanHeaderStruct &scan_header, float
   *pData, size_t &size)
{
  if(!__o.is_open()) return false; // the file isn't open yet.

  // we don't want to write any more scans after
  // the index has been written
  if(__index_written) return false;

  __o.seekp(0, ios::end);
  size_t __c_offset = __o.tellp();
  __index.push_back(__c_offset);

  // write the scan properties...
  float _bp_intensity = -1.0;
  float _bp_mz = 0.0;

  // locate the base peak:
  size_t index = 0;
  for(index = 0; index < size; index += 2) {
    if(pData[index + 1] > _bp_intensity) {
      _bp_intensity = pData[index + 1];
      _bp_mz = pData[index];
    }
  }

  // index now stores the number of mz/intensity pairs
  index /= 2;

  string s;

  s = "\n␣␣<scan␣num=\"";
  s += stringify(++__current_scan);
  s += "\"\n␣␣msLevel=\"";
```

```
    s += stringify(scan_header.ms_level);
    s += "\"\n  peaksCount=\"";
    s += stringify(index);
    s += "\"\n  polarity=\"";
    s += scan_header.polarity;
    s += "\"\n  scanType=\"";
    s += scan_header.scan_type;
    s += "\"\n  retentionTime=\"PT";
    s += stringify(scan_header.retention_time);
    s += "\"\n  lowMz=\"";
    s += stringify(scan_header.low_mz);
    s += "\"\n  highMz=\"";
    s += stringify(scan_header.high_mz);
    s += "\"\n  basePeakMz=\"";
    s += stringify(_bp_mz);
    s += "\"\n  basePeakIntensity=\"";
    s += stringify(_bp_intensity);
    s += "\"\n  totIonCurrent=\"";
    s += stringify(scan_header.total_ion_current);
    s += "\">";

    __o.write(s.c_str(), s.length());

    int precision = 8 * sizeof(pData[0]);

    // start of <peaks tag
    s = "\n  <peaks precision=\"";
    s += stringify(precision);
    s += "\"\n  byteOrder=\"network\"";
    s +=   "\n  pairOrder=\"m/z-int\">";
    __o.write(s.c_str(), s.length());

    // get a base-64 encoded version of the data..
    s = pack(pData, size);
    __o.write(s.c_str(), s.length());

    s = "</peaks>\n  </scan>";
    __o.write(s.c_str(), s.length());

    return !__o.fail();
}




/**
```

```
 * @brief Writes the peak index to the end of the file
 *
 * @return true if the index was written successfully,
 *         false if there was no index to write or if
 *         there was an error
 */
bool Writer::write_index()
{
  // make sure that there is an index to write
  if(__index.size() < 1) return false;

  // and that we haven't already written the index.
  if(__index_written) return false;

  __o.seekp(0, ios::end);
  string s = "\n</msRun>";
  __o.write(s.c_str(), s.length());

  __index_offset = __o.tellp();

  s = "\n<index␣name=\"scan\">";
  __o.write(s.c_str(), s.length());

  for(size_t i = 1; i <= __index.size(); ++i) {
    s = "\n␣␣<offset␣id=\"";

    s += stringify(i);
    s += "\">";

    // the indexing starts at zero, but scan numbers start at 1.
    s += stringify(__index[i-1]);
    s += "</offset>";
    __o.write(s.c_str(), s.length());
  }
  s = "\n</index>";
  __o.write(s.c_str(), s.length());

  // write the index offset:
  if(__index_offset < 0) {
    s = "\n<indexOffset␣xsi:nil=\"1\"/>";
  }
  else {
    s = "\n<indexOffset>";
    s += stringify(__index_offset);
    s += "</indexOffset>";
  }
  s += "\n</mzXML>";
  __o.write(s.c_str(), s.length());

  __index_written = true;

  return write_checksum();
}




/**
 * @brief Computes the SHA1 checksum for the
 *        file and writes the closing tag.
 *
 * @return true if the checksum was written without
 *         error, false if the index has been previously
 *         written or if there was an error writing to the file.
 */
bool Writer::write_checksum()
{
  // we don't want to write the checksum before the index
  if(!__index_written) return false;

  // the file isn't open.
  if(!__o.is_open()) return false;

  // the hash is computed from the first
  // character until the end of the <sha1>
  // tag, so we must first right that tag.
  __o.seekp(0, ios::end);
  __o.write("\n<sha1>", 7);
  __o.close();

  // re-open the file to read the data..
  ifstream __in;
  __in.open(__f_name.c_str(), ios::in);
  if(!__in.is_open()) {
    __in.close();
    return false;
  }

  __in.seekg(0, ios::end);
  size_t eof = __in.tellg();
  size_t block_size = 64;
  SHA1 sha;
  sha.init_state();

  char memblock[block_size];
  __in.seekg(0, ios::beg);
  // this is a strange hack so that we can compare
  // a streampos type (the result of tellg()) to the
  // eof.
  while((size_t)__in.tellg() < eof - block_size) {
    __in.read(memblock, block_size);
    sha.add_block(memblock, block_size);
  }

  // add the last of the data..
  block_size = eof - __in.tellg();
  __in.read(memblock, block_size);
  sha.add_block(memblock, block_size);
  __in.close();

  string hash = sha.get_hash();

  // reopen the output and write this data..
  __o.open(__f_name.c_str(), ios::app);
  __o.seekp(0, ios::end);
  hash += "</sha1>";
  __o.write(hash.c_str(), hash.length());

  return !__o.fail();
}
```

mzxml_types.h

```
#ifndef MZXML_TYPES_H_INCLUDED
#define MZXML_TYPES_H_INCLUDED

#include <map>
#include <vector>

namespace mzxml
{

  struct ScanHeaderStruct
```

```
{
  // number in sequence observed file (1-based)
  int seqNum;
  // scan number as declared in File (may be gaps)
  int num;
  int  ms_level;
  unsigned int  peaks_count;
  double total_ion_current;
  double retention_time;        /* in seconds */
  double base_peak_mz;
  double base_peak_intensity;
  double collision_energy;
  double ionization_energy;
  double low_mz;
  double high_mz;
  unsigned int precursor_scan_num;
  double precursor_mz;
  unsigned int precursor_charge;
  double precursor_intensity;
  std::string scan_type;
  /* currently unused, but technically part of the schema.
  int mergedScan;
  int mergedResultScanNum;
  int mergedResultStartScanNum;
  int mergedResultEndScanNum;
  */
  std::string polarity;
  size_t file_position;

  ScanHeaderStruct() : seqNum(0), num(0), ms_level(0),
    peaks_count(0),       total_ion_current(0.0),
    retention_time(0.0), base_peak_mz(0.0),
    base_peak_intensity(0.0), collision_energy(0.0),
    ionization_energy(0.0), low_mz(0.0), high_mz(0.0),
    precursor_scan_num(0), precursor_mz(0.0),
    precursor_charge(0), precursor_intensity(0.0),
    file_position(0) { }
};

struct RunHeaderStruct
{
  unsigned int scan_count;
  double lowMZ;
  double highMZ;
  double startMZ;
  double endMZ;
```

```
    double start_time;      //!< Start time in seconds
    double end_time;        //!< End time in seconds
  };

  typedef struct
  {
    std::string type;
    std::string name;
    std::string version;
  } SoftwareStruct;

  typedef struct
  {
    std::string file_name;
    std::string file_type;
    std::string file_SHA1;
  } ParentFileStruct;

  typedef struct
  {
    std::map<std::string, std::string> data;
    // contains information about previous processing..
    std::vector<SoftwareStruct>  software;
  } InstrumentStruct;

  typedef unsigned int word32;
}
#endif                          // MZXML_TYPES_H_INCLUDED
```

mzxml_sha1.h

```
#ifndef MZXML_SHA1_H_INCLUDED
#define MZXML_SHA1_H_INCLUDED

#include <string>

#include ”mzxml_types.h”
#include ”mzxml_lib.h”

namespace mzxml
{

  /**
   * @brief SHA1 is a pared-down implementation of the SHA1
   *         hash adapted from the cryptopp library
   */
  class SHA1
  {

    public:
      SHA1();
      ~SHA1();
      void        init_state();
      void        transform(const word32 * data);
      void        add_block(char * cData, int cLen);

      // returns a hex string of the hash.
      std::string get_hash();

    private:
      word32       state[5];
      word32       rotlFixed(word32 x, int y);
      word32       charsToW32(char * data,int start,  int stop);
      long long int     bitLen;
  };

}
#endif                                    // MZXML_SHA1_H_INCLUDED
```

mzxml_sha1.cpp

```
#include "mzxml_sha1.h"

using namespace mzxml;

#define blk0(i) (W[i] = data[i])
#define blk1(i) (W[i&15] =
   rotlFixed(W[(i+13)&15]^W[(i+8)&15]^W[(i+2)&15]^W[i&15],1))

#define f1(x,y,z)  (z^(x&(y^z)))
#define f2(x,y,z)  (x^y^z)
#define f3(x,y,z)  ((x&y)|(z&(x|y)))
#define f4(x,y,z)  (x^y^z)
```

```
/* (R0+R1), R2, R3, R4 are the
 * different operations used in SHA1 */
#define R0(v,w,x,y,z,i) z+=f1(w,x,y)+blk0(i) \
    +0x5A827999+rotlFixed(v,5);w=rotlFixed(w,30);
#define R1(v,w,x,y,z,i) z+=f1(w,x,y)+blk1(i) \
    +0x5A827999+rotlFixed(v,5);w=rotlFixed(w,30);
#define R2(v,w,x,y,z,i) z+=f2(w,x,y)+blk1(i) \
    +0x6ED9EBA1+rotlFixed(v,5);w=rotlFixed(w,30);
#define R3(v,w,x,y,z,i) z+=f3(w,x,y)+blk1(i) \
    +0x8F1BBCDC+rotlFixed(v,5);w=rotlFixed(w,30);
#define R4(v,w,x,y,z,i) z+=f4(w,x,y)+blk1(i) \
    +0xCA62C1D6+rotlFixed(v,5);w=rotlFixed(w,30);

SHA1::SHA1()
{
   init_state();
}

SHA1::~SHA1(){}

word32 SHA1::rotlFixed(word32 x, int y)
{
   return (x << y) | (x >> (32 - y));
}




/**
 * @brief Transform a block of characters into a 32-bit word
 *
 * @param data  A character array
 * @param start The index of the first character we care about
 * @param stop  The index of the last character we care about,
 *              up to start + 4
 * @return A 32-bit integer representation of the data
 */
word32 SHA1::charsToW32(char * data, int start,  int stop)
{
   word32 ret = 0x00000000L;
   stop = (stop > (start + 4)) ? (start + 4) : stop;
   stop = (stop < start) ? start : stop;

   int ind = 0;
   for(int i = start; i < stop; i++) {
      ret |= ((int)data[i] & 0xFF) << (8 * (3-ind++));
```

```
  }
  return ret;
}






/**
 * @brief Adds a block of characters to the hash
 *
 * This method makes the assumption that if the length
 * of the supplied data block is less than 64 characters,
 * then this must be the end of the hash. While in
 * general this is a bad assumption, since this hash
 * is only used to validate entire files that can be
 * read in blocks, we should be ok.
 */
void SHA1::add_block(char * cDat, int cLen)
{
  // cut the data into 32bit blocks (4 chars);
  cLen = (cLen < 0) ? 0 : cLen;
  word32 bDat[16];
  if(cLen >= 64) {
    for(int i = 0; i < 16; i++) {
      bDat[i] = charsToW32(cDat, (i * 4), cLen);
    }
    bitLen += 512;
    transform(bDat);
  }
  else {
    // we need to pad, etc.
    bitLen += (8 * cLen);
    char * finalDat = NULL;
    if(cLen > 55) {
      finalDat = (char *) realloc(finalDat, 128);
    }
    else {
      finalDat = (char *) realloc(finalDat, 64);
    }
    memcpy(finalDat, cDat, cLen);
    finalDat[cLen++] = (char)0x80;
    while((cLen % 64) != 56) {
      finalDat[cLen++] = (char) 0;
    }
```

```
      // add the bit length of the stream, in big-endian order:
      short sTest = 0x0001;
      char * byte = (char *) &sTest;
      if(byte[0]) {
        for(int i = 0; i < 8; i++) {
          finalDat[cLen++] = (char)((bitLen >> (8 * (7-i))) &
             0xff);
        }
      }
      else {
        for(int i = 7; i >= 0; i--) {
          finalDat[cLen++] = (char)((bitLen >> (8 * (7-i))) &
             0xff);
        }
      }
      for(int i = 0; i < 16; i++) {
        bDat[i] = charsToW32(finalDat, (i * 4), cLen);
      }
      transform(bDat);
      if(cLen > 64) {
        for(int i = 0; i < 16; i++) {
          bDat[i] = charsToW32(finalDat, (i + 16) * 4, cLen);
        }
        transform(bDat);
      }
    }
  }



  /**
   * @brief Updates the current hash state with a chunk of data
   */
  void SHA1::transform(const word32 * data)
  {
    word32 W[16];
    /* Copy context->state[] to working vars */
    word32 a = state[0];
    word32 b = state[1];
    word32 c = state[2];
    word32 d = state[3];
    word32 e = state[4];
    /* 4 rounds of 20 operations each. Loop unrolled. */
    R0(a,b,c,d,e, 0); R0(e,a,b,c,d, 1); R0(d,e,a,b,c, 2);
    R0(c,d,e,a,b, 3); R0(b,c,d,e,a, 4); R0(a,b,c,d,e, 5);
  R0(e,a,b,c,d, 6); R0(d,e,a,b,c, 7); R0(c,d,e,a,b, 8);
  R0(b,c,d,e,a, 9); R0(a,b,c,d,e,10); R0(e,a,b,c,d,11);
  R0(d,e,a,b,c,12); R0(c,d,e,a,b,13); R0(b,c,d,e,a,14);
  R0(a,b,c,d,e,15); R1(e,a,b,c,d,16); R1(d,e,a,b,c,17);
  R1(c,d,e,a,b,18); R1(b,c,d,e,a,19); R2(a,b,c,d,e,20);
  R2(e,a,b,c,d,21); R2(d,e,a,b,c,22); R2(c,d,e,a,b,23);
  R2(b,c,d,e,a,24); R2(a,b,c,d,e,25); R2(e,a,b,c,d,26);
  R2(d,e,a,b,c,27); R2(c,d,e,a,b,28); R2(b,c,d,e,a,29);
  R2(a,b,c,d,e,30); R2(e,a,b,c,d,31); R2(d,e,a,b,c,32);
  R2(c,d,e,a,b,33); R2(b,c,d,e,a,34); R2(a,b,c,d,e,35);
  R2(e,a,b,c,d,36); R2(d,e,a,b,c,37); R2(c,d,e,a,b,38);
  R2(b,c,d,e,a,39); R3(a,b,c,d,e,40); R3(e,a,b,c,d,41);
  R3(d,e,a,b,c,42); R3(c,d,e,a,b,43); R3(b,c,d,e,a,44);
  R3(a,b,c,d,e,45); R3(e,a,b,c,d,46); R3(d,e,a,b,c,47);
  R3(c,d,e,a,b,48); R3(b,c,d,e,a,49); R3(a,b,c,d,e,50);
  R3(e,a,b,c,d,51); R3(d,e,a,b,c,52); R3(c,d,e,a,b,53);
  R3(b,c,d,e,a,54); R3(a,b,c,d,e,55); R3(e,a,b,c,d,56);
  R3(d,e,a,b,c,57); R3(c,d,e,a,b,58); R3(b,c,d,e,a,59);
  R4(a,b,c,d,e,60); R4(e,a,b,c,d,61); R4(d,e,a,b,c,62);
  R4(c,d,e,a,b,63); R4(b,c,d,e,a,64); R4(a,b,c,d,e,65);
  R4(e,a,b,c,d,66); R4(d,e,a,b,c,67); R4(c,d,e,a,b,68);
  R4(b,c,d,e,a,69); R4(a,b,c,d,e,70); R4(e,a,b,c,d,71);
  R4(d,e,a,b,c,72); R4(c,d,e,a,b,73); R4(b,c,d,e,a,74);
  R4(a,b,c,d,e,75); R4(e,a,b,c,d,76); R4(d,e,a,b,c,77);
  R4(c,d,e,a,b,78); R4(b,c,d,e,a,79);
  /* Add the working vars back into context.state[] */
  state[0] += a;
  state[1] += b;
  state[2] += c;
  state[3] += d;
  state[4] += e;
}




/**
 * @brief Initializes the state of the hash
 */
void SHA1::init_state()
{
  bitLen = 0;
  state[0] = 0x67452301L;
  state[1] = 0xEFCDAB89L;
  state[2] = 0x98BADCFEL;
  state[3] = 0x10325476L;
  state[4] = 0xC3D2E1F0L;
}




/**
 * @brief Returns a hex string of the current hash state
 */
std::string SHA1::get_hash()
{
  std::string ret = "";

  ret.append(pad_left(to_hex(state[0]),8,'0'));
  ret.append(pad_left(to_hex(state[1]),8,'0'));
  ret.append(pad_left(to_hex(state[2]),8,'0'));
  ret.append(pad_left(to_hex(state[3]),8,'0'));
  ret.append(pad_left(to_hex(state[4]),8,'0'));
  return ret;
}
```

B.3 Graph theoretic utilities

B.3.1 Description

Graph theory is a useful construct both for simplifying complex datasets and determining subsets of data where certain relationships hold. In essence, a graph is a set of nodes, where edges can exist between any two nodes. While there are many different types of graphs, here we consider only undirected, unweighted graphs. For our purposes, we define three basic operations on graphs, shown in Table B-1. Of these, the key operation as it applies to the analysis of MS data is the ability to decompose the graph into disjoint subsets, where each elements in each subset are linked to every other element in the subset by some traversal through defined edges.

The MsGraph class is a template class that can be used to build a graph out of any data type, and supports all of the operations listed in Table B-1. This class has been implemented as a list, where each node points to a piece of data as well as an adjacency list. As most graphs used in this this project are sparse[‡], this is significantly

[‡] A given node only has edges with a small fraction of the other nodes.

Table B-1: Operations on a graph required for MS analysis

1. Add a new node to the graph
2. Create an edge between two nodes
3. Decompose the graph into disjoint subsets as defined by the edges in the graph

more efficient than an adjacency matrix. This class also defines an iterator over the main graph list to facility access to the individual nodes, and the build_edge function takes two of these iterators as arguments (and builds an edge between the two referenced nodes). Finally, the $get\_disjoint\_subsets()$ function returns a list of lists of the original data type, where elements in each sublist were related in the final graph[§].

The MsGraph class also supports many of the standard features of the $std :: list$ data type, include $begin()$, $end()$, $clear()$, $rbegin()$, $rend()$, $empty()$, $size()$, $max\_size()$, and $push\_back()$.

B.3.2 C++ code

ms_graph.h

```
#ifndef MS_GRAPH_H_INCLUDED
#define MS_GRAPH_H_INCLUDED

#include <list>

// debugging:
#include <iostream>

/**
 * @brief A generic node type for a graph that contains
 *        data (_Tp) and an adjacency list
 */
template <typename _Tp>
struct _MS_graph_node
{
  typedef  std::list<_MS_graph_node<_Tp>*>  _Node_list_type;

  //!< A list of adjacent nodes
  _Node_list_type _M_adjacent;
```

[§] Note: this function clears the original graph. This implementation was chosen as often times the graphs utilized are extremely large, and such an implementation allows the graph to be decomposed without copying data.

```

  //!< The actual data stored at this node
  _Tp  _M_data;

   //!< A flag indicated whether or not this
   //    node has been visited in a traversal
  bool _M_visited;


  _MS_graph_node(_Tp __x) : _M_data(__x), _M_visited(false){ }
};

/**
 * @brief A generic graph type that supports
 *        decomposition into disjoint sub-graphs.
 *
 * This is an implementation of a graph structure that uses an
 * adjacency list for storing edge information. At this point,
 * the edges are unlabeled only only contain pointers to the
 * nodes that are adjacent to a given node. The main data in the
 * graph is stored as a standard std::list, and access to this
 * list is provided through iterators, with most of the standard
 * list features supported.
 */
template <typename _Tp>
class MsGraph
{
  private:
    typedef _MS_graph_node<_Tp>   _Node_type;
    typedef _Tp                   _Data_type;
    typedef std::list<_Node_type> _List_type;
    typedef typename _MS_graph_node<_Tp>::_Node_list_type
        _Node_list_type;

    //!< The main list that contains the data
    _List_type  _M_list;

    void get_sublist(_Node_type *n, std::list<_Tp> &_r);
    void reset_labels();

  public:

    //!< An iterator type for the main list
    typedef struct _List_type::iterator iterator;

    //!< A reverse iterator type for the main list
  typedef struct _List_type::reverse_iterator
      reverse_iterator;

  // MsGraph inherits a large majority of its
  // methods from it's base list. We want the
  // main list itself to behave exactly as list

  //!< Returns an iterator to the beginning of the graph data
  iterator begin() {
    return _M_list.begin();
  }

  //!< Returns an iterator to the end of the graph data
  iterator end() {
    return _M_list.end();
  }

  //!< Returns a reverse iterator to the beginning
  //   of the graph data
  iterator rbegin() {
    return _M_list.rbegin();
  }

  //!< Returns a reverse iterator to the end of the graph data
  iterator rend() {
    return _M_list.rend();
  }

  //!< Returns true of the graph is currently empty
  bool empty() {
    return _M_list.empty();
  }

  //!< Returns a size_type with the number of
  //   nodes in the graph
  size_t size() {
    return _M_list.size();
  }

  //!< Returns a size_type indicating the maximum
  //   number of nodes the graph can hold
  size_t max_size() const

  {
    return _M_list.max_size();
  }
```

```

    // Add a few more of the content handling functions
    // that are basic to lists.

    //!< Creates a new data node at the end of the graph
    void push_back(_Tp __x)
      { _M_list.push_back(_MS_graph_node<_Tp>(__x)); }

    //!< Creates a new data node at the beginning of the graph
    void push_front(_Tp __x)
      { _M_list.push_front(_MS_graph_node<_Tp>(__x)); }

    //!< Removes the first node from the graph
    void pop_front()
      { _M_list.pop_front(); }

    //!< Removes the last node in the graph
    void pop_back()
      { _M_list.pop_back(); }

    //!< Adds a new node at the position indicated by position
    iterator insert(iterator position, const _Tp& __x)
      {return  _M_list.insert(position, __x); }

    //!< Removes the node at the supplied iterator position
    //   and returns an iterator to the proceeding node
    iterator erase(iterator position)
      { return _M_list.erase(position); }

    //!< Removes the nodes between start and end (including
    //   start but not end) and returns an iterator to end
    iterator erase(iterator start, iterator end)
      { return _M_list.erase(start, end); }

    //!< Deletes all data in the graph
    void clear()
      { _M_list.clear(); }

    //!< Reverses the order of elements in the graph
    void reverse()
      { _M_list.reverse(); }

    void add_edge(iterator node1, iterator node2);

    void print();

    std::list< std::list<_Tp> >
      get_disjoint_lists();

};

/**
 * @brief Creates an edge between the two supplied nodes
 *
 * @param node1   The first node
 * @param node2   The second node
 */
template <typename _Tp>
void MsGraph<_Tp>::add_edge(iterator node1, iterator node2)
{
  node1->_M_adjacent.push_back(&(*node2));
  node2->_M_adjacent.push_back(&(*node1));
}



// for testing and debugging purposes.
template <typename _Tp>
void MsGraph<_Tp>::print()
{
  iterator __iter = begin();
  iterator __end = end();
  std::cout << "======graph::print()=========\n";

  for(; __iter != __end; ++__iter) {
    std::cout << "(" << (__iter->_M_visited ? "true" : "false")
              << ")␣␣" << __iter->_M_data;
    typename std::list<_MS_graph_node<_Tp>* >::iterator
              _sublist_begin = (__iter)->_M_adjacent.begin();
    typename std::list<_MS_graph_node<_Tp>* >::iterator
              _sublist_end =   (__iter)->_M_adjacent.end();
    for( ; _sublist_begin != _sublist_end; ++_sublist_begin) {
      std::cout << "--" << (*_sublist_begin)->_M_data << "␣("
                << ((*_sublist_begin)->_M_visited ? "true" :
                   "false")
                << ")";
    }
    std::cout << "\n";

  }
  std::cout << "======END=graph::print()====\n";
```

```
}



/**
 * @brief A public method that returns a list of
 *        disjoint lists based on the graph This
 *        function calls get_sublist for each node
 *        in the list, which then traverses  the graph
 *        to obtain a disjoint sublist, marking all
 *        visited nodes as `visited'.
 *
 * @return A sorted list of disjoint sublists from the graph.
 */
template <typename _Tp>
std::list<std::list<_Tp> > MsGraph<_Tp>::get_disjoint_lists()
{
  std::list<std::list<_Tp> > __r;
  iterator __iter = begin();
  iterator __end = end();

  for(; __iter != __end; ++__iter) {
    if(__iter->_M_visited) continue;
    std::list<_Tp> tmp;
    get_sublist(&(*__iter), tmp);
    tmp.sort();
    __r.push_back(tmp);
  }

  return __r;
}



/**
 * @brief A private method that traverses the graph
 *        starting at the supplied node get_sublist recursively
 *        builds up the disjoint sub-graph containing the
 *        supplied node, adding each element to the provided
 *        list reference _r. Each visited node is then marked
 *        as visited so that each supplied subgraph is unique.
 *
 * @param n      The starting node
 * @param _r     The return sub-graph containing the node n
 */
template <typename _Tp>
void MsGraph<_Tp>::get_sublist(_Node_type *n, std::list<_Tp>
   &_r)
{
  // check to see if this has been visited
  if(n->_M_visited) return;

  // if not, now it has.
  n->_M_visited = true;
  _r.push_back(n->_M_data);
  typename _MS_graph_node<_Tp>::_Node_list_type::iterator
     __iter, __end;
  __end = n->_M_adjacent.end();
  /* this is a bit ugly. __iter is a pointer to the
   * data stored in the node, which is in turn a
   * pointer to a node. To access the node, we need
   * **__iter, and we then need to pass this as
   * a pointer, so the overall expression is &(**__iter).
   */
  for(__iter = n->_M_adjacent.begin(); __iter != __end;
     ++__iter)
    get_sublist(&(**__iter), _r);
}




/**
 * @brief Resets the visited flag on each node in the graph
 */
template <typename _Tp>
void MsGraph<_Tp>::reset_labels()
{
  typename _MS_graph_node<_Tp>::_Node_list_type::iterator
     __iter, __end;
  __end = end();
  for(__iter = begin(); __iter != __end; ++__iter)
    (*__iter)._M_visited = false;
}
#endif                                          // MS_GRAPH_H_INCLUDED
```

ms_scan_graph.h

```
#ifndef SCAN_GRAPH_H_INCLUDED
#define SCAN_GRAPH_H_INCLUDED

#include "ms_graph.h"
#include "ms_feature.h"

#ifndef ABS
#define ABS(x) ((x) < 0 ? -(x) : (x))
#endif

/**
 * @class ScanGraph This is a simple graph datatype that allows
 * the user to add a scan in the form of mz/intensity pairs and
 * subsequently define linkages between peaks.  Finally, the
 * graph can be decomposed into disjoint subgraphs.  As it
 * stands this class defines linkages between nodes based solely
 * on m/z values, and thus decomposed a set of peaks into
 * subsets that are possibly isotopically related.  However,
 * redefinition of the function need_edge(...) allows for a much
 * broader range of applications for this class, for instance in
 * de novo sequencing applications.
 */
template <typename _Tp>
class ScanGraph
{
  public:
    ScanGraph(unsigned short c_start, unsigned short c_end,
       float delta_m) :
    _c_start(c_start), _c_end(c_end), _delta_m(delta_m) { }

        //!< Clear the graph
    void clear()
      { _M_graph.clear(); }

    void add_scan(_Tp *data, size_t data_size);
    void build_edges();
    std::list<MsDetectedFeature<_Tp> >
       get_convoluted_features();

  private:
    //!< An iterator to the graph
    typedef typename MsGraph<pair<_Tp,_Tp> >::iterator iterator;
    MsGraph<pair<_Tp,_Tp> > _M_graph;    //!< The graph
```

```
        //!< The lowest charge state to be considered
    unsigned short _c_start;
    //!< The highest charge state to be considered
    unsigned short _c_end;
    //!< The m/z tolerance when building the graph
    float _delta_m;

    bool need_edge(iterator &_l, iterator &_r);
    inline bool mz_values_align(_Tp __d);
    MsDetectedFeature<_Tp>
       list_to_feature(std::list<std::pair<_Tp,_Tp> > &_list);

};

/**
 * @brief Adds the points in a current scan to the graph
 *
 * @param data        A list of points in mz / intensity pairs
 * @param data_size   The number of points in data
 */
template <typename _Tp>
void ScanGraph<_Tp>::add_scan(_Tp *data, size_t data_size)
{
  for(size_t i = 0; i < data_size; i += 2) {
    _M_graph.push_back(pair<_Tp,_Tp>(data[i],data[i+1]));
  }
}



/**
 * @brief Builds edges in the current graph based on whether or
    not
 *        two peaks could be isotopically related
 */
template <typename _Tp>
void ScanGraph<_Tp>::build_edges()
{
  iterator __start, __tmp, __end = _M_graph.end();
  for(__start = _M_graph.begin(); __start != __end; ++__start) {
    __tmp = __start;
    ++__tmp;
    for(; __tmp != __end; ++__tmp) {
      if(need_edge(__start, __tmp)) {
        _M_graph.add_edge(__start, __tmp);
      }

      // make this function ~O(n);
      if( __tmp->_M_data.first > __start->_M_data.first + 2.0 +
         _delta_m) break;
    }

  }

}




/**
 * @brief Returns true if we need an edge between two nodes,
 *        false otherwise
 *
 * @param _l  An iterator to one graph node
 * @param _r  An iterator to a second graph node
 * @return true if an edge is required, false otherwise
 */
template <typename _Tp>
bool ScanGraph<_Tp>::need_edge(iterator &_l, iterator &_r)
{
  return mz_values_align(ABS(_l->_M_data.first -
     _r->_M_data.first));
}




/**
 * @brief Returns true if two peaks could be isotopically
 *        related, false otherwise
 *
 * @param _d  The m/z difference (absolute value) between two
 *        peaks
 * @return true if the distance could be between two isotopic
 *        peaks, false otherwise
 */
template <typename _Tp>
inline bool ScanGraph<_Tp>::mz_values_align(_Tp __d)
{
  for(unsigned short c = _c_start; c <= _c_end; ++c) {
    if( abs(__d - (1.0 / c)) < _delta_m) return true;
    //  one missed peak.
    if( abs(__d - (2.0 / c)) < _delta_m) return true;
  }

  return false;
}




/**
 * @brief Decompose the graph that we've build into disjoint
 *        features
 *
 * @return A list of detected features of disjoint sets in
 *         the graph
 */
template <typename _Tp>
std::list<MsDetectedFeature<_Tp> >
   ScanGraph<_Tp>::get_convoluted_features()
{
  std::list< std::list< std::pair<_Tp,_Tp> > > features =
     _M_graph.get_disjoint_lists();
  _M_graph.clear();

  std::list<MsDetectedFeature<_Tp> > __r;

  // we want to convert each one of these
  // disjoint lists into a detected feature
  typename std::list< std::list< std::pair<_Tp,_Tp> >
     >::iterator __start, __end = features.end();
  for( __start = features.begin(); __start != __end; ++__start) {
    __r.push_back(list_to_feature(*__start) );
  }

  return __r;
}




/**
 * @brief Converts a list of m/z , intensity pairs into an
 *        MsDetectedFeature
 *
 * @param _list   A list of m/z, intensity pairs
 * @return An MsDetectedFeature containing all of the supplied
 *         points
 */
template <typename _Tp>
MsDetectedFeature<_Tp>
   ScanGraph<_Tp>::list_to_feature(std::list<std::pair<_Tp,_Tp>
   > &_list)
{
  MsDetectedFeature<_Tp> __r;

  typename std::list<std::pair<_Tp,_Tp> >::iterator __start,
     __end = _list.end();
  for(__start = _list.begin(); __start != __end; ++__start) {
    __r.push_back(MsPeak<_Tp>(__start->first, __start->second)
       );

  }

  return __r;
}
#endif       // SCAN_GRAPH_H_INCLUDED
```

ms_feature_graph.h

```
#ifndef FEATURE_GRAPH_H_INCLUDED
#define FEATURE_GRAPH_H_INCLUDED

#include <list>
#include <assert.h>

#include "ms_graph.h"
#include "matched_pattern.h"

#ifndef ABS
#define ABS(x) ((x) < 0 ? -(x) : (x))
#endif

/**
 * This class is a wrapper for a generic graph class which
 * builds up a list of matched patterns. Edges can then be built
 * based on a variety of criteria, and finally the decomposition
 * of the graph can be obtained
 */
template <typename _Tp>
class FeatureGraph
{
  public:
    typedef typename MsGraph<MatchedPattern<_Tp> >::iterator
        iterator;

    iterator begin() {
      return _M_graph.begin();
    }

    iterator end() {
      return _M_graph.end();
    }

    void push_back(MatchedPattern<_Tp> _feat) {
      _M_graph.push_back(_feat);
    }

    size_t size() {
      return _M_graph.size();
    }

    void build_edges(unsigned int max_scan_gap, float delta_m);
    std::list<FinalFeature<_Tp,10> > decompose();

  private:
    MsGraph<MatchedPattern<_Tp> > _M_graph;
    bool need_edge(iterator &_l, iterator &_r, unsigned int
        gap, float delta_m);

};

/**
 * @brief Build edges between the nodes based on whether or not
 *        two nodes are temporally close and could be from the
 *        same molecular species
 *
 * @param max_scan_gap The maximum number of scans between
 *        two nodes before they are not considered related
 * @param delta_m  The m/z tolerance for matching two peaks
 *        - note, this should probably be larger than /
 *        required for pattern matching as m/z values are scaled
 *        by charge before comparison
 */
```

```
template <typename _Tp>
void FeatureGraph<_Tp>::build_edges(unsigned int max_scan_gap,
    float delta_m)
{
  // build up edges here.... we make the assumption that
  // features have been added in order by scan numbers
  iterator __iter, __tmp, __end = _M_graph.end();

  for(__iter = _M_graph.begin(); __iter != __end; ++__iter) {
    __tmp = __iter;
    ++__tmp;
    while(__tmp != __end) {
      if(__tmp->_M_data.scan_num > __iter->_M_data.scan_num +
          max_scan_gap)
        break;

      // see if we need an edge..
      if(need_edge(__iter, __tmp, max_scan_gap,  delta_m)) {
        _M_graph.add_edge(__iter, __tmp);
      }
      ++__tmp;
    }
  }

}




/**
 * @brief Decide if we need an edge between two nodes
 *
 * @param _l An iterator pointing to the first node
 * @param _r An iterator pointing to the second node
 * @param gap The maximum gap (in scan numbers) between
 *        two nodes
 * @param delta_m The m/z tolerance for considering two
 *        peaks to be the same or of the same molecular
 *        species
 * @return true if we should put an edge between these
 *         nodes, false otherwise
 */
template <typename _Tp>
bool FeatureGraph<_Tp>::need_edge(iterator &_l, iterator &_r,
    unsigned int gap, float delta_m)
{
  assert(_r->_M_data.scan_num <= _l->_M_data.scan_num + gap);
  if( ABS(_l->_M_data.parent_mass - _r->_M_data.parent_mass) <
     delta_m) return true;

  return false;
}




template <typename _Tp>
std::list<FinalFeature<_Tp,10> > FeatureGraph<_Tp>::decompose()
{
  std::list<FinalFeature<_Tp,10> > __r;

  std::list<std::list< MatchedPattern<_Tp> > > matched =
     _M_graph.get_disjoint_lists();
  typename std::list<std::list<MatchedPattern<_Tp> >
     >::iterator __iter, __end = matched.end();

  for(__iter = matched.begin(); __iter != __end; ++__iter) {
    // make a feature out of this...
    __r.push_back(FinalFeature<_Tp,10>(*__iter));
  }

  return __r;
}
#endif    // FEATURE_GRAPH_H_INCLUDED
```

B.4 Pattern searching

B.4.1 Description

As described in Chapter 2, the main pattern searching algorithm works in two steps. First, each full scan from an LC-MS file is searched for the desired isotopic pattern. This is done through the use of a graph theoretic utility, scan_graph, which separates each full scan into sets of related peaks. These sets are then analyzed by the class pattern_lib_mod, which searches for the desired pattern and compares scores to other potential patterns. Putative matches are then stored as nodes in a graph through the class feature_graph, and the next scan in the file is processed. After all scans have been analyzed, edges are created on the feature_graph based on previously described criteria. Once all edges have been drawn, the graph is decomposed into disjoint sets which are stored in matched_pattern types. Information about these sets is then stored—for example, as a detailed summary and an m/z inclusion list.

B.4.2 C++ code

ms_pattern_search.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <boost/regex.hpp>

#include ”mzxml_lib/mzxml_reader.h”

#include ”params_helper.h”
#include ”pattern_lib_mod.h”
#include ”ms_tag.h”
#include ”matched_pattern.h”

#include ”scan_graph.h”
#include ”feature_graph.h”

using namespace std;
using namespace mzxml;


/**
 * This is the main program for pattern searching.  It takes as
 * input an mzXML data file, a parameters file, and the
 * destination of an output file.  The program then searches for
 * the pattern defined in the parameters file, accounting for
 * possible alternative patterns, and generates a list of
 * putative matches.  After generation of this list, a graph
 * theoretic approach is used to reduce the number of false
 * positive matches based on assumptions about LC–MS data: that
 * real peptide ions will likely be detected in more than one
 * charge state, and that the same ions will likely be
 * detectable in several adjacent scans.
 */
int main(int argc, char** argv)
{
  string infile = ””, outfile = ””, pfile = ””;
```

```
boost::cmatch what;
Isotope<float> is;

for(int i = 0; i < argc; ++i) {
  if(boost::regex_search(argv[i],what,boost::regex("^-i")))
    infile = argv[i+1];

  if(boost::regex_search(argv[i],what,boost::regex("^-o")))
    outfile = argv[i+1];
  if(boost::regex_search(argv[i], what, boost::regex("^-p")))
    pfile = argv[i+1];
}

if( (infile.length() < 1) || (outfile.length() < 1) ||
   pfile.length() < 1) {
  cout << "Invadlid arguments. Please enter files in the
     format \"-i <infile> -o <outfile> -p <params_file>\"" <<
     endl;
  return 0;
}
gsh::search_params params =
   gsh::read_parameters(pfile.c_str());
cout << "parameters read!" << endl;



Reader r;
r.open(infile);
ofstream LOGFILE;
LOGFILE.open(params.log_file.c_str(), ios::out);
LOGFILE << "Processing the file \"" << infile << "\"" << endl;
LOGFILE << "Storing results in \"" << outfile << "\"" << endl;
LOGFILE << "The searching will be logged in \"" <<
   params.log_file << "\"" << endl;
LOGFILE << (r.read_index_offset() ? "index offset read" :
   "failed reading index offset") << endl;
LOGFILE << "\tIndexOffset = " << r.get_index_offset() << endl;
LOGFILE << (r.read_index() ? "index read" : "failed reading
   index") << endl;
LOGFILE << "\tIndexSize    = " << r.get_scan_count() << endl;
LOGFILE << (r.read_run_header() ? "run header read." :
   "failed reading run header") << endl;
LOGFILE << (r.read_instrument_header() ? "instrument header
   read." : "failed reading instrument header") << endl;
LOGFILE << (r.read_parent_file_header() ? "parent file info
   read" : "failed reading parent file info") << endl;
```

```
LOGFILE << "there␣are␣" << r.get_scan_count() << "␣scans␣in␣
    this␣file." << endl;


// create some generic objects that we're going to need.
ScanGraph<float> scan_graph(params.charge_min,
    params.charge_max, params.mz_tolerance_2);
PatternLibMod<float, double> pl(params.sigma,
    params.per_sigma);
FeatureGraph<float> feature_graph;
list<MatchedPattern<float> > matched;



// set up patterns that we're going to match against:
vector<float> c_pattern;
vector<MsTag<float> > patterns;
typedef PatternLib<float,double>::pattern_type  _T_pattern;



// Add the search pattern first, then all of the alternate
// patterns afterwards.
patterns.push_back(params.search_pattern);
list<_T_pattern>::iterator __pattiter, __pattend =
    params.alt_patterns.end();
for(__pattiter = params.alt_patterns.begin(); __pattiter !=
    __pattend; ++__pattiter)
  patterns.push_back(*__pattiter);

// Process each scan in the file.
int index = 1;
while(r.scan_exists(index)) {
  cout << "reading␣index␣" << index << endl;
  size_t data_size;
  float * data = r.read_scan(index, data_size);


  scan_graph.add_scan(data, data_size);
  scan_graph.build_edges();
  list<MsDetectedFeature<float> > found =
      scan_graph.get_convoluted_features();
  list<MsDetectedFeature<float> >::iterator __start, __end =
      found.end();
      for(__start = found.begin(); __start != __end; ++__start) {
        list<MatchedPattern<float> > matched =
            pl.score_convoluted_feature(*__start,patterns, 1, 5,
            index);

        list<MatchedPattern<float> >::iterator _miter, _mend =
            matched.end();
        for(_miter = matched.begin(); _miter != _mend; ++_miter) {
          /*
           * Score filtering IS performed within the function
           * score_convoluted_feature, so long as the
           * PatternLibMod class is used as opposed to the
           * PatternLib class.
           */
          feature_graph.push_back(*_miter);
        }

      }

      /*
       * The scan_graph contains information about peaks in the
       * current scan. To process the next scan in the file, we
       * need to clear it, otherwise future data will be added
       * to the same graph.
       */
      scan_graph.clear();

      ++index;
    }

    cout << "Graph␣size:␣" << feature_graph.size() << "␣matched␣
       patterns." << endl;
    cout << "Building␣edges␣in␣feature␣graph..." << endl;

    // the build edges function is where we define how far apart
    // scans can be temporally before we will no longer consider
    // them to be linked.
    feature_graph.build_edges(params.scan_tolerance,
       params.mz_tolerance);
    cout << "...␣edges␣built." << endl;

    /*
     * This is again somewhat confusing, but the net scoring of
     * the final features is done in the constructor of the final
     * feature class. Unfortuantely, this does not make things
```

```
   * particularly transparent, and requires modification of
   * the class itself to change how features are scored. At
   * the moment, this code can be found in the FinalFeature
   * class, in the file "matched_pattern.h"
   */
  list<FinalFeature<float,10> > found =
     feature_graph.decompose();
  cout << "...␣graph␣decomposed." << endl;


  ofstream OUT;
  OUT.open(outfile.c_str(), ios::out);
  int max_size = 0;
  list<FinalFeature<float,10> >::iterator __iter, __end =
     found.end();
  int printed_count = 0;
  list<float> ions_list;
  list<float> parent_ions;
  cout << "...␣analyzing␣matches....";


  // Loop through all of the found matches, outputting
  // as necessary.
  for(__iter = found.begin(); __iter != __end; ++__iter) {

    if(__iter->score >= params.min_score) {
      ostringstream ions;
      OUT << "----------------\nFeature:" << endl;
      OUT << "\tParent_MW:␣" << __iter->parent_mass << "␣+/-␣"
          << __iter->stddev << endl;
      OUT << "\tEluting␣from␣scan␣" << __iter->scan_start << "␣
          to␣" << __iter->scan_end << endl;
      OUT << "\tCharge␣states:␣";


      // update the parent mass so that it points to
      // the desired peak of the pattern.
      __iter->parent_mass += params.include_mass_mod;
      for(int i = params.charge_min; i <= params.charge_max;
          ++i) {
        if(__iter->charge_states[i]) {
          OUT << "+" << i << "␣";
          ions << "\t" << ((__iter->parent_mass + i) / i) << "␣
              (+" << i << ")" << endl;
          ions_list.push_back((__iter->parent_mass + i) / i);
        }
      }
      parent_ions.push_back(__iter->parent_mass);
      OUT << "over " << __iter->count << " features." << endl;
      OUT << "Ions found:\n" << ions.str() << endl;
      OUT << "\tMinimum RMS error: " << __iter->min_rms_error
         << endl;
      OUT << "\tMaximum Probability: " <<
         __iter->max_probability << endl;
      OUT << "total score: " << __iter->score << endl;
      OUT << endl;
      ++printed_count;
    }

    if(__iter->count > max_size)
      max_size = __iter->count;
  }

  cout << "...done." << endl;
  ions_list.sort();
  parent_ions.sort();
  int duplicates = 0;
  cout << "...de-duplicating inclusion list...";



  // deduplicate the ions list here...
  list<float>::iterator ions_iter, ions_tmp, ions_end =
     ions_list.end();
  for(ions_iter = ions_list.begin(); ions_iter != ions_end;
     ++ions_iter) {
    ions_tmp = ions_iter;
    ++ions_tmp;
    if(ions_tmp == ions_end) break;
    if( (*ions_tmp) < (*ions_iter) + 0.01 ) {
      // the same m/z within tolerance
      ++duplicates;
      ions_list.erase(ions_tmp);
    }
  }


  OUT.close();
  cout << "done." << endl;
  cout << "... analyzing parent ions....";
```

```
    // Remove m/z values that are within a certain
    // tolerance of other ions, so that the
    // inclusion list generated isn't unnecessarily large.
    ions_end = parent_ions.end();
    for(ions_iter = parent_ions.begin(); ions_iter != ions_end;
        ++ions_iter) {
      ions_tmp = ions_iter;
      ++ions_tmp;
      if(ions_tmp == ions_end) break;
      if( (*ions_tmp) < (*ions_iter) + 0.01) {
        parent_ions.erase(ions_tmp);
      }
    }
    cout << "done." << endl;



    ofstream fIL;
    outfile += ".include";
    fIL.open(outfile.c_str(), ios::out);
    ions_end = ions_list.end();
    cout << "...sending␣ions␣list␣to␣output␣in␣" << outfile <<
        "...";
    cout << "ions_list.size():␣" << ions_list.size() << "␣";

    // print all ions to the inclusion list.
    for(ions_iter = ions_list.begin(); ions_iter != ions_end;
        ++ions_iter) {
      fIL << (*ions_iter) << endl;
    }

    fIL.close();

    cout << "done." << endl;
    return 1;
}

// end pattern_search.cpp
```

params_helper.h

```
#ifndef PARAMS_HELPER_H_INCLUDED
```

```
#define PARAMS_HELPER_H_INCLUDED

#include <boost/regex.hpp>
#include <boost/algorithm/string.hpp>

#include <string>
#include <list>
#include <fstream>
#include <ostream>

#include "pattern_lib_mod.h"

namespace gsh
{
  /**
   * @brief Convert a string to a numeric value
   *
   * @param s A string representation of the numeric value
   * @return The numeric value of s in the given type.
   */
  template <typename _Tp>
  _Tp val_from_string(std::string s) {
    std::istringstream i(s);
    _Tp __r;
    i >> __r;
    return __r;
  }




  /**
   * @brief A structure that stores all of the searching
   *        parameters.
   *
   * This structure is a container that contains all the
   * graph searching parameters for the main program to use.
   */
  typedef struct search_params
  {
    int charge_min;
    int charge_max;
    float mz_tolerance;
    float mz_tolerance_2;
    float sigma;
    float per_sigma;
    float include_mass_mod;
    int scan_tolerance;
    size_t pattern_size;
    int min_score;
    std::string log_file;
    PatternLib<float,double>::pattern_type search_pattern;
    std::list<PatternLib<float,double>::pattern_type>
       alt_patterns;

    // file outputs:
    bool full_output;
    bool inclusion_list;
    bool mz_charge;
    bool mz_charge_scan;

    // construct everything to zero.
    search_params() : charge_min(0), charge_max(0),
       mz_tolerance(0.0),
      mz_tolerance_2(0.0), sigma(0.0), per_sigma(0.0),
         include_mass_mod(0.0),
      scan_tolerance(0), pattern_size(0), min_score(0),
         full_output(false),
      inclusion_list(false), mz_charge(false),
         mz_charge_scan(false){}

  } search_params;



  /**
   * @brief Normalizes the points in a pattern so that they
   *        sum to unity.
   *
   * @param p A pattern to be normalized.
   */
  void normalize_pattern(PatternLib<float,double>::pattern_type
     &p) {
    double sum = 0.0;
    for(size_t i = 0; i < p.pattern.size(); ++i)
      sum += p.pattern[i];
    if(sum <= 0.0) {
      std::cout << ”WARNING:␣null␣pattern␣encountered.” <<
         std::endl;
      return;
    }
    for(size_t i = 0; i < p.pattern.size(); ++i)
      p.pattern[i] /= sum;
  }




  /**
   * @brief Returns a vector of points from a comma-separated
   * list of values in a string.
   *
   * @param s A string of comma-separated floating point values.
   * @return A vector<float> of the values in the string
   */
  std::vector<float> parse_points(std::string s) {
    std::vector<float> __r;
    std::vector<std::string> strs;
    boost::split(strs, s, boost::is_any_of(","));
    for(size_t i = 0; i < strs.size(); ++i) {
      __r.push_back(val_from_string<float>(strs[i]));
    }
    return __r;
  }



  /**
   * @brief Parses out the isotopic pattern, prior probability,
   *        and molecular weight of a search pattern based on
   *        a string
   *
   * NOTE: function done.
   *
   * @param s A string containing the data to be parsed.
   * @return A pattern_type containing the information for a
   *          particular pattern.
   */
  PatternLib<float, double>::pattern_type
     parse_search_pattern(std::string s) {

    double mw = 0.0;
    double prior = 0.0;
    std::vector<float> points;

    PatternLib<float, double>::pattern_type __r;
    boost::cmatch what;
    std::string re =
       "^\\s*(search_pattern|alt_pattern)\\s*:(.*)$";
    if(regex_search(s.c_str(), what, boost::regex(re.c_str()
       ))) {
      std::string data(what[2].first, what[2].second);
      std::vector<std::string> strs;
      boost::split(strs, data, boost::is_any_of(";"));

      // we now have a list of individual data poitns...
      for(size_t i = 0; i < strs.size(); ++i) {
        re = "pattern\\s*=([0-9,\\.\\s]*)";
        if(regex_search(strs[i].c_str(), what,
             boost::regex(re.c_str() ))) {
           points = parse_points(std::string(what[1].first,
              what[1].second));
        }
        re = "mw\\s*=\\s*([0-9\\.]*)";
        if(regex_search(strs[i].c_str(), what,
             boost::regex(re.c_str() )))
           mw =
              val_from_string<double>(std::string(what[1].first,
              what[1].second));

                 re = "prior\\s*=\\s*([0-9\\.]*)";
        if(regex_search(strs[i].c_str(), what,
             boost::regex(re.c_str() )))
           prior =
              val_from_string<double>(std::string(what[1].first,
              what[1].second));

        std::cout << strs[i] << std::endl;
      }
    }
    else {
      return __r;
    }

    __r = PatternLib<float, double>::pattern_type(points,
       prior, mw);
    normalize_pattern(__r);
    return __r;
  }
```

```

/**
 * @brief Takes a file name and returns a list
 * of parameters after parsing the file.
 *
 * @param fname The name of the parameters file
 * @return The parameters that were able to be parsed
 * from the file.
 */
search_params read_parameters(char const* fname) {
  search_params __r;

  // set up some reading stuff..
  size_t blocksize = 1028;
  char memblock[blocksize];
  std::ifstream __in;
  __in.open(fname, std::ios::in);
  boost::cmatch what;

  std::cout << "Parsing the file " << fname << std::endl;

  // read the entire file line by line.
  while(__in.getline(memblock, blocksize)) {
    // parse out all comments:
    std::string data = std::string(memblock);
    if(regex_search(memblock, what, boost::regex("^([^#]*)")))
      data = std::string(what[1].first, what[1].second);

    // we can now check all the specifics.
    if(regex_search(data.c_str(), what,
       boost::regex("charge_min:\\s*([0-9]+)")))
      __r.charge_min =
         val_from_string<int>(std::string(what[1].first,
         what[1].second));
    if(regex_search(data.c_str(), what,
       boost::regex("charge_max:\\s*([0-9]+)")))
      __r.charge_max =
         val_from_string<int>(std::string(what[1].first,
         what[1].second));
    if(regex_search(data.c_str(), what,
       boost::regex("mz_tolerance:\\s*([0-9\\.]+)")))
      __r.mz_tolerance =
         val_from_string<float>(std::string(what[1].first,
         what[1].second));
    if(regex_search(data.c_str(), what,
       boost::regex("scan_tolerance:\\s*([0-9]+)")))
```

```
          __r.scan_tolerance =
             val_from_string<int>(std::string(what[1].first,
             what[1].second));
        if(regex_search(data.c_str(), what,
           boost::regex("pattern_size:\\s*([0-9]+)")))
          __r.pattern_size =
             val_from_string<size_t>(std::string(what[1].first,
             what[1].second));
        if(regex_search(data.c_str(), what,
           boost::regex("min_score:\\s*([0-9]+)")))
          __r.min_score =
             val_from_string<int>(std::string(what[1].first,
             what[1].second));
        if(regex_search(data.c_str(), what,
           boost::regex("log_file:\\s*(\\S+)")))
          __r.log_file = std::string(what[1].first,
             what[1].second);
        if(regex_search(data.c_str(), what,
           boost::regex("(search_pattern:.*)")))
          __r.search_pattern = parse_search_pattern(data);
        if(regex_search(data.c_str(), what,
           boost::regex("alt_pattern:")))
          __r.alt_patterns.push_back(parse_search_pattern(data));
        if(regex_search(data.c_str(), what,
           boost::regex("mz_tolerance_?2:\\s*([0-9\\.]+)")))
          __r.mz_tolerance_2 =
             val_from_string<float>(std::string(what[1].first,
             what[1].second));
        if(regex_search(data.c_str(), what,
           boost::regex("sigma:\\s*([0-9\\.]+)")))
          __r.sigma =
             val_from_string<float>(std::string(what[1].first,
             what[1].second));
        if(regex_search(data.c_str(), what,
           boost::regex("per_sigma:\\s*([0-9\\.]+)")))
          __r.per_sigma =
             val_from_string<float>(std::string(what[1].first,
             what[1].second));
        if(regex_search(data.c_str(), what,
           boost::regex("include_mass_mod:\\s*([0-9\\.]+)")))
          __r.include_mass_mod =
             val_from_string<float>(std::string(what[1].first,
             what[1].second));

        // what types of output are we interested in?
```

```
      if(regex_search(data.c_str(), what,
         boost::regex("full_output:\\s*true")))
        __r.full_output = true;
      if(regex_search(data.c_str(), what,
         boost::regex("inclusion_list:\\s*true")))
        __r.inclusion_list = true;
      if(regex_search(data.c_str(), what,
         boost::regex("mz_charge:\\s*true")))
        __r.mz_charge = true;
      if(regex_search(data.c_str(), what,
         boost::regex("mz_charge_scan:\\s*true")))
        __r.mz_charge_scan = true;

    }

    __in.close();

    // we need to resize the patterns to pattern_size;
    __r.search_pattern.pattern.resize(__r.pattern_size, 0.0);
    std::list<PatternLib<float,double>::pattern_type>::iterator
       __iter, __end = __r.alt_patterns.end();
    for(__iter = __r.alt_patterns.begin(); __iter != __end;
       ++__iter)
      __iter->pattern.resize(__r.pattern_size, 0.0);

    // we need to log the result somewhere
    if(__r.log_file.length() < 2)
      __r.log_file = "graph_search.log";

    return __r;
  }

}; // end namespace gsh
#endif // PARAMS_HELPER_H_INCLUDED
```

pattern_lib_mod.h

```
#ifndef PATTERN_LIB_MOD_H_INCLUDED
#define PATTERN_LIB_MOD_H_INCLUDED

#include <list>
#include "ms_pattern_lib.h"
```

```
#include "matched_pattern.h"



/**
 * This is a specialization of the PatternLibrary template that
 * can be used to match multiple patterns within a supplied
 * feature. The main difference between this function and its
 * parent is that this function returns a list of the first
 * points in sufficiently ‘good’ patterns (according to the
 * first supplied MsTag reference) rather than just the score
 * of that pattern. This is useful for building lists (or other
 * containers) of sufficiently good patterns found within
 * the data.
 */
template <typename _Tp1, typename _Tp2>
class PatternLibMod : public PatternLib<_Tp1,_Tp2>
{
  public:
    typedef typename PatternLib<_Tp1,_Tp2>::pattern_list_type
       pattern_list_type;

    PatternLibMod() : PatternLib<_Tp1,_Tp2>() {
       //!< Default constructor.
      is = new Isotope<_Tp1>();
    }



    /**
     * @brief A constructor that sets that parameters for sigma
     *        and percentage_sigma, giving the tolerance of
     *        pattern searching.
     *
     * @param sigma The absolute standard deviation of a
     *        pattern's intensity used for pattern matching.
     * @param per_sigma The percentage variance of a pattern's
     *        intensity used for pattern matching.
     */
    PatternLibMod(_Tp1 sigma, _Tp2 per_sigma) :
       PatternLib<_Tp1,_Tp2>(sigma, per_sigma) { is = new
       Isotope<_Tp1>();}

    std::list<MatchedPattern<_Tp1> > score_convoluted_feature(
      MsDetectedFeature<_Tp1> &_feat,
      pattern_list_type &_refs,
      unsigned short c_start,
      unsigned short c_end,
      unsigned short scan_num);

  private:
    Isotope<_Tp1> *is;

};




/**
 * @brief Given a feature and reference patterns, this function
 *        returns a list of matches that are sufficiently good.
 *
 * @param _feat   The feature to be scored - this does not need
 *        to be charge-state consistent.
 * @param _refs   The reference patterns to be scored against.
 *         The first pattern is the pattern that we are
 *         interested in, while the others serve as control
 *         patterns for scoring purposes.
 * @param c_start The lowest charge state that we are
 *        interested in.
 * @param c_end   The highest charge state that we are
 *        interested in
 * @param scan_num The current scan number
 */
template <typename _Tp1, typename _Tp2>
std::list<MatchedPattern<_Tp1> >
   PatternLibMod<_Tp1,_Tp2>::score_convoluted_feature(
MsDetectedFeature<_Tp1> &_feat,
pattern_list_type &_refs,
unsigned short c_start,
unsigned short c_end,
unsigned short scan_num)
{
  if(c_start > c_end) swap(c_start, c_end);

  // we can't handle zero charge here..
  assert(c_start > 0);

  std::vector<_Tp2> __tmp_probs;
  std::list<MatchedPattern<_Tp1> > __r;
```

```
std::vector<pair<_Tp2,_Tp2> > __probs;
std::vector<_Tp2>             __rms_error;
__probs.resize(_refs.size());
__rms_error.resize(_refs.size());
// __r.resize(_refs.size(), match_type());

// make sure that our pattern isn't too small...
if(_feat.size() < 3) {
  return __r;
}

size_t max_pattern_size = 0;

// copy the prior probabilities
for(size_t i = 0; i < _refs.size(); ++i) {
  if(_refs[i].pattern.size() > max_pattern_size)
    max_pattern_size = _refs[i].pattern.size();
  __probs[i].first = 0.0;
  __probs[i].second = _refs[i].prior_probability;
}

vector<vector<_Tp1> > c_pattern;
c_pattern.resize(_refs.size());

// Loop through all of the charge states that we
// possibly want to consider.
for(unsigned int charge = c_start; charge <= c_end; ++charge)
   {
  // get the feature at this charge..
  std::list<MsDetectedFeature<_Tp1> > tmp =
     get_feature_subset(_feat, charge, 0.03);


  /*
   * Now, we have a list of putative features that are
   * charge-state dependent, and consistent with respect
   * to the currently considered charge (charge). We need to
   * loop through all of them and score each one against
   * every pattern that we are considering.
   */
  typename std::list<MsDetectedFeature<_Tp1> >::iterator
     _tmp_iter, _tmp_end = tmp.end();
  for(_tmp_iter = tmp.begin(); _tmp_iter != _tmp_end;
     ++_tmp_iter) {
    // try all of the patterns.
```

```
_Tp1 mass = charge * get_mass(_tmp_iter->begin(),
   _tmp_iter->end() );


/*
 * We have a specific set of data peaks that we can
 * score, so we need to score it against all of the
 * patterns that we are considering. The first step
 * is to update each reference pattern with the
 * contaminating isotopes for a species of the
 * current molecular weight.
 */
for(size_t i = 0; i < _refs.size(); ++i) {
  if(mass > _refs[i].mass) {
    c_pattern[i] =
       is->update_distribution(_refs[i].pattern,
       is->null_distribution(mass - _refs[i].mass));
  }
  else {
    c_pattern[i] = _refs[i].pattern;
  }
  c_pattern[i].resize(max_pattern_size, 0.0);
}
// Each pattern is now convoluted with the isotopic
// distribution of a peptide of the current MW.

int loop_size = _tmp_iter->size() - max_pattern_size;


/*
 * Since the process up to this point may have produces a
 * list of peaks that are charge-state consistent but
 * *may* have been produced by >1 molecular species, we'll
 * loop through all subsets of the list that are
 * sufficiently long that we could score against them.
 * This feature may or may not be desirable, and more
 * testing should be done to establish this.
 */
typename MsDetectedFeature<_Tp1>::iterator __loop_iter =
   _tmp_iter->begin(), __loop_end = _tmp_iter->end();
while(--loop_size >= 0) {
  for(size_t i = 0; i < _refs.size(); ++i) {
    __probs[i].first = (__loop_iter == __loop_end ? 0.0 :
       get_aligned_score(c_pattern[i], __loop_iter,
       __loop_end));
```

```
                    __rms_error[i] = (__loop_iter == __loop_end ? 0.0 :
                       get_aligned_rms_error(c_pattern[i], __loop_iter,
                       __loop_end));
                }

                __tmp_probs = get_bayesian_probility(__probs);


                /*
                 * This is the only source of filtering that happens in
                 * this function, the rest is just looping and scoring.
                 * To tighten up the searching, more elaborate filtering
                 * can be added at this step. Alternatively, more
                 * elaborate filtering can be added at the total feature
                 * scoring / graph theoretic analysis of the
                 * putative matches.
                 */
                if(__tmp_probs[0] > 0.5 ) {
                    __r.push_back(MatchedPattern<_Tp1>(__loop_iter->mz,
                       __loop_iter->intensity, scan_num, charge,
                       __tmp_probs[0], __rms_error[0]));
                }
                ++__loop_iter;
            }

        }
    }

    return __r;
}
#endif    // PATTERN_LIB_MOD_H_INCLUDED
```

ms_tag.h

```
#ifndef MS_TAG_H_INCLUDED
#define MS_TAG_H_INCLUDED

/* this file contains two main structures, the MsTag
 * structure that contains isotopic distribution information,
 * prior probability, tag mass, and any other attributes that
 * may prove to be useful.
 *
 * The second class is an MsMatch type, which includes the
 * peaks that were matched to create the given pattern,
 * the probability, retention time, etc.
 */
#include "isotope.h"
#include "ms_feature.h"






/**
 * @struct MsTag
 * @brief Contains information on various tags including
 *        molecular weight, prior probability, and mass
 *
 * @var pattern The isotopic pattern of the tag
 * @var prior_probability The probability that any random
 *      molecular species would be labeled with this tag
 * @var mass The molecular weight of the tag
 */
template <typename _Tp>
struct MsTag
{
  typename Isotope<_Tp>::isotope_distribution pattern;
  _Tp prior_probability;
  _Tp mass;

  MsTag() : prior_probability(0.0), mass(0.0) { }
  MsTag(typename Isotope<_Tp>::isotope_distribution __p,
    _Tp prob, _Tp __m) : pattern(__p), prior_probability(prob),
    mass(__m) { }
};






/**
 * @struct MsMatch
 * @brief Contains information on a matched isotopic pattern,
 *        including the peaks that matched, the score of the
 *        match, and the scan number
 *
 * @var peaks An MsDetectedFeature of the peaks that matched
 *      the pattern
 * @var probability The score of the data against the pattern
 * @var score A secondary score that can be used along with
 *      probability for a match, e.g., weighted RMS difference.
 * @var scan_num The scan number in which the current match was
 *      located
 * @var charge The charge state of the best matched feature
 * @var start_mz The m/z value of the first peak in the best
 *      matched feature
 */
template <typename _Tp>
struct MsMatch
{
   MsDetectedFeature<_Tp> peaks;
   _Tp probability;
   _Tp score;
   unsigned short scan_num;
   unsigned short charge;
   _Tp start_mz;

   MsMatch() : probability(0.0), score(0.0), scan_num(0),
       charge(0), start_mz(0.0) { }
   MsMatch(MsDetectedFeature<_Tp> __peaks,
     _Tp prob, unsigned short scan, unsigned short __c, _Tp mz)
        : peaks(__peaks),
     probability(prob), scan_num(scan), charge(__c),
        start_mz(mz) { }

};
#endif      // MS_TAG_H_INCLUDED
```

matched_pattern.h

```
#ifndef MATCHED_PATTERN_H_INCLUDED
#define MATCHED_PATTERN_H_INCLUDED

#include <iostream>
#include <vector>
#include <list>
#include <math.h>

#include "ms_feature.h"

```

```
#ifndef SQUARE
#define SQUARE(x) ((x)*(x))
#endif




/**
 * @brief This class is derived from the generic MsPeak
 *        template, with an added variable to keep track
 *        of charge.
 */
template <typename _Tp>
class MatchedPattern : public MsPeak<_Tp>
{
  public:
    unsigned short charge;
    //!< The molecular weight of the parent ion of this match.
    _Tp parent_mass;
    //!< The probability of this match being real
    _Tp probability;
    //!< The secondary score of this match, e.g., the RMS
    // error between the best match and the reference.
    _Tp score;


    MatchedPattern() : MsPeak<_Tp>(), charge(0) { }

    MatchedPattern(_Tp __mz, _Tp __int, unsigned int __s,
       unsigned short __charge) :
    MsPeak<_Tp>(__mz, __int, __s), charge(__charge),
       probability(0.0), score(1000.0) {parent_mass = (__mz -
       1.0) * __charge; }

    MatchedPattern(_Tp __mz, _Tp __int, unsigned int __s,
       unsigned short __charge, _Tp prob, _Tp __score) :
    MsPeak<_Tp>(__mz, __int, __s), charge(__charge),
       probability(prob), score(__score) {parent_mass = (__mz -
       1.0) * __charge; }

    MatchedPattern(_Tp __mz, _Tp __int, unsigned int __s) :
       MsPeak<_Tp>(__mz, __int, __s), charge(0) ,
       probability(0.0), score(1000.0){ }

};
```

```
/**
 * This class is an encapsulation of lists of MatchedPattern
 * features that are supplied from a graph decomposition.
 * This class keeps track of elution range, charge states
 * found, parent ion mass, etc
 */
template <typename _Tp, unsigned int MAX_CHARGE>
class FinalFeature
{
  public:
    _Tp parent_mass;
    _Tp stddev;
    _Tp total_intensity;
    _Tp max_intensity;
    unsigned int scan_start;
    unsigned int scan_end;
    vector<bool> charge_states;
    int count;
    float score;
    float max_probability;
    float min_rms_error;

    FinalFeature(std::list<MatchedPattern<_Tp> > &_l);

};




/**
 * @brief Constructs a feature from a list of matched patterns
 *        that have been somehow correlated.
 *
 * @note This is the function that should be modified to alter
 *       feature scoring of subgraphs.
 *
 * @param _l A list of MatchedPattern objects used to construct
 *        a view of the mapped features
 */
template <typename _Tp, unsigned int MAX_CHARGE>
FinalFeature<_Tp,MAX_CHARGE>::FinalFeature(std::list<MatchedPattern<_Tp> > &_l)
{
  scan_start = 10000000;
  scan_end = 0;
  charge_states.resize(MAX_CHARGE + 1, false);

  // implement parent mass as an average.
  // NOTE: this could be done as an intensity-weighted average
  double tmp_parent_mass = 0.0;
  double SS_parent_mass = 0.0;
  total_intensity = 0.0;
  max_intensity = 0.0;
  count = 0;

  min_rms_error = 1000.0;
  max_probability = 0.0;

  typename std::list<MatchedPattern<_Tp> >::iterator __iter,
     __end = _l.end();

  for(__iter = _l.begin(); __iter != __end; ++__iter) {
    ++count;
    tmp_parent_mass += (__iter->mz - 1.0) * __iter->charge;
    charge_states[__iter->charge] = true;

    if( __iter->probability > max_probability )
      max_probability = __iter->probability;

    if( __iter->score < min_rms_error )
      min_rms_error = __iter->score;

    if(__iter->scan_num < scan_start)
      scan_start = __iter->scan_num;
    if(__iter->scan_num > scan_end)
      scan_end = __iter->scan_num;
    if(__iter->intensity > max_intensity)
      max_intensity = __iter->intensity;

    SS_parent_mass += SQUARE( (__iter->mz - 1.0) *
       __iter->charge);
    total_intensity += __iter->intensity;

  }

  if(count < 2) {
    stddev = -1.0;
  }
  else {
```

```
    stddev = SS_parent_mass - (SQUARE(tmp_parent_mass) / count);
    stddev /= count - 1;
    stddev = sqrt(stddev);
  }

  parent_mass = tmp_parent_mass / count;

  int c_states = 0;
  int max_charge_state = 0;
  for(size_t i = 0; i <= MAX_CHARGE; ++i) {
    if(charge_states[i]) {
      ++c_states;
      max_charge_state = i;
    }
  }

  // Scoring function stuff goes here.
  score = 2.0;

  if(c_states >= 2) score += 2.0;
  if(c_states >= 3) score += 2.0;
  if( (c_states == 1) && (charge_states[1]) ) score -= 2.0;
  if(count >= 10) score += 2.0;
  if(count >= 25) score += 2.0;
  if(count >= 50) score += 2.0;
  if(count >= 100) score += 2.0;
  if(count >= 250) score += 2.0;
  if(min_rms_error < 0.1) score += 2.0;
  if(min_rms_error < 0.05) score += 2.0;

  if(scan_end - scan_start >= 10) score += 2.0;
  if(max_charge_state >= 4) score += 2.0;

}
#endif    // MATCHED_PATTERN_H_INCLUDED
```

sample parameters file

```
# Charge states to consider:
charge_min: 1
charge_max: 5
```

```
# m/z tolerance - usually slightly looser than the actual
# instrumental requirements would dictate. A value of
# mz_tolerance: 0.025 works well for orbitrap data. The
# mz_tolerance_2 value is a parameters used to determine
# if peaks are potentially isotopically related, and is
# typically a smaller value than mz_tolerance.
# mz_tolerance_2 = 0.014 is a good value for orbitrap data.
mz_tolerance: 0.025
mz_tolerance_2: 0.014;


# the number of scans that two features can be separated by and
# still be considered 'related'
scan_tolerance: 2


# the number of points to be considered in matching a feature:
pattern_size: 6


# The minimum score for reported features:
min_score: 10


# The peak reported in an inclusion list. For example, with the
# dibrmoide tag it is desired that the middle (M+2) peak be used
# for an inclusion list as it is the most intense peak. Note
# that include_mass_mode accepts floating point values, but
# values that are significantly different from integer
# quantities do not make sense. This value defaults to 0.0
include_mass_mod: 2.0


# pattern matching parameters. Sigma is the absolute variance in
# peak signal intensity for pattern matching purposes, while
# per_sigma is the percentage variance in signal intensity for
# pattern matching purposes. The value of sigma does not seem to
# be overly important, while the value of per_sigma = 0.085
# appears to be idea for matching the dibromide pattern in
# orbitrap data.
sigma: 0.0
per_sigma: 0.085


# Name of file to log output:
log_file: ms_searching.log
```

```
# The pattern of interest to search for: as a list of peak
# intensities separated by 1 m/z unit. For example, the
# dibromide pattern would be input at
#     pattern: 0.25, 0, 0.5, 0, 0.25; mw=300
# where trailing zeroes are appended to reach the defined
# pattern size, and the entire pattern is normalized to a
# total signal of 1.0. Patterns longer than pattern_size
# are removed. The mw= tag is used to input the molecular
# weight of the pattern to be accounted for. This number
# plays a role in how the pattern is adjusted to account
# for contaminating isotopes in a data-dependent fashion.
# Using a large mw= tag will force the program to use the
# pattern unaltered during searching. The prior= tag is a
# weighting factor that is used in a Bayesian method to
# determine a final score for the search pattern accounting
# for the goodness of fit of the other alt_patterns defined
# below. Priors are effectively normalized to 1.0.
search_pattern: pattern=0.25, 0.0, 0.5, 0.0, 0.25; mw=100;
    prior=0.0001


# Alternative patterns to score against. At least one
# alternative pattern is necessary (usually
#     alt_pattern: 1; mw=0; prior=1),
# while additional patterns can be added empirically
# to reduce the occurance of false-positive matches.
alt_pattern: pattern=1; mw=0; prior=1;
alt_pattern: pattern=0.1, 0.5, 0.25; mw=300; prior=0.0001
alt_pattern: pattern=1, 1, 1, 1, 1, 1, 1; mw=10000; prior=0.001


# Outputs to use. Values are assumed to be false if not defined.
full_output: true        # A detailed output of all search
                         # results.
inclusion_list: false    # only m/z values
mz_charge: true          # m/z values with charge state
mz_charge_scan: true     # the m/z value, charge state, and scan
                         # numbers of detected features
```

B.5 Isotopic envelope computation

B.5.1 Description

This is an implementation of the isotopic envelope calculation algorithm discussed in Section 4.4. The present implementation only supports the biologically common elements (hydrogen, carbon, nitrogen, oxygen, sulfur) along with chlorine, bromine, and zinc. However, additional elements can be added readily, and comments can be found in the code indicating the location to do so.

B.5.2 C++ code

isotope.h

```
#ifndef ISOTOPE_H_INCLUDED
#define ISOTOPE_H_INCLUDED

#include <vector>
#include <iostream>

// A general minimum function
#ifndef __ms_min
#define __ms_min(x,y) ((x) < (y) ? (x) : (y))
#endif

/**
 * @class Isotope
 * @brief This class contains information for rapidly computing
 *        isotopic distributions of organic molecules
 *
 *        This class relies on the fundamental idea that the
 *        isotopic pattern for a molecule is idepdendent of the
 *        way in which the elements are grouped. For example,
 *        the isotopic pattern of C5 is the same as C(C4) =
 *        C((C2)2). From this, we can start with the basic
 *        distribution of the natural abundance of each isotope
 *        of interest, and then compute a `basis set' of
 *        isotopic distribution of the 2-powers of elements,
 *        e.g. C, C2, C4, C8, ... C256, etc. The isotopic
 *        distributions of more complex molecules can then
 *        be computed using the binary representation of the
 *        element counts. For exampe, to determine the isotopic
 *        pattern generated from C6H12O5N, we start with
 *           d = dist(C4),
 *         then:
 *           d = convolute(d, dist(C2))
 *           d = convolute(d, dist(H8))
 *           d = convolute(d, dist(H4))
 *           d = convolute(d, dist(O4))
 *           d = convolute(d, dist(O))
 *           d = convolute(d, dist(N))
 *         In addition, isotopic patterns of peptides can be
 *         predicted solely from their mass by using the
 *         `averagine' composition.
 */
template <typename _Tp>
class Isotope
{
  public:
    typedef std::vector<_Tp>  isotope_distribution;

    /**
     * @struct averagine
     * @brief Contains information on the `averagine' peptide.
     *
     * Each variable contains the expected number of elements of
     * that type that would be found in a peptide of molecular
     * weight 1.0. This is a useful format since the expected
     * number of, e.g., carbons in a peptide of mass M is
     * easily given by M * averagine::C
     */
    struct averagine
    {
      static const _Tp C = 0.044439885;    //!< Carbon
      static const _Tp N = 0.012217729;    //!< Nitrogen
      static const _Tp H = 0.069815722;    //!< Hydrogen
      static const _Tp O = 0.01329399;     //!< Oxygen
      static const _Tp S = 0.000375252;    //!< Sulfur
    };

    // some access constants -- in order to add
    // additional elements to this class, access constants
    // should be made for those elements.
    static const unsigned short bromine    = 0;
    static const unsigned short carbon     = 1;
    static const unsigned short hydrogen   = 2;
    static const unsigned short nitrogen   = 3;
    static const unsigned short oxygen     = 4;
    static const unsigned short sulfur     = 5;
    static const unsigned short chlorine   = 6;
    static const unsigned short zinc       = 7;
    static const unsigned short carbon13   = 8;
    static const unsigned short num_isotopes = 9;



    Isotope();
    isotope_distribution convolute(isotope_distribution &first,
       isotope_distribution &second);

    isotope_distribution convolute(
      isotope_distribution &first,
      isotope_distribution &second,
      size_t size);

    isotope_distribution
       update_distribution(isotope_distribution &__r,
       isotope_distribution second);

    isotope_distribution
       update_distribution(isotope_distribution &__r, const
       unsigned short type, unsigned int num);

    isotope_distribution null_distribution(_Tp mass);

    isotope_distribution get_distribution(unsigned short type,
      unsigned short count);

  private:
    typedef std::vector<isotope_distribution>
       isotope_distribution_list;

    void convolute_direct(unsigned short type,
      unsigned short index,
      isotope_distribution &ret);

    //!< A basis set of element compositions
    std::vector<isotope_distribution_list>   __basis_set;

};

/**
 * @brief The default constructor - initializes
 *         the isotopic basis sets
 */
```

```
template <typename _Tp>
Isotope<_Tp>::Isotope()
{
  isotope_distribution  tmp;
  isotope_distribution_list   tmp_arr;

  tmp_arr.push_back(tmp);

  __basis_set.resize(num_isotopes, tmp_arr);

  // bromine
  tmp.push_back(0.5069);
  tmp.push_back(0.0);
  tmp.push_back(0.4931);
  __basis_set[bromine][0] = tmp;

  // carbon
  tmp.clear();
  tmp.push_back(0.9893);
  tmp.push_back(0.0107);
  __basis_set[carbon][0] = tmp;

  // nitrogen
  tmp.clear();
  tmp.push_back(0.99636);
  tmp.push_back(0.00364);
  __basis_set[nitrogen][0] = tmp;

  // sulfur
  tmp.clear();
  tmp.push_back(0.9493);          // 32S
  tmp.push_back(0.0076);
  tmp.push_back(0.35904734);      // 34S
  tmp.push_back(0.0);
  tmp.push_back(0.0002);          // 36S
  __basis_set[sulfur][0] = tmp;

  // oxygen
  tmp.clear();
  tmp.push_back(0.99757);
  tmp.push_back(0.00038);
  tmp.push_back(0.00205);
  __basis_set[oxygen][0] = tmp;

  // hydrogen
  tmp.clear();
  tmp.push_back(0.999885);
  tmp.push_back(0.000115);
  __basis_set[hydrogen][0] = tmp;

  // carbon 13
  tmp.clear();
  tmp.push_back(0.0);
  tmp.push_back(1.0);
  __basis_set[carbon13][0] = tmp;

  // chlorine
  tmp.clear();
  tmp.push_back(0.7576);
  tmp.push_back(0.0);
  tmp.push_back(0.2424);
  __basis_set[chlorine][0] = tmp;

  // zinc
  tmp.clear();
  tmp.push_back(0.48268);    // 64Zn
  tmp.push_back(0.0);
  tmp.push_back(0.27975);    // 66-Zn
  tmp.push_back(0.04102);    // 67-Zn
  tmp.push_back(0.19024);    // 68-Zn
  tmp.push_back(0.0);
  tmp.push_back(0.00631);    // 70-Zn
  __basis_set[zinc][0] = tmp;

  // Add additional elements here by creating
  // a distribution of the natural abundances
  // of their isotopes relative to the lightest
  // existing isotopes, as can be seen above.

  for(size_t i = 0; i < __basis_set.size(); ++i) {
    for(int j = 0; j < 12; ++j) {
      __basis_set[i].push_back(convolute(__basis_set[i][j],
         __basis_set[i][j], 8));
    }
  }
}




/**
 * @brief Returns the combined isotopic signature
 *         of two separate patterns
 *
 * @param first   The first isotopic pattern
 * @param second  The second isotopic pattern
 * @return The convoluted isotopic signature of the
 *         two supplied distributions
 */
template <typename _Tp>
std::vector<_Tp> Isotope<_Tp>::convolute(
isotope_distribution &first,
isotope_distribution &second)
{
  isotope_distribution __r;
  if( (first.size() < 1) || (second.size() < 1) )
    return __r;

  __r.resize(first.size() + second.size() - 1, 0.0);

  for(size_t i = 0; i < first.size(); ++i)
    for(size_t j = 0; j < second.size(); ++j)
      __r[i+j] += first[i] * second[j];

  return __r;
}




/**
 * @brief Returns the combined isotopic signature
 *        of two separate patterns
 *
 * @param first   The first isotopic pattern
 * @param second  The second isotopic pattern
 * @return The convoluted isotopic signature of
 *         the two supplied distributions
 */
template <typename _Tp>
std::vector<_Tp> Isotope<_Tp>::update_distribution(
isotope_distribution &first,
isotope_distribution second)
{
  isotope_distribution __r;
  if( (first.size() < 1) || (second.size() < 1) )
    return __r;
  __r.resize(first.size() + second.size() - 1, 0.0);

  for(size_t i = 0; i < first.size(); ++i)
    for(size_t j = 0; j < second.size(); ++j)
      __r[i+j] += first[i] * second[j];

  return __r;
}




template <typename _Tp>
std::vector<_Tp> Isotope<_Tp>::update_distribution(
isotope_distribution &first,
const unsigned short type,
unsigned int num)
{
  if(num < 1) return first;
  return update_distribution(first, get_distribution(type,num));
}




/**
 * @brief Returns the combined isotopic
 *        signature of two separate patterns
 *
 * @param first    The first isotopic pattern
 * @param second   The second isotopic pattern
 * @param max_size The maximum length of the
 *        returned isotopic pattern
 * @return The convoluted isotopic signature of
 *        the two supplied distributions
 */
template <typename _Tp>
std::vector<_Tp> Isotope<_Tp>::convolute(
isotope_distribution &first,
isotope_distribution &second,
size_t max_size)
{
  isotope_distribution __r;
  if( (first.size() < 1) || (second.size() < 1) )
    return __r;
  max_size = __ms_min(max_size, first.size()
           + second.size() - 1 );
  __r.resize(max_size, 0.0);

  for(size_t i = 0; i < first.size(); ++i) {
    for(size_t j = 0; j < second.size(); ++j) {
      if( (i + j) >= max_size )
        break;

      __r[i+j] += first[i] * second[j];
    }
  }

  return __r;
}




/**
 * @brief Returns the default distribution for a peptide of a
 *        given mass
 *
 *        This function estimates the number of each elemental
 *        type based on the 'averagine' peptide composition,
 *        and then uses this prediction to estimate the
 *        isotopic pattern of a peptide of the provided
 *        molecular weight. For speed of calculation, this
 *        function only  includes terms which are estimated to
 *        contribute >5% to the final distribution.
 *
 * @param mass    The mass of the peptide
 * @return The estimated isotopic pattern of a peptide
 *        of the given mass
 */
template <typename _Tp>
std::vector<_Tp> Isotope<_Tp>::null_distribution(_Tp mass)
{
  isotope_distribution __r;
  if(mass <= 0) return __r;

  __r = get_distribution(carbon, (int)(averagine::C * mass *
     0.5) );

  // These cutoffs speed up computation considerably
  // by only considering elements that are likely
  // to be significant contributors to the net
  // distribution, but would lead to errors if the
  // results are used for further calculations.
  if(mass > 1880) __r = update_distribution(__r,
     get_distribution(oxygen, (int)(averagine::O * mass +
     0.5)));
  if(mass > 1145) __r = update_distribution(__r,
     get_distribution(nitrogen, (int)(averagine::N * mass +
     0.5)));
  if(mass > 4548) __r = update_distribution(__r,
     get_distribution(hydrogen, (int)(averagine::H * mass +
     0.5)));
  if(((int)(averagine::S * mass + 0.5)) > 0)
    __r = update_distribution(__r, get_distribution(sulfur,
       (int)(averagine::S * mass + 0.5)));

  return __r;
}




/**
 * @brief Returns the isotopic pattern of a given
 *        number of a given element
 *
 * @param type    The element the pattern is based on
 * @param count   The number of atoms of the given element
 * @return The isotopic pattern of the given number of
 *        atoms of a given type
 */
template <typename _Tp>
std::vector<_Tp> Isotope<_Tp>::get_distribution(unsigned short
   type,
unsigned short count)
{
  isotope_distribution __r;

  // We don't know this element
  if( type >= __basis_set.size() ) return __r;
  __r.push_back(1.0);

  // null distribution.
  if( count == 0 ) return __r;

  for(int i = __basis_set[type].size() -1; i >= 0; --i) {
    if((0x01 << i) <= count) {
      convolute_direct(type, i, __r);
      count -= (0x01 << i);
    }
  }

  return __r;
}




/**
 * @brief Convolutes the supplied pattern with a
 *        given number of elements of a given type
 *
 * @param type    The element the pattern is based on
 * @param count   The number of atoms of the given element
 * @param ret     The isotopic distribution that we are
 *                starting with
 */
template <typename _Tp>
void Isotope<_Tp>::convolute_direct(unsigned short type,
unsigned short index,
isotope_distribution &ret)
{
  if(type >= __basis_set.size() ) return;
  if(index >= __basis_set[type].size() ) return;

  ret = convolute(ret, __basis_set[type][index]);
}
#endif                                        // ISOTOPE_H_INCLUDED
```

B.6 Peak integration

B.6.1 Description

The following C++ files constitute a generalized mass spectral peak integration module, allowing for the relative quantification of three-dimensional mass spectral peaks. Individual peak_integrand objects define a bounded two-dimensional area over which the intensity should be integrated, defined by scan numbers in the range $[t_1, t_2]$ and

m/z values in the range $[mz - \delta, mz + \delta]$, where δ is a parameter defining the m/z width of each integrand. After processing all relevant data, each peak integrand will contain the value

$$\text{peak\_integrand}(m/z, t_1, t_2, \delta m) = \int_{t_1}^{t_2} \int_{m/z-\delta m}^{m/z+\delta m} I(m/z, t) d(m/z) dt \tag{B.1}$$

which can be obtained by calling the peak_integrand.integrand() method. The class ion_integrand is a wrapper class that contains a list of peak_integrand objects, each of which is initialized so as to represent isotopic peaks of a given molecular species at a defined charge state. The class mol_integrand is a further abstraction, which contains a list of ion_integrands, representing different charge states of the same molecular species.

The total integral values for each of these classes can be called through the .integrand() method. This is defined as

$$\text{ion\_integrand}(\vec{mz}, t_1, t_2, \delta m) = \sum_{mz_i \in \vec{mz}} \int_{t_1}^{t_2} \int_{mz_i-\delta m}^{mz_i+\delta m} I(mz, t) d(mz) dt \tag{B.2}$$

for the ion integrand class, where $\vec{mz}$ is the list of all m/z values in the ion, each representing an isotopic peak of the ion's isotopic envelope. For the mol_integrand object, ion_integrands are summed over charge states, and the total value then becomes

$$\begin{aligned}
&\text{mol\_integrand}(mass, c_{min}, c_{max}, t_1, t_2, \delta m) \qquad \text{(B.3)}\\
&= \sum_{c=c_{min}}^{c_{max}} \left(\sum_{mz_{i,c} \in \vec{mz}_c} \int_{t_1}^{t_2} \int_{mz_{i,c}-\delta m}^{mz_{i,c}+\delta m} I(mz, t) d(mz) dt \right)\\
&= \sum_{c=c_{min}}^{c_{max}} \text{ion\_integrand}(\vec{mz}_c, t_1, t_2, \delta m)
\end{aligned}$$

where $mz_{i,c}$ is the m/z value of the i^{th} isotopic peak of the ion with charge c, and $\vec{mz}_c$ is the list of all isotopic peaks for the ion with charge c.

The class integration_engine is the main engine that must be called to integrate peaks, and is operated in two distinct phases. In the first phase, pointers to peak_integrand objects are passed to the class through the push_back($\cdots$) method, building an internal list of peaks that must be integrated. After building this list, the class is locked to further addition of integrands, and raw MS data (in the form of lists $(m/z, intensity)$ pairs) is passed to the engine in order of scan number. After all data has been passed to the integration class, the data can be accessed through the original integrand objects (peak_integrand, ion_integrand, or mol_integrand). The integration_engine takes any of these data types when building its internal list through the push_back(· · ·) method.

The file input_parser.h includes code that can parse an input file containing information about peak-, ion-, or molecule integrands. The main function, main.cpp, takes two inputs: an input file containing information on the peaks to be integrated, and an mzXML file to be the source of the data. The file sample_input.txt gives an example of the formatting required for the first input.

B.6.2 C++ code

peak_integrand.h

```cpp
#ifndef PEAK_INTEGRAND_H_INCLUDED
#define PEAK_INTEGRAND_H_INCLUDED

#include <string>
#include <sstream>

/**
 * @class  peak_integrand This class stored information
 *          about a given area in an MS datafile that should
 *         be integrated, including thestarting and ending scans
 *         (time domain) and an m/z value with awindow (m/z
 *         domain). The object can be used to perform a two-
 *         dimensional integration of the form:
 *         \f[
 *            \int_{t_1}^{t_2}\int_{m/z - \delta}^{m/z + \delta}
 *                 I(t, m/z)d(m/z)dt
 *         \f]
 *         which is useful for quantitative applications of MS,
 *         including ICAT-like systems.
 */
template <typename _Tp>
class peak_integrand
{
  public:
    //    Ctors
    peak_integrand();
    peak_integrand(int start, int end, _Tp mz, _Tp delta):
       _start(start), _end(end), _mz(mz), _d_mz(delta),
       _val(0.0) {}
    ~peak_integrand(){};
```

```
/**
 * @brief  Adds an intensity value to the current integrand.
 * @param  intensity The value to be added to the current
 * integrand.
 */
void add(_Tp intensity) {
  _val += intensity;
}

/**
 * @brief  Returns the current value of the integrand, if
 *         integration is complete, this should be the value
 *         \f[
 *           \int_{t_1}^{t_2}
 *           \int_{m/z * - \delta}^{m/z + \delta}
 *             I(t, m/z) d(m/z)dt
 *         \f]
 *         if the integration has been completed.
 * @return  The current value of the integrand.
 */
_Tp integrand() {
  return _val;
}

/**
 * @brief  Returns the first scan of the integrand,
 *           \f$ t_1 \f$
 * @return  The first scan of the integrand, \f$ t_2 \f$
 */
int start() {
  return _start;
}

/**
 * @brief  Returns the final scan of the integrand,
 *           \f$ t_2 \f$
 * @return  The final scan of the integrand, \f$ t_2 \f$
 */
int end() {
  return _end;
}

/**
 * @brief  Returns the m/z value of the center of the
 *         peak.
 * @return  The m/z value of the center of the peak.
```

```
   */
  _Tp mz() {
    return _mz;
  }

  /**
   * @brief  Returns the distance from the center m/z value
   *         that will be included in the integrand,
   *         \f$ \delta \f$.
   * @return  The distance from the center m/z value that
   *          will be included in the integrand, \f$ \delta \f$.
   */
  _Tp delta() {
    return _d_mz;
  }

  /**
   * @brief Tests to see whether a given m/z value is within
   *        the range of the current integration object.
   * @param  mz The m/z value to be checked.
   * @return true if the supplied value is in the interval
   *         \f$ [m/z - \delta, m/z + \delta] \f$,
   *         false otherwise.
   */
  bool in_range(_Tp mz) {
    bool __r = false;
    if( (mz >= (_mz - _d_mz)) &&
      (mz <= (_mz + _d_mz)) )
      __r = true;
    return __r;
  }

  /**
   * @brief  Tests whether a given m/z value is less than than
   *         \f$ m/z - \delta \f$.
   * @param  mz The m/z value to be checked.
   * @return true if the supplied value is greater than
   *         \f$ m/z + \delta \f$, false otherwise.
   */
  bool is_before(_Tp mz) {
    bool __r = false;
    if( (_mz + _d_mz) < mz)
      __r = true;
    return __r;
  }
```

```
/**
 * @brief  Tests whether the given iterator is completely
 *         past the supplied m/z value.
 * @param  mz The m/z value to be checked.
 * @return true if the supplied value is less than
 *         \f$ m/z - \delta \f$, false otherwise.
 */
bool is_after(_Tp mz) {
  bool __r = false;
  if((_mz - _d_mz) > mz)
    __r = true;
  return __r;
}

/**
 * @brief  A comparator function that compares two
 *         integration objects by their starting scans.
 */
static bool by_start(peak_integrand<_Tp>& a,
    peak_integrand<_Tp> &b) {
  return (a.start() < b.start());
}

/**
 * @brief  A comparator function that compares two pointers
 *         to integration objects by their starting scans.
 */
static bool by_start_p(peak_integrand<_Tp>* a,
    peak_integrand<_Tp>* b) {
  return (a->start() < b->start());
}

/**
 * @brief  A comparator function that compares two pointers
 *         to integration objects by their starting scans.
 */
static bool by_end(peak_integrand<_Tp>& a,
    peak_integrand<_Tp> &b) {
  return (a.end() < b.end());
}

/**
 * @brief  A comparator function that compares two pointers
 *         to integration objects by their starting scans.
 */
```

```
  static bool by_end_p(peak_integrand<_Tp>* a,
     peak_integrand<_Tp>* b) {
    return (a->end() < b->end());
  }

  /**
   * @brief  A comparator function that compares two
   *          integration objects by their center m/z values.
   */
  static bool by_mz(peak_integrand<_Tp> &a,
     peak_integrand<_Tp> &b) {
    return (a.mz() < b.mz());
  }

  /**
   * @brief  A comparator function that compares two pointers
   *          to integration objects by their center m/z
   *          values.
   */
  static bool by_mz_p(peak_integrand<_Tp>* a,
     peak_integrand<_Tp>* b) {
    return (a->mz() < b->mz());
  }

  /**
   * @brief  A comparator function that compares two
   *          integration objects by their minimum m/z
   *          values, \f$ m/z - \delta \f$
   */
  static bool by_first_mz(peak_integrand<_Tp> &a,
     peak_integrand<_Tp> &b) {
    return ((a.mz() - a.delta()) < (b.mz() - b.delta()));
  }

  /**
   * @brief  A comparator function that compares two pointers
   *          to integration objects by their minimum m/z
   *          values,\f$ m/z - \delta \f$
   */
  static bool by_first_mz_p(peak_integrand<_Tp>* a,
     peak_integrand<_Tp>* b) {
    return ((a->mz() - a->delta()) < (b->mz() - b->delta()));
  }

  /**
   * @brief  Returns a string representation of the current
```

```
     *          integrand.
     * @return  A string representation of the current
     *           integrand object.
     */
    std::string print() {
      std::ostringstream _o;
      _o << "(" << _start << ",␣" << _end << ")␣-->␣" << _mz <<
         "␣+/-␣" << _d_mz << "␣(total=" << _val << ")";
      return _o.str();
    }

  protected:
    // The beginning scan number of the integrand, t_1
    int _start;

    // The ending scan number of the integrand, t_2
    int _end;

    // The center of the m/z region to be integrated
    _Tp _mz;

    // The distance from the center m/z value to be
    // integrated, \f$ m/z - \delta to m/z + \delta \f$
    _Tp _d_mz;

    // The total value of the integrand,
    // \f[
    // \int_{t_1}^{t_2}\int_{m/z  -
    // \delta}^{m/z + \delta} I(t,  m/z)
    // d(m/z)dt    \f]
    _Tp _val;
};
#endif                    //    PEAK_INTEGRAND_H_INCLUDED
```

ion_integrand.h

```
#ifndef ION_INTEGRAND_H_INCLUDED
#define ION_INTEGRAND_H_INCLUDED

#include <list>
#include <string>
#include <sstream>
```

```
#include ”peak_integrand.h”

/**
 * @class  ion_integrand Ion Integrand is a class that
 *         integrates the peaks for an entire ion over
 *         a given scan range, including a defined number
 *         of isotopic peaks for the ion and a width for the
 *         integration of each of those peaks. For each object,
 *         the total integration of the the isotopic peaks with
 *         \f$ N \f$ peaks in the envelope, will be given by
 *         \f$ \sum_{k=1}^N
 *             \int_{t_1}^{t_2}
 *             \int_{mz_k - \delta}^{mz_k + \delta}
 *             I(t, mz) d(mz)dt
 *         \f]
 *         where \f$ mz_k \f$ is the m/z value for the
 *         \f$ k^{th} \f$ peak in the isotopic envelope, and
 *         is determined based on the mass of the parent
 *         molecule and the charge state being considered as
 *         \f$ mz_k = (\f$base mass \f$ + k + \f$
 *             charge\f$ - 1) / \f$ charge.
 */
template <typename _Tp>
class ion_integrand
{
  protected:
    // The integrands of each individual isotopic peak.
    std::list<peak_integrand<_Tp> > peaks;

  public:
    typedef typename std::list<peak_integrand<_Tp> >::iterator
       integrand_iterator;

    /**
     * @brief  Returns an iterator to the beginning of the
     *         integrands list.
     */
    integrand_iterator begin() {
      return peaks.begin();
    }

    /**
     * @brief  Returns an iterator to the end of the
     *         integrands list.
     */
    integrand_iterator end() {
      return peaks.end();
    }

    /**
     * @brief  Returns the total integration of all peaks
     *         in the ion integrand.
     */
    _Tp integrand() {
      _Tp __r = 0.0;
      integrand_iterator __iter, __end = end();
      for(__iter = begin(); __iter != __end; ++__iter)
        __r += __iter->integrand();
      return __r;
    }

    //   Ctor
    ion_integrand(_Tp mass, int charge, _Tp delta, size_t
       n_peaks,
      int start, int end);

    /**
     * @brief  Prints out a string representation
     *         of the integrand.
     */
    std::string print() {
      std::ostringstream ost;
      integrand_iterator __iter, __end = end();
      for(__iter = begin(); __iter != __end; ++__iter)
        ost << __iter->print() << std::endl;

      return ost.str();
    }
};

/**
 * @brief  Constructor that makes an entire list of integration
 *         objects based on a provided scan range, an ion mass,
 *         ion charge, m/z tolerance, and number of peaks.
 * @param  mass The mass of the ion
 * @param  charge The charge of the ion.
 * @param  delta The tolernace in the m/z value. Integrattion
 *         will be over the range $$mz - \delta, mz + delta$$
 *         for each peak with m/z value $ mz $.
 * @param  n_peaks The number of peaks to be integrated
 *         for the ion.
 * @param  start The first scan to be integrated
 * @param  end The last scan to be integrated.
 */
template <typename _Tp>
ion_integrand<_Tp>::ion_integrand(_Tp mass, int charge, _Tp
      delta,
size_t n_peaks, int start, int end)
{
    for(size_t i = 1; i <= n_peaks; ++i) {
      _Tp mz = (mass + charge + i - 1.0) / ((_Tp)charge);
      peaks.push_back(peak_integrand<_Tp>(start, end, mz, delta));
    }
}
#endif                              //    ION_INTEGRAND_H_INCLUDED
```

mol_integrand.h

```
#ifndef MOL_INTEGRAND_H_INCLUDED
#define MOL_INTEGRAND_H_INCLUDED

#include <list>
#include <string>
#include <sstream>

#include ”ion_integrand.h”

/**
 * @class  mol_integrand Mol Integrand is a class that
 *         integrates the peaks for an entire molecule
 *         over a given scan range, for a given mass over
 *         an indicated number of charge states, for each
 *         charge state including a defined number of isotopic
 *         peaks for the ion and a width for the integration
 *         of each of those peaks. For each object, the total
 *         integration of the the isotopic peaks with \f$ N \f$
 *         peaks in the envelope, will be given by
 *         \f[
 *           \sum_{c \in charges}\sum_{k=1}^N \int_{t_1}^{t_2}
 *           \int_{mz_{k,c} - \delta}^{mz_{k,c} + \delta}
 *           I(t, mz) d(mz)dt
 *         \f]
 *         where \f$ mz_{k,c} \f$ is the m/z value for the
 *          \f$ k^{th} \f$ peak in the isotopic envelope with
 *          charge state \f$ c \f$, and is determined based on
 *          the mass of the parent molecule and the charge state
 *          being considered as \f$ mz_{k,c} = (\f$base
 *          mass\f$ + k + c - 1)
 *          / c \f$.
 */
template <typename _Tp>
class mol_integrand
{
  protected:
    std::list<ion_integrand<_Tp> > ions;

  public:
    typedef typename std::list<ion_integrand<_Tp> >::iterator
       ion_iterator;

    /**
     * @brief  Returns an iterator to the beginning of the
     *         ions list.
     */
    ion_iterator begin() {
      return ions.begin();
    }

    /**
     * @brief  Returns an iterator to the end of the
     *         ions list.
     */
    ion_iterator end() {
      return ions.end();
    }

    /**
     * @brief  Returns the integrand for all peaks
     *         integrated for the molecule.
     */
    _Tp integrand() {
      _Tp __r = 0.0;
      ion_iterator __iter, __end = end();
      for(__iter = begin(); __iter != __end; ++__iter)
        __r += __iter->integrand();
      return __r;
    }

    /**
```

```
   * @brief  Returns a string representation of all
   *         peaks in the mol_integrand.
   */
  std::string print() {
   std::ostringstream ost;
   ion_iterator __iter, __end = end();
   for(__iter = begin(); __iter != __end; ++__iter)
    ost << __iter->print() << std::endl;
   return ost.str();
  }

  mol_integrand(_Tp mass, int charge_min, int charge_max, _Tp
     delta, size_t n\
   _peaks, int start, int end);

};

/**
 * @brief  Constructor that makes an entire list of
 *         ion_integrand objects based on a provided
 *         scan range, charge range, an ion mass, m/z
 *         tolerance, and number of peaks per ion.
 * @param  mass The mass of the ion
 * @param  charge_min The minimum charge to consider
 * @param  charge_max The maximum charge to consider
 * @param  delta The tolernace in the m/z value. Integration
 *         will be over the range
 *           \f$ [mz - \delta, mz + delta] \f$
 *         for each peak with m/z value \f$ mz \f$.
 * @param  n_peaks The number of peaks to be integrated
 *         for each ion.
 * @param  start The first scan to be integrated
 * @param  end The last scan to be integrated.
 */
template <typename _Tp>
mol_integrand<_Tp>::mol_integrand(_Tp mass, int charge_min, int
   charge_max, _Tp delta, size_t n_peaks, int start, int end)
{
 for(int c = charge_min; c <= charge_max; c += 1.0)
  ions.push_back(ion_integrand<_Tp>(mass, c, delta, n_peaks,
     start, end));

}
#endif                     //   MOL_INTEGRAND_H_INCLUDED
```

integration_engine.h

```
#ifndef INTEGRATION_ENGINE_H_INCLUDED
#define INTEGRATION_ENGINE_H_INCLUDED

#include <vector>
#include <list>
#include <string>
#include <sstream>

#include "peak_integrand.h"
#include "ion_integrand.h"
#include "mol_integrand.h"

/**
 * @class integration_engine Integration engine is a class that
 *     provides a framework for performing two-dimensional
 *     integrations on LC-MS datasets over areas bounded in
 *     the time domain as well as in the m/z domain of the
 *     form \f[ \int_{t_1}^{t_2}\int_{m/z - \delta}^{m/z +
 *     \delta} I(t, m/z) d(m/z)dt \f] where the function \f$
 *     I(t,m/z)\f$ is the signal intensity at the time \f$
 *     t\f$ and the m/z value \f$ m/z \f$. This class works
 *     by adding data in two phases: first, all integration
 *     areas (of the type peak_integrand) are added to the
 *     working list. After all areas have been added, peak lists
 *     (in the form of an std::vector, in mz / intensity pairs)
 *     are analyzed sequentially by scan number. Upon the
 *     addition of each peak list, integration objects are
 *     moved to an active list if the current scan number is
 *     greater or equal to the starting scan for that object.
 *     Then, active integration objects are sorted by m/z
 *     values, and intensities that fall within an active
 *     object's integration range are added to the total
 *     integration of that object. Finally, after the scan is
 *     analyzed, integration objects that are "finished" are
 *     moved to a third list.
 */
template <typename _Tp>
class integration_engine
{
  protected:
    std::list<peak_integrand<_Tp> *> starting_integrands;
    // The initial list of  peak_integrands
    // as they are added to the   class.
    std::list<peak_integrand<_Tp> *> active_integrands;
    // The list of currently active   peak_integrands.
    std::list<peak_integrand<_Tp> *> finished_integrands;
    // The list of peak_integrands   which
    // have been completely    processed.

    typedef typename std::list<peak_integrand<_Tp> >::iterator
       integr\
     and_iterator;
    typedef typename std::list<peak_integrand<_Tp>*>::iterator
       integr\
     and_pointer_iterator;

    // A flag indicating whether or no the class is locked
    // to the addition of more peak_integrands.
    bool _locked;

    // A flag indicating whether or not there has been an error
    bool _err;

    // The number of the previous error.
    int  _err_no;

    // A description of the previous error.
    std::string _err_string;

    // The first scan required for integration.
    int _first_scan;

    // The last scan required for integration
    int _last_scan;

    // The next scan expected during integration.
    int _expected_scan;


    void integrate_scan(std::vector<float> &peaks);
    void update_active(int scan_num);
    void update_finished(int scan_num);

  public:
    integration_engine() : _locked(false), _err(false),
       _err_no(-1){}

    std::string print_starting();
    std::string print_active();
    std::string print_finished();
```

```
std::string print_all();

/**
 * @brief  Adds a pointer to a peak_integrand to the initial
 *         list.
 * @param  A pointer to a peak integrand which is to be
 *         added to the list.
 */
void push_back(peak_integrand<_Tp> * a) {
  if(!_locked) {
    starting_integrands.push_back(a);
  }
  else {
    _err = true;
    _err_no = 1;
    _err_string = "attempted␣to␣add␣peak_integrand␣to␣
       locked␣object.";
  }
}

/**
 * @brief  Adds a pointer to a peak_integrand to the initial
 *         list.
 * @param  An std::list<peak_integrand<_Tp> >::iterator for
 *         which the data should be added to the list.
 */
void push_back(integrand_iterator __iter) {
  if(!_locked) {
    starting_integrands.push_back(&*__iter);
  }
  else {
    _err = true;
    _err_no = 1;
    _err_string = "attempted␣to␣add␣peak_integrand␣to␣
       locked␣object.";
  }
}

/**
 * @brief  Adds a pointer to a peak_integrand for each
 *         integrand in an ion_integrand object.
 * @param  a A reference to an ion_integrand object.
 */
void push_back(ion_integrand<_Tp> &a) {
  integrand_iterator __iter, __end = a.end();
  for(__iter = a.begin(); __iter != __end; ++__iter)
```

```
      push_back(__iter);
  }

  /**
   * @brief  Adds a pointer to a peak_integrand for each
   *         integrand in each ion of a mol_integrand objet.
   * @param  a A reference to a mol_integrand object.
   */
  void push_back(mol_integrand<_Tp> &a) {
    typename mol_integrand<_Tp>::ion_iterator __iter, __end =
       a.end();
    for(__iter = a.begin(); __iter != __end; ++__iter)
      push_back(*__iter);
  }

  /**
   * @brief  Returns the current error state of the class.
   * @return  True if an error has been encoutered, false
   *          otherwise.
   */
  bool err() {
    return _err;
  }

  /**
   * @brief  Returns a description of the last error.
   * @return  A description of the last error.
   */
  std::string err_string() {
    return _err_string;
  }

  /**
   * @brief  Returns the number of the previous error.
   * @return  The number of the previous error.
   */
  int err_no() {
    return _err_no;
  }

  /**
   * @brief  Clears the error status of the class.
   */
  void clear() {
    _err = false;
  }
```

```
    int first_scan();
    int last_scan();
    void lock();

    void add_scan(std::vector<float> &peaks, int scan_num);
    bool done();

    void sort_active() {
      active_integrands.sort(peak_integrand<_Tp>::by_start_p);
    }

};


/**
 * @brief Prints all of the integrands in the
 *        initial list, which are integrands that
 *        have not yet been activated.
 */
template <typename _Tp>
std::string integration_engine<_Tp>::print_starting()
{
  std::ostringstream __o;
  typename std::list<peak_integrand<_Tp> *>::iterator __iter,
     __end =\
     starting_integrands.end();
  for(__iter = starting_integrands.begin(); __iter != __end;
     ++__iter)
    __o << (*__iter)->print() << std::endl;

  return __o.str();
}



/**
 * @brief Prints all of the active integrands.
 */
template <typename _Tp>
std::string integration_engine<_Tp>::print_active()
{
  std::ostringstream __o;
  typename std::list<peak_integrand<_Tp> *>::iterator __iter,
     __end =\
     active_integrands.end();
  for(__iter = active_integrands.begin(); __iter != __end;
     ++__iter)
    __o << (*__iter)->print() << std::endl;

  return __o.str();
}



/**
 * @brief Prints the contents of the finished integrands.
 */
template <typename _Tp>
std::string integration_engine<_Tp>::print_finished()
{
  std::ostringstream __o;
  typename std::list<peak_integrand<_Tp> *>::iterator __iter,
     __end =\
     finished_integrands.end();
  for(__iter = finished_integrands.begin(); __iter != __end;
     ++__iter)
    __o << (*__iter)->print() << std::endl;

  return __o.str();
}



/**
 * @brief Prints the contents of all internal lists.
 */
template <typename _Tp>
std::string integration_engine<_Tp>::print_all()
{
  std::ostringstream __o;
  __o << "----------------\nStarting:" << std::endl;
  __o << print_starting() << std::endl;
  __o << "----------------\nActive:" << std::endl;
  __o << print_active() << std::endl;
  __o << "----------------\nFinished:" << std::endl;
  __o << print_finished() << std::endl;
  return __o.str();
}
```

```

/**
 * @brief  Returns the first scan in the starting list.
 */
template <typename _Tp>
int integration_engine<_Tp>::first_scan()
{
  if(!_locked) {
    _err = true;
    _err_no = 2;
    _err_string = "attempted␣to␣read␣scan␣information␣before␣
       locking␣\
class.";
  }
  return _first_scan;
}



/**
 * @brief  Returns the first scan in the starting list.
 */
template <typename _Tp>
int integration_engine<_Tp>::last_scan()
{
  if(!_locked) {
    _err = true;
    _err_no = 2;
    _err_string = "attempted␣to␣read␣scan␣information␣before␣
       locking␣\
class.";
  }
  return _last_scan;
}



/**
 * @brief  Locks the object so that no more integration objects
 *         can be added, determines the first and last scans
 *         that  are necessary, and allows the actual
    integration
 *         processes to begin.
 */
template <typename _Tp>
void integration_engine<_Tp>::lock()
{
  if(_locked) return;

  _locked = true;

  // determine the first    scan.
  if(starting_integrands.size() < 1) return;

  starting_integrands.sort(peak_integrand<_Tp>::by_start_p);
  integrand_pointer_iterator __iter =
     starting_integrands.begin();
  _first_scan = (*__iter)->start();

  starting_integrands.sort(peak_integrand<_Tp>::by_end_p);
  starting_integrands.reverse();
  __iter = starting_integrands.begin();
  _last_scan = (*__iter)->end();

  // we want to end with everything sorted in order    by
  // starting scan    number.
  _expected_scan = _first_scan;
  starting_integrands.sort(peak_integrand<_Tp>::by_start_p);
}




/**
 * @brief  Integrates the next scan in the file.
 * @param  peaks The MS data to be integrated, in mz/intensity
 *         pairs with m/z values at even indices.
 * @param  scan_num The scan number from which the data came.
 *      Scans are expected to be processed in numerical order.
 */
template <typename _Tp>
void integration_engine<_Tp>::add_scan(std::vector<float>
   &peaks,
            int scan_num)
{
  if(!_locked) {
    _err = true;
    _err_no = 3;
    _err_string = "data processed before object locked.  
```

```
      Results␣may␣be␣inconsistent.";
  }

  // if this scan is outside of the range,
  // the rest of the function will do nothing.
  if(scan_num < _first_scan) return;
  if(scan_num > _last_scan) return;

  if(scan_num != _expected_scan) {
    // this is an error.
    _err = true;
    _err_no = 4;
    _err_string = "scans␣not␣processed␣in␣numerical␣order.␣␣
      Integration␣results␣may␣not␣be␣correct.";
  }

  update_active(scan_num);
  integrate_scan(peaks);
  update_finished(scan_num);

  // increment the expecte scan:
  _expected_scan += 1;
}




/**
 * @brief  Updates the list of active integrands based on the
 *         current scan number.
 * @param  scan_num The scan number from which the current
 *         set of peaks came from.
 */
template <typename _Tp>
void integration_engine<_Tp>::update_active(int scan_num)
{
  integrand_pointer_iterator __iter, __end =
     starting_integrands.end();
  for(__iter = starting_integrands.begin(); __iter != __end;) {
    if((*__iter)->start() == scan_num) {
      active_integrands.push_back(*__iter);
      __iter = starting_integrands.erase(__iter);
    }
    else {
      ++__iter;
    }
  }
}




/**
 * @brief  Updates the list of finished integrands based on the
 *         current scan number.
 */
template <typename _Tp>
void integration_engine<_Tp>::update_finished(int scan_num)
{
  integrand_pointer_iterator __iter, __end =
     active_integrands.end();
  for(__iter = active_integrands.begin(); __iter != __end;) {
    if((*__iter)->end() == scan_num) {
      finished_integrands.push_back(*__iter);
      __iter = finished_integrands.erase(__iter);
    }
    else {
      ++__iter;
    }
  }
}




/**
 * @brief  Integrates the current scan, analyzing integration
 *         objects that are currently in the active_integrands
 *         list. It is important that this list be up-to-date
 *         before calling integrate scan.
 * @param  peak A list of MS peaks in ms/intensity pairs
 *         with m/z values at even inidices.
 */
template <typename _Tp>
void integration_engine<_Tp>::integrate_scan(std::vector<float>
   &peaks)
{
  if(peaks.size() < 2) return;

  // sort the active integrands by starting m/z    value.
  active_integrands.sort(peak_integrand<_Tp>::by_mz_p);
  integrand_pointer_iterator __iter = active_integrands.begin();
  integrand_pointer_iterator __mid1 = __iter;
  integrand_pointer_iterator __mid2 = __iter;
  integrand_pointer_iterator __end = active_integrands.end();


  /* At each increment through the peaks array, the iterators
   * __mid1 and __mid2 are updated so that all integrands that
   * contain the point peaks[i] are in the range [__mid1,
   * __mid2). After these boundaries are setup, __iter is
   * allowed to iterate over the range, integrating all peaks
   * in that set. */

  /* NOTE: This loop is currently O(n^2), and it should be
   * possible to reduce it to O(n) with smart limits on the
   * range of integrands that is iterated over in each step
   * of the loop. This can be done if performance is an
   * issue. */

  for(size_t i = 0; i < peaks.size(); i += 2) {
  // initialize the starting   point
  /* while( ( __mid1)->is_before(peaks[0]) && (__mid1 != __end))
   * ++__mid1; // update the ending point while( !(
   * __mid2)->is_after(peaks[0]) && (__mid2 != __end)) ++__mid2;
   * std::cout << ”__mid1 == __mid2 ? ” << (__mid1 == __mid2 ? ”
   * yes ” : ” no”) << ”\n”; */


    for(__iter = __mid1; __iter != __end; ++__iter) {
      if((*__iter)->in_range(peaks[i]))
        (*__iter)->add(peaks[i+1]);
    }
  }
}




/**
 * @brief  Tests whether all objects in the class have been
 *         completely integrated.
 * @return  True if all integrands have been finished, false
 *          otherwise.
 */
template <typename _Tp>
bool integration_engine<_Tp>::done()
```

```
{
  bool __r = true;
  if(starting_integrands.size() > 0) __r = false;
  if(active_integrands.size() > 0 )  __r = false;
  return __r;
}
#endif              //      INTEGRATION_ENGINE_H_INCLUDED
```

input_parser.h

```
#ifndef MS_INTEGRAND_PARSER_H_INCLUDED
#define MS_INTEGRAND_PARSER_H_INCLUDED

#include <fstream>
#include <sstream>
#include <boost/regex.hpp>

#include ”peak_integrand.h”
#include ”ion_integrand.h”
#include ”mol_integrand.h”

namespace integrand_parser
{

  /**
   * @brief Converts a string to a numerical value
   * of the specified type.
   */
  template <typename _Tp>
  _Tp from_string(std::string s) {
    std::istringstream __i(s);
    _Tp __r;
    __i >> __r;
    return __r;
  }

  /**
   * @brief Parses an input file to obtain lists of peak,
   * ion, and molecular integrands, and adds their values to
   * the supplied lists.
   *
   * @param fname The file name of the list to be parsed.
   * @param PI  A reference to a list of peak integrands.
   * @param II  A reference to a list of ion integrands.
   * @param MI  A reference to a list of mol integrands.
   */
  template <typename _Tp>
    void parse(char* fname, std::list<peak_integrand<_Tp> > &PI,
    std::list<ion_integrand<_Tp> > &II,
  std::list<mol_integrand<_Tp> > &MI) {
    boost::cmatch what;
    size_t blocksize = 1024;
    char memblock[blocksize];
    std::string data;

    std::ifstream __in(fname, std::ios::in);
    // read the entire file.
    while(__in.getline(memblock, blocksize)) {

      // Comment lines start with a #.
      if(boost::regex_search(memblock, what,
         boost::regex("^\\s*#")))
        continue;

      // peak integrands
      // peak: mz, delta, start, stop
      std::string re;
      re  = "^\\s*peak:\\s*([0-9\\.]+),\\s*([0-9\\.]+),";
      re += "\\s*([0-9]+),\\s*([0-9]+)";
      if(boost::regex_search(memblock, what,
         boost::regex(re.c_str()))) {
        _Tp mz = from_string<_Tp>(std::string(what[1].first,
           what[1].second));
        _Tp delta = from_string<_Tp>(std::string(what[2].first,
           what[2].second));
        int start = from_string<int>(std::string(what[3].first,
           what[3].second));
        int stop  = from_string<int>(std::string(what[4].first,
           what[4].second));

        PI.push_back(peak_integrand<_Tp>(mz, delta, start,
           stop));
      }

      // ion integrands
      // ion: mass, charge, n-peaks, delta, start, stop
      re  = "^\\s*ion:\\s*([0-9\\.]+),\\s*([0-9]+),";
      re += "\\s*([0-9]+),\\s*([0-9\\.]+),\\s*";
```

```
re += "([0-9]+),\\s*([0-9]+)";
if(boost::regex_search(memblock, what,
   boost::regex(re.c_str()))) {
  // parse a bunch of values from here..
  _Tp mass = from_string<_Tp>(std::string(what[1].first,
     what[1].second));
  int charge =
     from_string<int>(std::string(what[2].first,
     what[2].second));
  size_t n_peaks =
     from_string<size_t>(std::string(what[3].first,
     what[3].second));
  _Tp delta = from_string<_Tp>(std::string(what[4].first,
     what[4].second));
  int start = from_string<int>(std::string(what[5].first,
     what[5].second));
  int stop  = from_string<int>(std::string(what[6].first,
     what[6].second));

  II.push_back(ion_integrand<_Tp>(mass, charge, delta,
     n_peaks, start, stop));
}

// molecule integrands:
// mol: mass, charge-start, charge-end, n-peaks, delta,
// start, stop
re  = "^\\s*mol:\\s*([0-9\\.]+),\\s*([0-9]+),\\s*";
re += "([0-9]+),\\s*([0-9]+),\\s*([0-9\\.]+),\\s*";
re += "([0-9]+),\\s*([0-9]+)";
if(boost::regex_search(memblock, what,
   boost::regex(re.c_str()))) {
  _Tp mass = from_string<_Tp>(std::string(what[1].first,
     what[1].second));
  int charge1 =
     from_string<int>(std::string(what[2].first,
     what[2].second));
  int charge2 =
     from_string<int>(std::string(what[3].first,
     what[3].second));
  size_t n_peaks =
     from_string<size_t>(std::string(what[4].first,
     what[4].second));
  _Tp delta = from_string<_Tp>(std::string(what[5].first,
     what[5].second));
  int start = from_string<int>(std::string(what[6].first,
     what[6].second));
```

```
        int stop   = from_string<int>(std::string(what[7].first,
           what[7].second));

        MI.push_back(mol_integrand<_Tp>(mass, charge1, charge2,
           delta, n_peaks, start, stop));
      }

    }// end while(file...)

    __in.close();

  }// end parse(...)

};// end namespace
#endif // MS_INTEGRAND_PARSER_H_INCLUDED
```

main.cpp

```
#include <iostream>
#include <list>
#include <algorithm>
#include <boost/assign/std/vector.hpp>

#include "peak_integrand.h"
#include "integration_engine.h"
#include "ion_integrand.h"
#include "mol_integrand.h"
#include "ms_integrand_parser.h"

// mzXML reading utilities
#include "../shared_lib/mzxml_lib/mzxml_reader.h"

using namespace std;
using namespace boost::assign;

int main(int argc, char** argv)
{
  // The integrand types that we may care about...
  list<peak_integrand<float> > PI;
  list<ion_integrand<float> > II;
  list<mol_integrand<float> > MI;
```

```
integration_engine<float> IE;

string mzxml_file, ions_file;

ions_file = std::string(argv[1]);
mzxml_file = std::string(argv[2]);

cout << "FILE␣NAME:␣" << ions_file << endl;
integrand_parser::parse(ions_file, PI2, II2, MI2);


/*
 * Load up the integration engine with all of the peaks,
 * molecules, ions, etc, that we want to integrate.
 */
list<peak_integrand<float> >::iterator piter, pend = PI.end();
list<ion_integrand<float> >::iterator iiter, iend = II.end();
list<mol_integrand<float> >::iterator miter, mend = MI.end();

for(piter = PI.begin(); piter != pend; ++piter)
  IE.push_back(piter);

for(iiter = II.begin(); iiter != iend; ++iiter)
  IE.push_back(*iiter);
  cout << iiter->print() << endl;

for(miter = MI.begin(); miter != mend; ++miter)
  IE.push_back(*miter);

// The integration engine has been loaded,
// lock it and parse a file.
IE.lock();



// load the mzXML file and read the index and headers.
mzxml::Reader infile(mzxml_file.c_str());
infile.read_index_offset() ? cout << "index␣offset␣read." <<
    endl : cout << "unable␣to␣read␣index␣offset" << endl;
infile.read_index() ? cout << "index␣read." << endl : cout <<
    "unable␣to␣read␣index." << endl;
infile.read_instrument_header() ? cout << "instrument␣header␣
    read." << endl : cout << "unable␣to␣read␣instrument␣
    header." << endl;
infile.read_run_header() ? cout << "run␣header␣read." << endl
    : cout << "unable␣to␣read␣run␣header" << endl;
  cout << "Scan count: " << infile.get_scan_count() << endl;

  cout << "Integrating scans " << IE.first_scan() << " to " <<
     IE.last_scan() << endl;

  for(int scan = IE.first_scan(); scan <= IE.last_scan();
     ++scan){
    if( scan > infile.get_scan_count() ){
      cout << "Warning: integration attempted past mzXML file 
         end. Data may not be valid." << endl;
      break;
    }

    size_t peaks_count;
    float* peaks = infile.read_scan(scan, peaks_count);
    vector<float> v_peaks(&peaks[0],&peaks[peaks_count]);
    IE.add_scan(v_peaks, (int)scan);

    // Check for errors.
    if(IE.err()){
      cout << "Error processing scan " << scan << ": " <<
        IE.err_no() << ": " << IE.err_string() << endl;
      IE.clear();
    }

  }

  // Done reading mzXML
  infile.close();

  // When finished, this list should be empty:
  cout << "All starting: " << endl;
  cout << IE.print_starting() << endl;

  // This list should contain all integrands. Integrands are
  // also accessible through the originally created objects
  // PI, II, and MI. These can easily be output to a file.
  cout << "All finished: " << endl;
  cout << IE.print_finished() << endl;

  return 1;
}
```

sample_input.txt

```
# This is a sample file containing information on peaks, ions,
# or molecules that should be integrated.  Each file can contain
# any number of any of these types.

# Integrate a peak:
peak: 510.25, 0.01, 2, 7;


# Ion integrand:
# ion: mass, charge, n-peaks, delta, start, stop
ion: 2000.5, 2, 5, 0.025, 1, 5


# Molecule integrand:
# mol: mass, charge_min, charge_max, n-peaks, delta, start, stop
mol: 1700.23, 1, 3, 3, 0.008, 4, 7
```

appendix c

Peak detection using a continuous wavelet transform

C.1 Overview

Mass spectrometers collect data that define peaks by a distribution of points. Such full scan data is termed 'profile' data. Before useful information can be extracted from this data, peaks must be identified. There are a variety of methods that can be employed to do this. For our purposes, there are a couple desired features – *m/z* values should be preserved as accurately as possible, and relative intensities should be identified accurately, noise should be reduced, and baseline should be eliminated. A version of the continuous wavelet transform (CWT) proposed in by Du *et al.* is ideal for these purposes as it estimates the area under the curve of a given peak, accurately identifies the *m/z* value, automatically corrects the baseline, and eliminates a majority of high-frequency noise.[1] Mathematically, the CWT is a two-dimensional transform defined as:

$$W(n,s) = \sum_{n'=0}^{N-1} x_{n'} \sqrt{\frac{\delta t}{s}} \Psi \left[\frac{(n'-n)\delta t}{s} \right] \tag{C.1}$$

where $\Psi(t)$ is the mother wavelet function, n is the index of the transform, x_n is the intensity data at the index n, and s is a scaling factor for the wavelet (larger s implies a wider wavelet). For our purposes, we used the mexican hat wavelet, which is a normalized second derivative of the gaussian function (a plot of which can be seen in Figure C-1):

$$\Psi(t,\sigma) = \frac{2}{\sqrt{3\sigma}\pi^{1/4}} \left(1 - \frac{t^2}{\sigma^2} \right) e^{-t^2/2\sigma^2} \tag{C.2}$$

In non-mathematical terms, the transform works by convoluting a peak-shaped function (termed the mother wavelet) with the sample data and integrating the result-

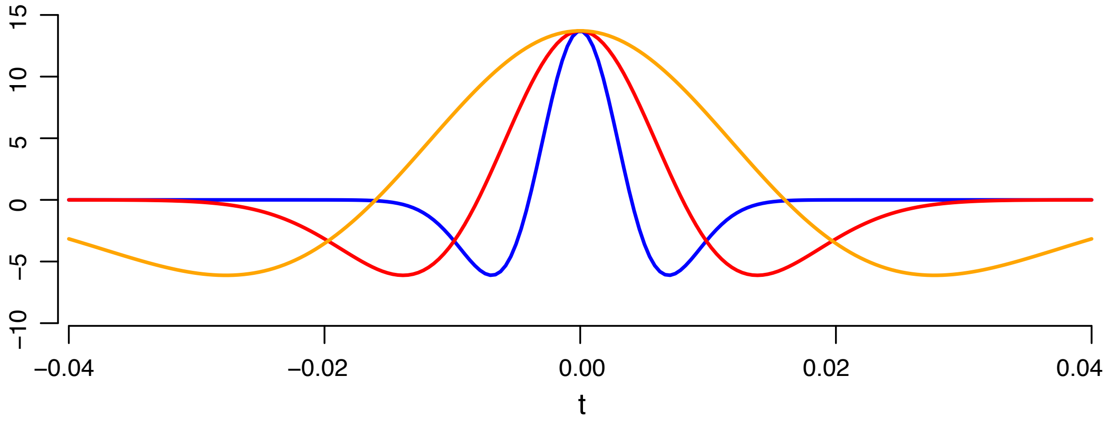


Figure C-1: A plot of the mexican hat wavelet with $\sigma = 0.004$ scaled at three different values of s: $s = 4$ (orange), $s = 2$ (red) and $s = 1$ (blue).

ing function at each point. Peak detection based on this transform starts by using a wavelet that is slightly wider than the average peaks in the data. Local maxima are detected in the transform of the data with this wavelet. The wavelet is then narrowed slightly, and the data transformed with this wavelet. Local maxima from this second transform are then compared with the original list of maxima and peaks within a certain tolerance are correlated together. Peaks from the original transform that are not correlated are flagged as having missed a 'scan'. These peaks are then removed if they do not correlate on the next round of the transform. If the new maximum is of higher intensity, this intensity and *m/z* value are recorded[*]. Peaks in the newer scan that are not correlated to the original scan are added to the searching list, and the process is continued using an even narrower wavelet until a wavelet that is narrower than all sample peaks is reached. As a side note, some caution must be used in choosing the proper starting and ending wavelets – by starting too wide, resolution is lost and peaks may not be identified. On the other hand, there are two potential problems if the wavelet is allowed to become too narrow – a narrower wavelet implies a higher frequency, so much of the noise filtering ability of the algorithm is lost if the wavelet is allowed to become too narrow. In the extreme, the wavelet can behave as a delta function, returning the sample data amplified to the extreme. In this case, no filtering of any sort is achieved and the feature detection algorithm breaks down.

The transform of a sample area of raw data can be seen in Figures C-2 and C-3.

[*] The maximum intensity for a given scan will occur when a wavelet that is the same width as the peak is used in the transform. This wavelet should most accurately estimate both the *m/z* value as well as the area under the curve of the peak.

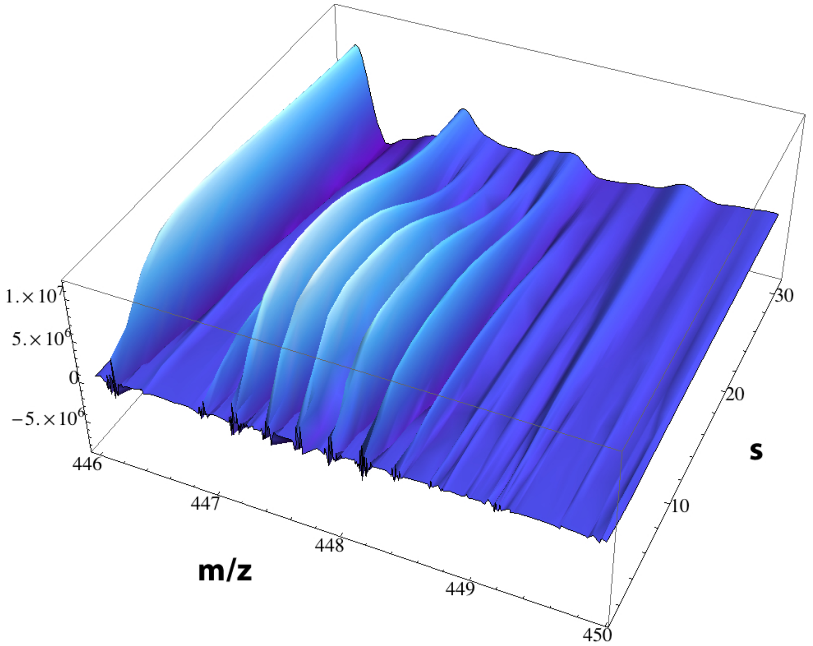


Figure C-2: A 3-dimensional plot of the continuous wavelet transform on a small area of a mass spectrum showing a synthetic dibromide-labeled glycopeptide. At larger values of s, the ridges are smooth and broad. As s is decreased, the ridges reach a maximum value depending on the shape of the actual peak in the raw data. At this maximum value, the CWT accurately estimates the area under the curve of that peak as well as the *m/z* value. As s is further increased, noise becomes more apparent and intensity then falls again. At even smaller s (not shown), the wavelet begins to behave as a delta function, and no filtering occurs.

The method in which the continuous wavelet transform resolves peaks can be most easily seen in Figure C-2, while the algorithm used to identify these peaks is probably most apparent from Figure C-3. The end result of this process can be seen in Figure C-4, where the original raw data is compared to the detected peaks.

C.2 Algorithm

The CWT peak detection and de-noising algorithm used by iteratively computing the one-dimensional transform where the value of s is fixed. The parameter s is iterated through a predetermined range of values, and at each value of s, the one-dimensional transform

$$W_s(n) = \sum_{n'=0}^{N-1} x_{n'} \sqrt{\frac{\delta t}{s}} \Psi \left[\frac{(n'-n)\delta t}{s} \right] \tag{C.3}$$

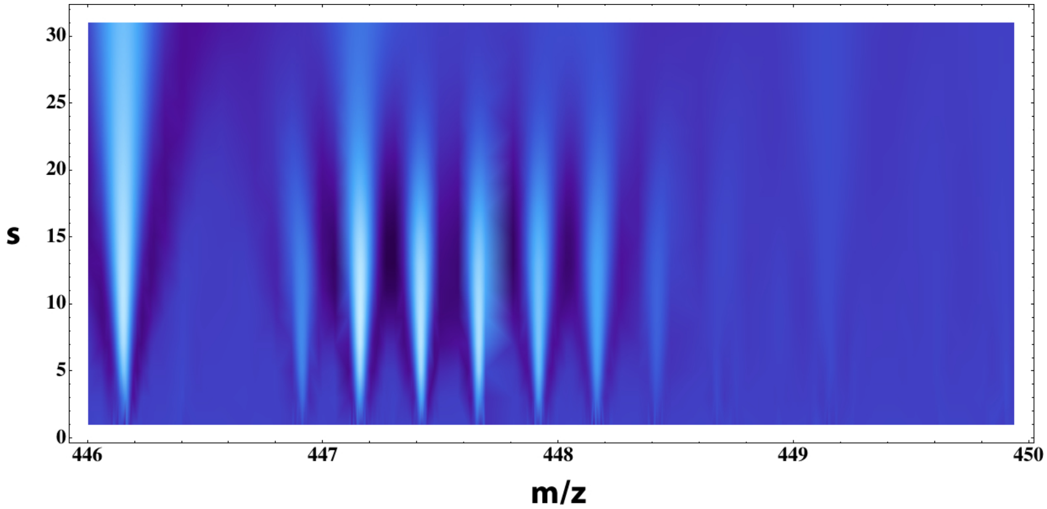


Figure C-3: A density plot of the same data shown in Figure C-2. Light colors represent high CWT coefficient, while dark colors represent low CWT coefficients. Peaks have a very clear ridge-line which can be traced from high values of s (top) to lower values of s (bottom).

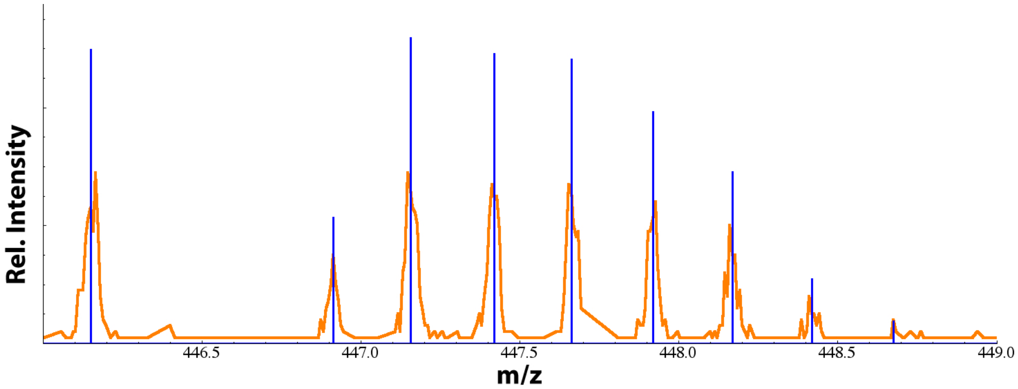


Figure C-4: The raw data (orange) and the identified peaks (blue) for the same data presented in Figures C-2 and C-3. The absolute intensities have been normalized separately, so they cannot be expected to line up exactly. However, the relative intensities between peaks are extremely accurate, as are the *m/z* assignments given to the peaks by the algorithm.

is computed, and local maxima are identified. s is then incremented to the next value, and the same processes repeated to determine a second set of local maxima, which is then compared to local maxima from the previous value of s. If the new list contains a maximum value within a given tolerance of a value in the old list, that list is update with the new data, and information about the number of times that maximum has been updated is stored along with information about the length of the ridge (in s-units). After each round of the transform, local maxima that have not been updated for a given number of scans are discarded, and maxima that were not updated in the previous round are marked as having "missed" an additional scan.

After all desired values of s are analyzed, the finalized list of local maxima are processed to remove ridges that are insufficient in length in the s-dimension. After this filtering, remaining ridges are turned into m/z-intensity pairs by determining the maximum intensity along the ridge, and the corresponding m/z value at this point. At the highest intensity along the peak ridge, the wavelet used approximates the profile of the peak, and the corresponding transform is proportional to the area under the curve[1].

C.3 Performance

On a modern desktop computer (8GB RAM, 2.66 GHz), the provided code can processes an orbitrap mzXML data file containing 5000-9000 full scan mass spectra in 5-10 minutes.

C.4 C++ code

The code for the continuous wavelet transform is stored in three files: first, cwt_pd.h and cwt_pd.cpp contain the code for the actual wavelet transform. The final file, cwt_peak_detection.cpp contains code to processes mzXML files, performing peak CWT-transform peak detection on each full scan mass spectrum.

cwt_pd.h

```
#ifndef CWT_CWT_H
#define CWT_CWT_H

#include <string>
#include <vector>
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include "DibroDatatypes.h"

using namespace std;

#define SIZE_FILE 512

```

```
/**
 * @class CWT This class provides functionality for performing
 * a continuous wavelet transform on mass spectral data.
 * Parameters for the transform are passed to the constructor
 * upon initialization,  after which any number of mass spectral
 * scans can be analyzed using the transform.
 *
 * The value of sigma, a parameter of key importance, should be
 * dependent upon the type of instrumentation being used. In my
 * hands, sigma = 0.0025 works very well for orbitrap data
 * (typical error in estimating peak integration is < 1%), and
 * a value of around 0.005 works well for Q/ToF data. For _best_
 * results, this should be optimized to each instrument, though
 * the algorithm is quite robust and a rough estimate (or even
 * a static value of ~ 0.005) will give reasonable (error in
 * peak integration < ~5%) results.
 */
class CWT
{

  public:
   CWT(double sigma = 0.005, unsigned int s_min = 5,unsigned
       int s_max = 20, unsigned int s_step = 5, unsigned int
       g_max = 3, unsigned int min_length = 3);

    void cwt(float * pData);
    void cwt(dbdt::data_t &data);

  private:
        // A few numerical constants:
    double psi_scale;
    double psi_sigma;
    double _1_O_PS_2;
    double _PS_2;

    float       psi(double t);
    float       transformPoint(dbdt::data_t &data, int index,
       int s, float bounds, float delta_t);
    float       transformPoint(float * data, int index, int s,
       float bounds, float delta_t, int dataSize);
    void        cleanMaxima(vector<dbdt::local_max_t>
      &mtMaxima,unsigned int &index, int minLength);
    void        consolidateMaxima(vector<dbdt::local_max_t>
      &mtMaxima, unsigned int &mtIndex, dbdt::data_t
      &tmpMaxima, unsigned int tmIndex, float corrDist);
    dbdt::local_max_t   setLMT(float mz, float intensity, int g, int length);

    /** a few parameters: **/
    unsigned int    _S_MAX, _S_MIN, _S_STEP;
    unsigned int    _G_MAX, _MIN_LEN;

};
#endif // CWT_CWT_H
```

cwt_pd.cpp

```
#include <math.h>
#include <iostream>

#include "cwt_pd.h"

using namespace std;

/**
 * @brief The standard constructor.
 *
 * @param sigma A parameter specifying the overall width of
 *          each wavelet.
 * @param s_min The minimum value of the variable s to be used.
 * @param s_max The maximum value of the variable s to be used.
 * @param s_step The step size between values of s to be used.
 * @param g_max The maximum size of a gap in a ridge before a
 *        peak is discarded.  Small values are more stringent,
 *        but values of ~2-3 are usually reasonable.
 * @param min_length The minumum length, in s-numbers of a ridge
 *        before it is considered to be a peak.
 */
CWT::CWT(double sigma, unsigned int s_min, unsigned int s_max, unsigned int s_step, unsigned int g_max, unsigned int min_length)
{
  psi_sigma = sigma;
  _1_O_PS_2 = 2.0 / (sqrt(psi_sigma * 3) * pow(3.14159, 0.25))
  _PS_2 = psi_sigma * psi_sigma;

  _G_MAX = g_max;
  _MIN_LEN = min_length;
  _S_MIN = s_min;
  _S_MAX = s_max;
  _S_STEP = s_step;
}


/**
 * @brief The mexican hat wavelet.
 *
 * the width of this should depend on the instrument being used-
 * pick a value so that the initial hat is slightly narrower
 * than the average peak width. If too narrow of a wavelet
 * function is used, the mother wavelet starts to behave like a
 * delta function with the result is that the original data is
 * scaled and returned with no filtering done.
 */
float  CWT::psi(double t)
{
  psi_scale = _1_O_PS_2 * (1  + (-t * t) / (_PS_2));

  return psi_scale * exp((-t * t) / (2 * _PS_2));
}


/**
 * @brief Calculates the continuous wavelet transform
 * on the currently loaded dataset, and automatically replaces
 * the dataset with the processed peak list.
 *
 * Several optimizations have been included in this function,
 * which may make it unusable for application outside of the
 * analysis of full-scan MS data without first validating
 * behavior.
 *
 * @param data The raw MS data
 */
void  CWT::cwt(dbdt::data_t &data )
{

  assert(data.size() > 1);

  // an array of local maxima
  vector<dbdt::local_max_t> mtMaxima;
  // a temporary array of local maxima.
  dbdt::data_t tmpMaxima;
```

```
// push_back() for vectors in O(n), so repeated calls to this
// function lead to slow run time. Instead, it is better to
// just waste the memory to start with to shorten run time
mtMaxima.reserve(data.size());

dbdt::point ptTmp;
ptTmp.mz = 0.0; ptTmp.intensity = 0.0;
tmpMaxima.resize(data.size(), ptTmp);

dbdt::point ptCur, ptPrev;
float delta_t = 0.0075f;

// depends on sigma.
float bounds = 3.0 * psi_sigma;
// the distance allowed between correlated slices.
float corr_dist;

// The correlation distance should be a function
// of psi_sigma, the width of the wavelet.
float corr_scale_dist = psi_sigma/2.0;


int step;
bool increasing = false;
unsigned int tmpIndex = 0;
unsigned int index = 0;

for(unsigned int s = _S_MAX; s >= _S_MIN; s -= _S_STEP) {

  corr_dist = s * corr_scale_dist;

  // go in steps that are dependent on s -- this significantly
  // reduces the time it takes to run, especially at higher
  // values of s, but the steps should be small enough that
  // detail isn't missed.
  step = max((int)(s / 6), 1 );
  increasing = false;
  ptPrev.mz = data[0].mz;
  ptPrev.intensity = transformPoint(data, 0, s, bounds,
     delta_t);

  tmpIndex = 0;

  // loop through the data at this value of s
  for(unsigned int i = 1; i < data.size(); i++) {
      ptCur.mz = data[i].mz;
      ptCur.intensity = transformPoint(data, i, s, bounds,
         delta_t);

      // check for local maxima
      if(increasing && (ptCur.intensity < ptPrev.intensity)) {
        // the previous point was actually the maximum
        tmpMaxima[tmpIndex++] = ptPrev;
      }

      increasing = (ptCur.intensity > ptPrev.intensity);
      ptPrev = ptCur;
    }

    // consolidate the maxima, tracing the ridge
    consolidateMaxima(mtMaxima, index,  tmpMaxima, tmpIndex,
       corr_dist);


  }// end for(s...

  cleanMaxima(mtMaxima,index, _MIN_LEN);
  data.resize(index);

  // at this point, mtMaxima should actually
  // contain all of the data that we want..
  for(unsigned int i = 0; i < data.size(); i++) {
    data[i].mz = mtMaxima[i].mz;
    data[i].intensity = mtMaxima[i].intensity;
  }
}


/**
 * @brief integrates the convolution of the data with
 * the wavelet function at the supplied index.
 *
 * @param data The raw data.
 * @param index The current index of the data
 * @param s The current value of the scaling factor s
 * @param bounds The m/z width over which the transform
 *        will be calculated (data[index] +/- bounds)
 * @param delta_t The sampling interval.
 * @return The continuous wavelet transform of the data
 *        at the supplied point.
 */
```

```
float CWT::transformPoint(dbdt::data_t &data, int index, int s,
    float bounds, float delta_t)
{
  float ret = 0.0;

  // estimate a good starting index for j -- this depends on the
  // wavelet coefficient s, delta_t (the sampling interval),
  // and bounds, which depends on psi_sigma, which defines the
  // width of the wavelet and is dependent on the resolution of
  // the instrument.
  int j = max(-1, (int)((index - ((bounds * s)/(delta_t)))));

  // increment until we are within the proper range,
  // j >= index - (bounds * s), which is equivalent to
  while( (data[j++].mz < (data[index].mz - (bounds * s))) );

  while(((data[++j].mz - data[index].mz) < (bounds*s)) && (j <
      (int)data.size()) ) {
    ret += data[j].intensity * psi( (data[index].mz-data[j].mz)
        / ((float) s) );
  }

  return ret;
}


/**
 * @brief integrates the convolution of the data with
 * the wavelet function at the supplied index.
 *
 * @param data The raw data.
 * @param index The current index of the data
 * @param s The current value of the scaling factor s
 * @param bounds The m/z width over which the transform
 *         will be calculated (data[index] +/- bounds)
 * @param delta_t The sampling interval.
 * @param dataSize The number of points in the dataset
 * @return The continuous wavelet transform of the data
 *         at the supplied point.
 */
float CWT::transformPoint(float * data, int index, int s, float
    bounds, float delta_t, int dataSize)
{
  float ret = 0.0;

  // estimate a good starting index for j -- this depends on the
  // wavelet coefficient s, delta_t (the sampling interval),
  // and bounds, which depends on psi_sigma, which defines the
  // width of the wavelet and is dependent on the resolution of
  // the instrument.
  int j = max(-2, (int)((index - (2* ((bounds *
     s)/(delta_t))))));

  // we are dealing with (mz/intensity) pairs, so index and j
  // need to be separated by a factor of 2n */
  if(!((j-index)%2)) --j;

  // increment until we are within the proper range,
  // j >= index - (bounds * s), which is equivalent to
  while( (data[j] < (data[index] - (bounds * s))) && (j <
     dataSize) ) {
    j += 2;
  }

  while(((data[j] - data[index]) < (bounds*s)) && (j <
     dataSize) ) {
    ret += data[j+1] * psi( (data[index]-data[j]) / ((float) s)
      );
    j += 2;
  }

  return ret;
}


/**
 * @brief Compares two lists of local maxima, correlating
 * peaks if they are within the m/z tolerance, otherwise
 * inserting a new maximum into the list before returning.
 *
 * @param mtMaxima The running list of maxima.
 * @param mtIndex
 * @param tmpMaxima A new list of local maxima.
 * @param tmpIndex
 * @param corrDist the maximum distance within which two
 *                 peaks will be correlated.
 */
void CWT::consolidateMaxima(vector<dbdt::local_max_t>
   &mtMaxima, unsigned int mtIndex, dbdt::data_t &tmpMaxima,
   unsigned int tmpIndex, float corrDist)
{
  if(tmpIndex <= 0) return;    // nothing to do.

  vector<dbdt::local_max_t> mtTmp;
  dbdt::local_max_t tmpCorr;

  // a temporary storage point..
  tmpCorr = setLMT(0.0, 0.0, 0, 0);

  // reserve enough space for a worse case scenario
  mtTmp.resize(mtIndex + tmpIndex, tmpCorr);

  // the current index of mtTmp that we are at..
  unsigned int index = 0;

  unsigned int i = 0;
  unsigned int j = 0;

  while( (i < mtIndex) && (j < tmpIndex) ) {

    // we don't want to add negative values -- this
    // can be an artefact from the transformation
    if( (tmpMaxima[j].intensity < 0.0) ) {
      j++; continue;
    }

    if(fabs(mtMaxima[i].mz - tmpMaxima[j].mz) < corrDist) {
      tmpCorr = mtMaxima[i];
      while( (j < tmpIndex) && (fabs(mtMaxima[i].mz -
           tmpMaxima[j].mz) < corrDist) ) {
        // find the highest correlating peak..
        if(tmpMaxima[j].intensity > tmpCorr.intensity) {
          tmpCorr.mz = tmpMaxima[j].mz;
          tmpCorr.intensity = tmpMaxima[j].intensity;

        }

        j++;
      }
      tmpCorr.g = 0;
      tmpCorr.length++;
      i++;
    }
    else {
      if(mtMaxima[i].mz < tmpMaxima[j].mz) {
        // add mtMaxima[i] to the list...
        tmpCorr = mtMaxima[i];
        tmpCorr.g++;
        tmpCorr.length++;
        i++;
      }
      else {
        // add tmpMaxima[j] to the list...
        tmpCorr = setLMT(tmpMaxima[j].mz,
           tmpMaxima[j].intensity, 0, 1);
        j++;
      }
    }// end if(fAbs...
    // add the result to the growing list..
    if(tmpCorr.g < _G_MAX)
      mtTmp[index++] = tmpCorr;

  }// end while...

  // add which ever list is not empty to the end...
  for(unsigned int k = i; k < mtIndex; k++) {
    // add the rest of this list to the end..
    mtMaxima[k].g++;
    mtMaxima[k].length++;
    if(mtMaxima[k].g < _G_MAX)
      mtTmp[index++] = tmpCorr;
  }

  for(unsigned int k = j; k < tmpIndex; k++) {
    if( (tmpMaxima[j].intensity < 0.0) ) continue;
    tmpCorr = setLMT(tmpMaxima[k].mz, tmpMaxima[k].intensity,
       0, 1);
    mtTmp[index++] = tmpCorr;
  }

  tmpCorr = setLMT(0.0, 0.0, 0, 0);
  mtIndex = index;
  if(mtIndex > mtMaxima.size()) {
    mtMaxima.resize(index, tmpCorr);
  }

  for(i = 0; i < index; i++) {
    mtMaxima[i] = mtTmp[i];
  }

}
```

```
/**
 * @brief Create a new local_max_t object from an
 * mz/intensity pair, a g-value, and a length.
 *
 * @param mz The m/z value
 * @param intensity The intensity of the point.
 * @param g The current g value
 * @param length The current length
 * @return a local_max_t object with the supplied parameters.
 */
dbdt::local_max_t CWT::setLMT(float mz, float intensity, int g,
   int length)
{
  dbdt::local_max_t ret;
  ret.mz = mz;      ret.intensity = intensity;
  ret.g = g;        ret.length = length;
  return ret;
}



/**
 * @brief Cleans up a list of local maxima by removing
 * points that are no longer being considered.
 *
 * @param mtMaxima A list of local maxima.
 * @param index the current index.
 * @param minLength The minimum length of a peak ridge to
 *        be saved. Shorter ridges are discarded.
 */
void CWT::cleanMaxima(vector<dbdt::local_max_t> &mtMaxima,
   unsigned int index, int minLength)
{
  vector<dbdt::local_max_t> ret;
  unsigned int size = 0;
  for(unsigned int i = 0; i < index; i++) {
    if(mtMaxima[i].length >= minLength) size++;
  }
  ret.resize(size);
  size = 0;
  for(unsigned int i = 0; i < index; i++) {
    if(mtMaxima[i].length >= minLength) ret[size++] =
       mtMaxima[i];
  }
  index = size;
  for(unsigned int i = 0; i < index; i++) {
      mtMaxima[i] = ret[i];
  }
}




/**
 * @brief A wrapper for the function void cwt(dbdt::data_t
   &data)
 * that accepts an array of float values, performs the
    transform, and then converts the
 * return back to a float array. Not the best solution, but a
    temporary fix.
 *
 * @param pData an array of floating point values
 *        representing MS data in mz/intensity
 *        pairs, with mz values at even indicies.
 */
void CWT::cwt(float * pData)
{
  // a wrapper for the cwt function above that accepts and
  // returns an array of float values.
  dbdt::data_t     data;
  dbdt::point      point;

  int index = 0;
  while(pData[index] >= 0) {
    point.mz = pData[index++];
    point.intensity = pData[index++];
    data.push_back(point);
  }
  cwt(data);
  pData = (float *)realloc(pData, 2 * sizeof(float) *
     (data.size() + 1));
  index = 0;
  for(unsigned int i = 0; i < data.size(); i++) {
    pData[index++] = data[i].mz;
    pData[index++] = data[i].intensity;
  }
  pData[index++] = -1.0;
  pData[index] = -1.0;

  return;

}
```

cwt_peak_detection.cpp

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "mzxml_lib/mzxml_writer.h"
#include "mzxml_lib/mzxml_reader.h"

#include "cwt_pd.h"
#include "DibroDatatypes.h"

using namespace std;
using namespace mzxml;

/**
 * @brief This program performs a scan-wise peak detection
 * on an mzXML datafile.  In the current version, only
 * full scan mass spectra data are considered, though
 * this can be easily modified.
 *
 * Command line syntax:
 * cwt_peak_detection -i <infile.mzXML> -o <outfile.mzXML>
 */
int main(int argc, char** argv)
{
  char * ifName = (char *)"";   // = "";
  char * ofName = (char *)"";   // = "";
  for(int i = 0; i < argc; i++) {
    if(strstr(argv[i], "-i") != NULL) {
      ifName = argv[i+1];
    }
    if(strstr(argv[i], "-o") != NULL) {
      ofName = argv[i+1];
    }
  }

  ostringstream ost;

  // this is only applicable when processing MS data that
  // includes tandem data.  In the case that only full scans
  // are processed, the map is created to show the mapping of
  // scan numbers between the unprocessed and processed files.
```

```
ostringstream oost;
oost << ofName << ".map";
oost.flush();
string scanMap = oost.str();

cout << "saving␣scan␣map␣in␣" << scanMap << endl;
ofstream fScanMap;
fScanMap.open(scanMap.c_str(), ios::out);


/** initialize reader and writer. **/
Reader reader;
Writer writer;

/** set parameters. **/
CWT *cwt = new CWT(0.005, 5, 20, 5, 3, 3);

RunHeaderStruct            runHeader;
InstrumentStruct           instrument;
ScanHeaderStruct       scanHeader;
vector<ParentFileStruct> parentFiles;

cout << "ifName␣=␣" << ifName << endl;

if(!writer.open(ofName)) exit(1);
if(!reader.open(ifName));

if(!reader.is_open()) {
  cout << "could␣not␣open␣" << ifName << "␣for␣input." <<
     endl;
  ost << "could␣not␣open␣" << ifName << "␣for␣input.";
  printLog(ost, PID);
}
if(!writer.is_open()) {
  cout << "could␣not␣open␣" << ofName << "␣for␣output." <<
     endl;
  ost << "coult␣not␣open␣" << ofName << "␣for␣output.";
  printLog(ost, PID);
}

cout << "C" << endl;

cout << "Processing␣file␣" << ifName << endl;
cout << "␣␣>␣outputting␣\"" << ofName << "\"" << endl;

cout.flush();
```

```
// Attempt to open and initialize the mzXML input file.
ost << "reading index offset........ " <<
   (reader.read_index_offset() ? "OK" : "FAILED");
ost << "reading index.............. " << (reader.read_index()
   ? "OK" : "FAILED");
ost << "reading instrument header... " <<
   (reader.read_instrument_header() ? "OK" : "FAILED");
ost << "reading parent file info.... " <<
   (reader.read_parent_file_header() ? "OK" : "FAILED");
ost << "reading run header.......... " <<
   (reader.read_run_header() ? "OK" : "FAILED");

// write the new header
runHeader = reader.get_run_header();
instrument = reader.get_instrument_header();
parentFiles = reader.get_parent_file_header();

int scanCount = reader.get_scan_count();
writer.write_header(runHeader, instrument, parentFiles);
int index = 1;
double retTimeFinal = 0.0;
int realScanCount = 0;

// Processes every scan.
while(reader.scan_exists(index)) {
  reader.read_scan_header(index);
  scanHeader = reader.get_scan_header();

  // only analyze full-scan mass spectra.
  if(scanHeader.ms_level > 1) {
    cout << "skipping MS2" << endl;
    index++;
    continue;
  }

  realScanCount++;
  fScanMap << index << "--->" << realScanCount << endl;

  cout << "scan " << index << " of " << scanCount << endl;

  reader.read_scan_header(index);
  scanHeader = reader.get_scan_header();
  retTimeFinal = scanHeader.retention_time;
  float * pData1 = reader.read_scan(index++);
```

```
        // The actual conversion.
        cwt->cwt(pData1);
        writer.write_scan(scanHeader, pData1);
    }

    //writer->adjustParams(retTimeFinal, realScanCount);

    writer.close();
    reader.close();

    // close the scan mapping file.
    fScanMap.close();

    ost << "Conversion completed successfully, " << (index - 1)
       << " scans written.";
    cout << ost.str() << endl;
    printLog(ost, PID);

    #ifndef __GNUC__
    getchar();
    #endif

    return 0;
}
```

References

[1] Du, P.; Kibbe, W. A.; Lin, S. M. *Bioinformatics* **2006**, *22*, 2059–2065.