# QALM

## A tool for automating Quantitative Analysis of LC-MS-MS/MS data

By
Kjartan Lerøy
01.06.2010


Master Thesis
Department of Bioinformatics
University of Bergen

# Preface

The goal of bioinformatics is to support science and research in the field of biology through the application of information technology. Proteomics is a field within biology that deals with the study of proteins.

This paper describes QALM, an application developed to automate and simplify a specific type of proteomics analysis. QALM is first and foremost a proof of concept through which certain options for implementing such automation have been explored. Although a functional and usable application has been created, this should primarily be considered a stepping stone for similar applications in the future.

Currently QALM is a desktop tool for importing and exporting data, integrating and communicating with external systems for the analysis of such data, and finally generating reports to present the results. It currently runs only under the Linux operating system, but it should be possible to change this fairly easily.

The hope for the future is that an expanded version of QALM may become a usefull tool for biologists and researchers in the field of proteomics. QALM has the potential to aid them in producing results faster while keeping track of large amounts of data in an effective way and simplifying statistical analyses of that data.

This thesis would not have been possible without the valuable help and guidance of Professor Ingvar Eidhammer, who has been the supervisor for the project. Thanks also to PhD candidate Marianne Brattås at the Department of Molecular Biology for providing usefull feedback and tips on the useage of R and XCMS.

The first two chapters describe briefly the biological and scientific foundations of proteomics and protein quantification in particular. Chapter 3 gives an overview of how a specific type of statistical analysis is currently carried out, and touches upon options for automating it. Chapter 4 discusses the main goals of the thesis and QALM, and lays the ground for the final solution and implementation details which follow in chapters 5 and 6. Finally, chapter 7 includes concluding remarks and some visions for the future.

# Contents

# Chapter 1

# Proteins and proteomics

Proteins encompass a vast and varied group of biological molecules and structures that play important roles in many chemical reactions, and are important for the structure of cells and tissues. Although they are a large group with many different functions, the basic structures of all proteins are the same: They consist of one or more chains of amino acids.

## 1.1 Amino acids and proteins

Amino acids are a group of fairly basic biological molecules with a comon structure that differs only in the section of each molecule known as the side chain (denoted by R, See Figure 1.1). There are 20 regularly occuring amino-acids that function as the building blocks that make up proteins. One might think of them as a set of rigid "links" in a chain. The various shapes, sizes and chemical properties of each such "link" affects the form and functions of the overall chain (the protein).

Amino acids can be joined together through a substituted amide linkage, known as a peptide bond. The part of an amino acid that remains after this linkage is refered to as a residue (hence the R in the figure). A chain consisting of two or more residues is called a peptide[1], while longer chains with a molecular weight over 10,000 are refered to as proteins, although there is no clear boundary, and the two terms may sometimes be used interchangably [NelsonCox08, p.82-85]. Although some proteins may contain chemical components that are not amino acids ("conjugated proteins", which most often contain a lipid or sugar moiety or a specific metal), amino acids are the main building blocks of all proteins, and the targets for the analyses described in the following chapters.

The ends of a polypeptide chain are often refered to as the N-terminal and the C-terminal, for the Amino-end (which has a Nitrogen atom) and the Carboxyl-end (with a Carbon atom) respectively. The N-terminal is conventionaly placed towards the left, and the C-terminal to the right, as in Figure 1.1.

[1] dipeptides, tripeptides, oligopeptides, and polypeptides consist of two, three, a few, or many amino acids, respectively.

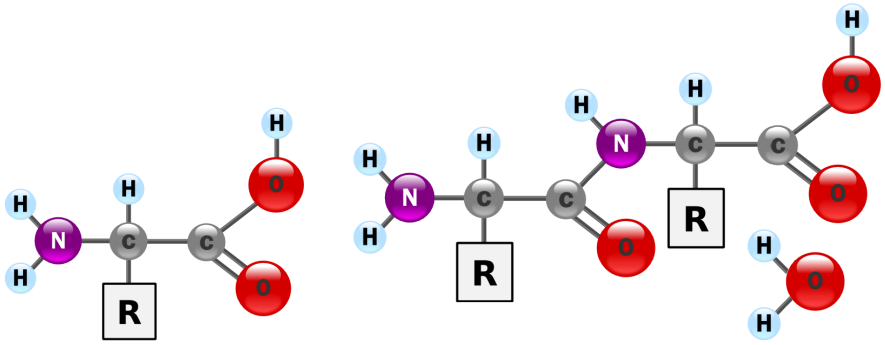



Figure 1.1: Two separate amino acids (left) are linked by an amide bond to create a dipeptide and water (right).

*Image: Public domain, courtesy of Wikipedia and YassineMrabet.*

*Source: http://en.wikipedia.org/wiki/File:AminoAcidball.svg (edited)*

### 1.1.1 Four levels of protein structures

The structure of a protein is generally divided into a conceptual hierarchy with four levels of complexity. The first, or **primary structure**, is the sequence order of all the individual amino acids in the chain. **secondary structure** is defined by certain reocuring structures, consisting of particularly stable arrangements of amino acids redidues.

A protein may consist of one or several peptide-chains or subunits. **Tertiary** structure refers to the tree-dimensional shape or folding of such a unit. If a protein consists of more than one subunit, then the arrangement of these are refered to as the protein's **quaternary structure**.

### 1.1.2 Protein variants and masses

Proteins have specific peptide sequences which when known, identify them uniquely. Nonetheless, the same protein may exist in a few different variants, and the differences between these may affect both the function of the protein and the process of identification.

#### Isotopes

The atoms in amino acids may have isotopes; nucleotides of the same chemical element, with the same atomic number, but differing molecular masses. Isotopes occur naturally, and occasionally cause two otherwise identical amino acids to have slightly different masses - an effect that is carried on to the proteins. A protein therefore, does not have a single defined mass, but rather a **mass distribution** that depends on the distribution of isotopes that may affect it.

#### Posttranslational modifications

Posttranslational modifications are changes in the chemical structure of a protein that are caused by the cellular machinery, but that occur after the protein

has been formed[2]. Such modifications play important roles, and may be neccessary for a protein to function correctly. They also affect properties used to identify proteins, including the mass. When calculating the theoretical mass for a protein, the possibility of posttranslational modifications must therefore be taken into account.

#### Calculation of the theoretical mass of a protein

The theoretical mass of a polypeptide is calculated by adding the theoretical mass of each amino acid residue in it's chain, and then adding the extra masses for the N- and C-terminus. If there are posttranslational modifications, then the effect of these must be added to the calculation subsequently.

## 1.2 Proteomics; Identification, characterisation and quantification

Proteomics is the study of functions and structures of proteins, and encompasses a wide variety of experiments and areas of research. Three typical objectives are:

- **Identification** of a protein, usually acheived by identifying it's amino acid sequence, or by measuring other properties that destinguish it.
- **Characterisation** of its biophysical and/or biochemical properties
- **Quantification** - determining the amount in a sample as either a relative or absolute value.

### 1.2.1 Protein databases

The tasks of proteomics require the use of protein databases. There are various databases for different types of protein-related information. Sequence-databases such as Swiss-Prot and TrEMBL[3] focus on the amino acid sequences of the proteins, while others are concerned with other properties such as possible posttranslational modifications or other chemical modifications[Eidhammer08, Chapter 1].

Several programs for sarching protein databases exist. This thesis uses and focuses on the search-engine Mascot, described in section 1.3.

### 1.2.2 Top-down and bottom-up approaches

There are several different ways to perform experiments and analyses in proteomics. Roughly speaking, the approaches may be divided into two groups: Top-down, in which the target protein is analyzed directly, and bottom-up, where it is first cleaved into shorter peptides, which are then analyzed individually.

[2]Posttranslational modifications may range from the relatively simple such as phosphorylation and dephosphorylation, to more complex reactions such as cleavage of the protein, for example to remove inhibitory sequences.

[3]These two databases are currently being combined under the name UniProt; See http://www.ebi.ac.uk/uniprot/

Of these two, the latter is most frequently used, due to several factors [Eidhammer08, Chapter 1]:

- The sensitivity of measurements is generally better for peptides than for whole proteins.
- The mass distribution due to isotopes is more complex for larger molecules and proteins than for short peptide-stretches.
- Posttranslational modifications have the same effect: Taking all possible modifications for a large protein into account may be substantially more complex than for individual peptides, where the possible modifications are more limited.
- In proteomics, as in all experiments involving physical measurements, errors and inaccuracies occur regularly. In mass spectrometry , such errors increase with increasing mass over charge (described in section 1.2.4).
- Some proteins are too large, or have other properties (eg. hydrophobicity[4]) that make it impossible to measure their mass.

### 1.2.3 Digestion and separation

Before any bottom-up proteomics experiment can be performed, the proteins to be analyzed must be cleaved into peptides with lengths appropriate for such experiments. This process is known as **digestion**, and is generally done by **proteases**, a class of enzymes[5] that cleave peptide chains at specific places. When the function of a protease is understood, this knowledge can be used to carry out the theoretical digestion of proteins in the database. Even so, irregularities may occur, possibly resulting in **missed cleavages**. This may make it difficult to identify the resulting peptides, as they might not match the theoretical peptides. A solution is to take the possibility of missed cleavages into account, for example by allowing for a small number of them when searching for matches (see section 1.3.1).

The most commonly used protease is trypsin, which is easy to obtain and purify, works in most experimental settings, and cleaves proteins reliably into peptides of suitable lengths (typically 6-20 peptides long and with few missed cleavages).

Usually, protein samples are so large and complex that digestion alone may not simplify them to a satisfactory degree. Often, it is desirable to divide samples into small enough parts or **fractions**, that there is only one single, or a very small number of proteins in each. Several procedures for attaining such **separation** exit. For separation on the protein-level (before digestion), the most commonly used method is 2D gel electrophoresis. For peptide-separation (when digestion has been performed before separation), the dominantly used method in proteomics is **high preassure liquid chromatography** (**HPLC**, or just **LC**).

The key to the function of LC is the characteristic **retention time** of individual components of a sample - the time it takes for the component to pass

[4] The repulsion of water (think of how oil acts when added to water).

[5] Enzymes are themselves, a class of proteins. They act as catalysts, helping to speed up or carry out speficic chemical reactions [NelsonCox08, p.183-184]

through some imobilized, porous substance[Eidhammer08, Chapter 4]: A peptide sample is injected into a "**mobile phase**" (a liquid) which then moves through a "**stationary phase**", generally a porous solid in a column. Thorughout this process, the different peptides interact with the stationary phase and the mobile phase to varying degrees, thereby moving at different speeds and separating from each other over time.

After LC, the various components turn up as bands in the column, or as peaks in the resulting **chromatogram** - a diagram displaying the distribution of the various components over time (see the top diagram in Figure 1.2). Since the molecules interact with the phases on an individual level, even molecules of the same type will tend to spread out a little. This causes a chromatogram to contain peaks around the average retention-time of each component, instead of a single clear column for each of them.

Depending on the type of experiment being carried out, separation may take place either before or after digestion. In the first case, the separated components are intact proteins; in the latter case, similar peptides from several proteins are grouped together prior to the main experimental procedure.

After experiments, the observed **experimental masses** of the resulting peptides can be compared to **theoretical masses** for identification. These are masses that have been calculated through the simulation of cleavages of candidate proteins in sequence databases.

### 1.2.4 The basics of mass spectrometry

Mass spectrometry is a procedure through which the **mass-to-charge** ratio, (often designated **m/z**) of the components in a chemical sample is measured. The physical measurement is done with an instrument known as a **mass spectrometer**. There are various types of mass spectrometers, but they are all based on the same underlying principle: An **ionization source** charges the sample by adding protons to it. This makes it possible for a **detector** to register the sample later, but also adds 1 Da of mass to the molecule for each proton that is added, a side effect that must be accounted for later, during calculation of the result.

After ionization, the sample is transfered to the **mass analyzer**, which separates the different components in it by mass-over-charge (m/z), after whichn the components are registered by the **detector**, and a **mass spectrum** is created.

One relatively simple instrument for carrying out mass spectrometry experiments is the MALDI TOF[6] [Eidhammer08, Chapter 8]. In the "MALDI" part of the instrument, short pulses of light are used to ionize small organic molecules (known as the "matrix") that have been mixed with the sample containing the peptides. The ionized organic molecules may then transfer protons to the peptides. The energy from the short bursts of light also causes the mixture to evaporate. After evaporation, the ions in the resulting gas is accelerated by an electric field, then sent through a "drift tube" in which there is no such field. The velocity an ion achieves depends on its mass and charge, and so is therefor the time required to reach the detector, which is located at the end of the

[6]MALDI and TOF stand for Matrix Assisted Laser Desorption Ionization, and Time Of Flight, respectively

drift tube. This allows for the calculation of m/z-values for the ions, and the generation of an MS-spectrum.

### Information in a mass spectrum

A mass spectrum is a diagram with m/z-values along it's horizontal axis and intensities along it's vertical axis. From this, the masses of components can be calculated if their **charge-states** are known (a registered component will usually have a charge state of 1, 2 or 3, by the addition of that number of protons).

**Example:**

> Assume there are two components, one with mass 420, and one with mass 360. With one charge, the first will be visible as a peak at the m/z-value:
> (420 + 1) / 1 = 421
>
> With two charges, the same component will be registered at m/z:
> (420 + 2) / 2 = 211
>
> In the same way, the second component may be registered with a m/z-value of either 361, or 181.
>
> □

Isotopes, described in section 1.1.2, also play a role here. The abundance of different isotopes for different chemical elements vary, with $^{13}C$ (heavy carbon) being by far the most commonly seen in peptieds, with a natural abundance of about 1 percent [Eidhammer09, Chapter 5]. This means that if a certain type of peptide contains 30 carbon atoms for instance, one can expect roughly one out of three such peptides to contain one $^{13}C$, while the rest of the carbon atoms will be the more common $^{12}C$. A small number of peptide molecules might contain two or more $^{13}C$. In a mass spectrum this might result in one peak for the "regular" peptides and another, smaller peak above it for the heavier peptides. If peptides with more than two $^{13}C$ occur, an even smaller peak might be visible above the first two.

A single such peak is refered to as a **monoisotopic peak**. A set of monoisotopic peaks (whose m/z-positions are in accordance with that expected due to isotopes) are refered to as **isotopic envelopes**. Knowledge about such patterns may be usefull for calculating the possible charges and masses of observed components[7].

The peaks in a monoisotopic envelope may be combined into a single peak later (so-called deisotoping) to simplify the calculation and identification of peptides. The term **peptide image** may be used to signify an isotopic envelope, where the peaks are believed to originate from a single peptide. An observed

[7]The distances between isotopic peaks depend on their charges: If z=1, distance will be (m/z)=1, if z=2, distance will be (m/z)= $\frac{1}{2}$, etc. Simultaneously, the m/z position will be changed according to the formula: (M + z*$H^+$)/z$H^+$, where z represents charge, M the original mass of the peptide, and H the additional mass due to added protons [Eidhammer08, p.71].

peptide therefore has at least one peptide image in each spectrum, but may have more than one if it occurs with different charges.

In larger molecules different isotopes may occur more regularly, and molecules containing only a single isotope will be more rare. Instruments may not always be able to clearly identify monoisotopic peaks, so for large peptides or proteins the average mass is often used instead of the monoisotopic mass.

#### Raw data and peak-lists

Mass spectrums can have one of two formats: That of the **raw data**, in which the peaks are displayed as they were observed in the MS scan, or that of a **peak-list**, in which the data has been processed.

A peak in a raw data spectrum will be stretched over a short range of m/z-values and contain numerous measurements along small increments in this range. In a peak-list, these intensities will have been combined into a single line or column, so the total intensities of the peak are specified at a single point.

### 1.2.5 MS/MS analysis

#### PMF versus Tandem Spectrometry

Two main approaches are used in bottom-up proteomics. **Peptide Mass Fingerprinting** (PMF), also refered to as mass profile fingerprinting or peptide maps, is the process of comparing *experimental* peptide masses (from experiments) to *theoretical* peptide masses (information from a protein database) [Eidhammer08, Chapter 1]. If an observed protein has a corresponding protein in the database, then for each theoretical peptide mass from the database protein, a corresponding peptide mass should be observed in the experiment. Full coverage of the protein sequence is not achieved in practice however; the norm is a coverage of around 20-40%. In stead, search tools do statistical calculations to give indications of the confidence of a proposed identification.

A weakness of the PMF approach is that it will usually only determine a single property of peptides, namely their mass. Since there may be a great number of peptides with identical or very similar masses, determining the exact amino acid contents of an observed peptide is difficult, and establishing the positions and sequene of the residues may be impossible.

The alternate approach, **Tandem Spectrometry** or **MS/MS**, is distinguished by the fact that it involves carrying out spectrometry on two (or possibly more[8]) "levels" [Eidhammer08, Chapter 1]. On the first level, an MS scan is used to select ionized peptide molecules from a certain m/z interval for further analysis. The selected molecules (refered to as the **precursors**) may be fragmented before the next level of MS scans (denoted **MS/MS**), in which the m/z values of the resulting fragments are finally measured, yielding data from which the peptide sequence may be derived.

Often the interval of selection will be small enough that only a single precursor will be selected for MS/MS at a time. Each MS scan is followed by a number of MS/MS scans, usually between 0 and 5. Typically, the most intense

[8]Repeated MS-scans are possible, generating "$MS^n$"-analyses. Procedures employing two levels of scans are most common however, and are the focus of this thesis.

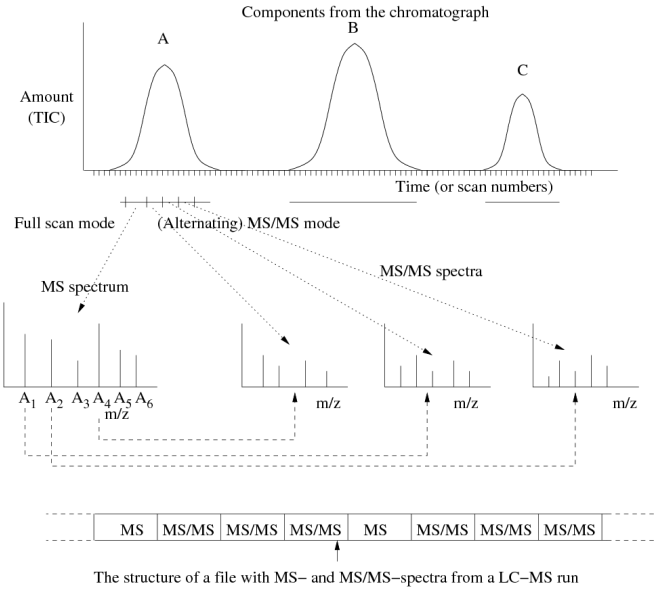



Figure 1.2: Illustration of the process used for recording LC-MS/MS spectra. The three highest peaks from the MS spectrum are selected for MS/MS.
*Image: Original from [Eidhammer08, Chapter 8]; this is an expanded version acquired directly from the author.*

peaks from the MS scan are selected for MS/MS. This process is illustrated in Figure 1.2.

Before either a PMF or an MS/MS analysis can be performed, the proteins must be separated and digested. In the case of PMF, the separation into fractions usually occurs before digestion. Ideally, one would like to end up with groups of intact proteins, with all the molecules of a single protein in one and the same fraction, but this is hard to achieve. If each fraction only contains a small number of proteins however, they are still managable and can be used for further analysis.

In MS/MS, the proteins are only separated after digestion, which results in large amounts of peptides and greater complexity. It is possible to do a separation step before the digestion, then do a second separation step on the peptides afterwards. Such a precedure would remove a lot of complexity, but would at the same time require a great deal extra work, as each step would have to be repeated for each fraction from the preceeding steps. Such large amounts of work would slow down a large analysis process and make it impractical.

### Selecting peaks for MS/MS

The peaks from an MS spectrum for which MS/MS is to be performed are usually selected automatically by a MALDI instrument, but criterias for this selection can be set by the user. These may for instance include:

- The number of peaks to include (usually 3-8)
- The charge state (This is usually determined by the spacing between the

isotope peaks)

- A list of specific m/z values to analyze
- A list of specific m/z values to ignore, for instance due to known contaminants that are likely to pe present here.

## 1.3 Protein identification with Mascot

Mascot is a search engine that uses data from mass spectrometry to identify proteins by searching sequence databases. It uses several different search methods to acheive this; in the context of this thesis, the most important of these is MS/MS Ion search, in which data from MS/MS spectra are the input data. The goal for Mascot is to find an exact match if possible, or as close a match as possible. Citing the Mascot website[9]:

> *If the "unknown" protein is present in the sequence database, then the aim is to pull out that precise entry. If the sequence database does not contain the unknown protein, then the aim is to pull out those entries which exhibit the closest homology, often equivalent proteins from related species.*

### 1.3.1 Carrying out an MS/MS Ion search

Mascot can accept several different peak list formats. A list of such formats is available from the Mascot website. In this thesis and in QALM, the format used is the XML-based *mzData*, version 1.05, which will be described in chapter 6.6.1.

Several settings for a Mascot search may be specified as needed, depending on the experiment and the peak list to be used as search input. Among other things, a user may[10]:

- select which databases to search
- specify which enzyme should be used to perform the theoretical digestion of the proteins (trypsin, mentioned in section 1.2.3, is typically the default).
- specify how many missed cleavages to take into account
- select a taxonomy which limits the search, for example to proteins from a specific species or group of species, thereby speeding up the search and filtering out results known to be irrelevant.
- set various tolerance-values for the search

[9] http://www.matrixscience.com/search_intro.html

[10] For a web-based example of searching with Mascot, see:
http://www.matrixscience.com/cgi/search_form.pl?FORMVER=2&SEARCH=MIS

### 1.3.2 Results of a search: Mascot's raw data format

Results from Mascot searches are available in various formats, some of which depend on what type of search is being done. A part of QALM has been created specifically for mining the raw data format (or **.dat files**) resulting from a mascot search. These are plain text files divided into several sections that together contain all the result data from a search. The format of such files will be described in chapter 6.6.2.

# Chapter 2

# Protein Quantification

Protein quantification involves studying the quantities of proteins (also known as *protein profiles*) in some specified situation and under specific circumstances. Such knowledge is essential in a number of scientific fields and may be incorporated into different types of projects with various goals. Most of the theory in this chapter is taken from *Protein Quantication by Mass Spectrometry - A Compendium for the Course INF389* by Ingvar Eidhammer [Eidhammer09].

### Specifying sites and circumstances

Some examples of situations where protein quantification may be applied are:

- when investigating how the changes in protein profiles depend on changing states; for example how they change in response to various drugs or when exposed to certain contaminants.
- when examining how protein profiles differ between corresponding sites in individuals under different circumstances; for instance comparing results from a sick individual to those from a healthy individual.
- when investigating how protein profiles from corresponding sites in different species differ.
- when examining how protein profiles in a specific site varies over time.

It is important to specify precisely the sites and circumstances under which the protein quantities are measured; for an investigation of protein quantities to be meaningful and valid, the compared samples must be comparable in some specified way. In addition, it is important to ensure that factors that may affect the result are as similar as possible. Such factors may be:

- Biological, such as age, sex and weight
- Physiological, such as states of hunger or stress, etc.
- Environmental factors surounding the individuals, such as temperature and humidity

- Other things, depending on the experiment and what type of sample is being taken.

,

## 2.1 Relative and absolute quantification

**Relative quantification** involves comparing the abundance of proteins from two or more different samples and determining the ratios between them. This approach is primarily used to check for **differentially expressed proteins**; instances in which the abundances of one or more proteins in one or more of the samples differ from the corresponding abundances in the rest of the samples by a statistically significant degree. XCMS, the program that lies at the core of the statistical component in QALM, carries out statistical calculations for pairwise comparisons of such situations (XCMS is described in the next chapter, see section 3.2).

**Absolute quantification** involves measuring the amount of a specific protein in a mixture and specifying it in absolute terms (typically in moles[1]). Because measuring such quantities can be significantly more complicated than relative quantification, relative quantification is used more frequently than absolute quantification [Eidhammer09, Chap.1].

## 2.2 Label-based and label-free quantification

There are two main procedures to choose from when carrying out quantification projects. In **label-based quantification** the proteins or peptides in one situation are labeled, or proteins or peptides in both situations are labeled but with different labels. The samples from both situations are mixed prior to the MS-procedure, and the labels should cause the two versions of the peptides appear as two peaks in the same MS spectra, with a distance between them that corresponds to the weight of the label (or the difference in weight between the two labels if both situations are labeled)[2].

Stable isotopes are often used as labels, as this lets both the unlabeled peptide and it's labeled conterpart retain *the same chemical properties*, but have *differing masses*. The first point here is noteworthy, as it is important for the labels not to affect steps in the process such as the ionization or the chromatography. Furthermore, heavy isotopes are simple; only a single label is added to a peptide, and they do not complicate the MS/MS spectra significantly.

Ideally, the observed amount of peptide ions should mirror the relative peptide amount in the combined sample (both situations). After performing the MS, the spectrums are therefore searched for potential peak-pairs that may represent two versions of the same peptides (heavy and light). MS/MS is thereafter performed to confirm that the pairs do originate from the same peptides,

[1] A mole of a substance contains aproximately $6.022 * 10^{23}$ molecules of the substance; this is defined in such a way that the mass in grams of a mole of a substance will be equal to the substance's molecular mass in atomic mass units.

[2] This assumes the charge for each peptide is one. If there is more than one charge, then the distances will be reduced by a factor of 1/z, as explained in a footnote under section 1.2.4

and the relative abundances between them is found by creating extracted ion chromatograms (XIC) (see section 2.3.2 for details).

In **label-free quantification**, the data for two samples are created in two separate MS runs. A quantification project of this type may be roughly divided into five steps:

1. Perform an LC-MS run (or an LC-MS-MS/MS run) for each sample.
2. Search for instances of **common peptides** - peptides that occur accross several or all samples.
3. Estimate the abundance of each of the common peptides in each of the samples.
4. Using MS/MS data, identify the protein origins of these common peptides.
5. Calculate the abundance of each protein accross the samples.

The first result of this process should be a table showing the observed abundances of each peptide in each of the samples. This is then used to create a similar table describing the abundances of each identified protein in each of the samples. Note that this data must be normalized for the values from different samples to be comparable (see section 2.2.1).

Varying peptide properties cause different peptides to be ionized to different degrees. This means that the observed abundances (peaks) of different peptides may not be comparable, so relative abundance can not be determined for different peptides. Observations of the same peptides in different (but comparable[3]) samples are comparable however, and can be used to determine the relative abundance of the same peptide in different samples.

### 2.2.1 Normalization

In a broad sense, normalization involves removal of or accounting for possible errors in measurements or sets of data. This is a broad field in itself which will only be mentioned briefly here.

In LC-MS projects it is neccessary to compare results from various experiments carried out on different samples. As in all physical measurements, results may vary slighly from MS-run to MS-run, even for samples that are identical. Contaminants in the samples or instrument, slight variations in the way samples are handled and other things may affect the observed abundances and effect variations in the results that should be corrected for through normalization. Normalization may be performed in several ways, among others:

- If it is assumed that the total abundance in each sample is equal and that shifts are relatively equally distributed among them, abundances may be transformed to: $p'_i = \frac{Tot_Q}{Tot_P} * p_i$, where $p'_i$ is the result of normalizing the observed abundance of $p_i$, and $Tot_Q$ and $Tot_P$ are the total abundance in the respective samples Q and P.
- By doing linear regression normalization if suspecting a systematic shift that increses with the magnitude of the abundances.

[3]In this context, "Comparable samples" are samples that can be assumed to contain approximately the same peptides.

## 2.3 Quantification projects

In broad terms, all quantification projects consist of four major steps:

1. Define the problem and plan the project (What situations are there and what type of statistical procedures are to be performed? How many samples will there be, how should they be grouped, and how will the protein profiles be created?)
2. Carry out the main experimental procedures.
3. Calculate the abundances of the proteins observed in each sample.
4. Perform statistical analyses on the resulting data.

### 2.3.1 The central experiments: From peptide abundances to protein abundances

Calculating the abundance of a protein is done by first calculating the abundances of peptides that originate from the specefied protein. The process of selecting peptides and finding their abundance is somewhat complex. Figure 2.1 gives a schematic overview of some of the steps involved[Eidhammer09, Chap.1]:

1. First the protein sample is digested.
2. If neccessary the resulting peptide sample may be divided into smaller fractions more suitable for the following LC-MS procedures. Instances of the same peptide may occur in several fractions.
3. For each fraction, an LC run is performed. Individual instances of the same peptide may sometimes occur in different LC peaks (i.e.have differing retention times).
4. For each LC peak, there will generally be several MS runs (3 are illustrated in Figure 2.1). This means that a peptide in a single LC peak may be observed in several MS spectrums.
5. Because of the possibility of varying charges among the different instances of a peptide ion, the same peptide may appear at several places in a single spectrum. The MS spectras in Figure 2.1 have been **deisotoped**; the various isotopic peaks have been added together, so each isotopic envelope is only visible as a single peak in the diagrams.

The observed abundance of a single peptide may vary considerably due to differences in ionization, experimental variations, and errors. The calculation of the abundance of a protein is therefore based on the estimated abundances of several peptides.

### 2.3.2 Comparing two situations

A common type of experiment is the comparison of protein profiles from two different situations, for instance when investigating the effects of some contaminant. A typical procedure for such a project might include:

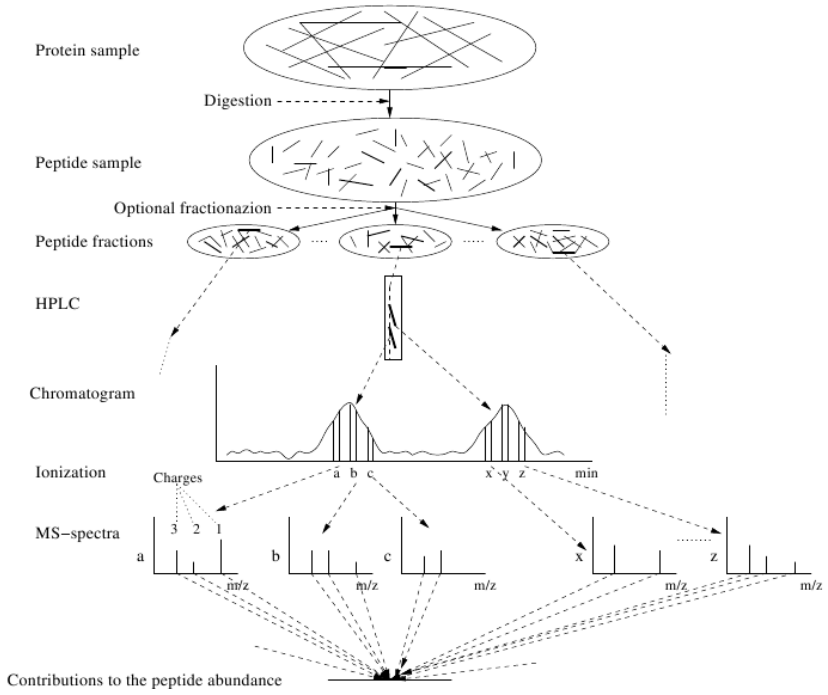



Figure 2.1: Calculating the abundance of a single peptide
*Image: From [Eidhammer09, Chapter 6].*

1. Defining the situations and groups. If testing for effects of contaminants, it might be natural to define two groups of samples; one **controll group** with samples from "healthy" individuals, and one **treatment group** with samples from "contaminated" individuals.

2. Determine number of **replicates** needed to counteract biological or experimental variations. These may be:

   - **Biological replicates**: Similar samples taken from different individuals in the same group to account for biological diversity.
   - **Technical replicates**: Several similar samples from the same individual to account for variations in the experimental procedure.

   The total number of replicates, also known as the **group size**, depends on the statistical analyses that are to be performed and the desired accuracy of the results.

3. Select the comparable individuals for both groups, perform the experiments and obtain the protein samples.

4. Perform the LC-MS experiments for each sample

5. Find the relative abundances of proteins in each sample

6. Apply statistics to the observed intensities to analyze the effects.

### Calculating protein abundances from peptide abundances

When comparing two samples $S_1$ and $S_2$, the relative abundance of some protein in the two samples will be $\mathbb{R} = \mathbb{P}_1/\mathbb{P}_2$. The corresponding relative abundances for each peptide should then in theory be $\mathbf{p_1^{(i)}}/\mathbf{p_2^{(i)}} = \mathbf{r^{(i)}} = \mathbb{R}$, where $\mathbf{p_1^{(i)}}$ and $\mathbf{p_2^{(i)}}$ are corresponding abundances of peptides from protein 1 and 2 respectively. Due to variations and uncertainties in the experiments the observed relative abundance for a single peptide is seldom correct, so the relative abundances of several peptides are incorporated. The result can be calculated in two ways:

- By estimating $\mathbb{P}_1$ from the abundances of $\mathbf{p_1^{(i)}}$ for each peptide $i$, and $\mathbb{P}_2$ from the abundances of $\mathbf{p_2^{(i)}}$ for each $i$, and finally calculating $\mathbb{R}$ from these.
- By estimating $\mathbf{r^{(i)}}$ using $\mathbf{p_1^{(i)}}/\mathbf{p_2^{(i)}}$ for each common peptide $i$, then using the resulting values to calculate $\mathbb{R}$.

The second approach is the most commonly used, since calculating the relative abundance of each peptide first may account for some of the variations in the experiment.

This requires that each peptide in the calculation has been confirmed to originate from the target protein, and that it has been quantified as correctly as possible in each of the two situations. This may be somewhat involved, since the same peptide may appear in different spectrums even when the protein abundances are aproximately the same. They may also appear several times in the same spectrum due to a varying number of charges. An **extracted ion chromatogram** (**XIC**) is therefore created; this is a diagram designed to show the overall intensities of only a very specific mass (see Figure 2.2).

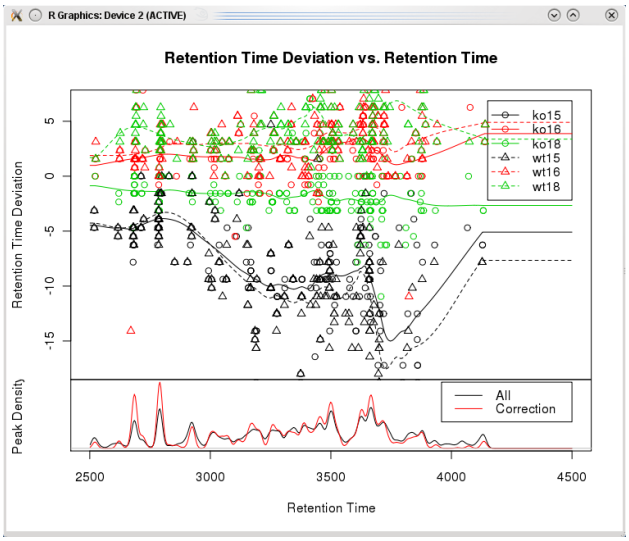



Figure 2.2: Example of an extracted ion chromatogram (XIC) showing datapoints for regression, resulting deviation profiles, and the distribution of peak groups accross retention time.

*Image: Generated by XCMS from sample data.*

## 2.4 Label-free quantification by ion current

The goal of label-free quantification is to create a table describing the quantities of various proteins in each of the samples. In an ideal case, each peptide would have the same retention time in each sample, and occur only in a short time interval, causing it to be observed in exactly two MS-spectra (one for each sample), and then with the same m/z-value. Such ideal situations do not occur in practice, however. Noise, variation in preparations of the samples and the instruments complicate the situation so that the following must be taken into account:

- Retention times for the same peptide may vary between samples.
- Varying charges may give the same peptide different peptide images.
- The same peptide may occur in different chromatographic peaks.

Taking these factors into account, the following steps are neccessary when analyzing the spectra:

1. Common peptides must be found (generally done by finding common peptide images).
2. The abundances of the common peptides in each sample must be determined.
3. The proteins from which the peptides originate must be identified (normally done by performing a database search using MS/MS data as input, as will be described in section 1.3).
4. The abundences of proteins must be calculated from the abundances of common peptides.

There are two possible procedures for step 3 and four: The first, which is the type of procedure QALM has been developed for, involves doing LC-MS-MS/MS from the start. The second procedure consists of doing an LC-MS run first and identifying peptides where the observed abundances differ between samples. The protein origins of these peptides are then determined through a new MS/MS run.

### Aligning chromatograms

As peaks for the same peptide may have differing retention times in different chromatograms, they have to be aligned to each other so that the correct peaks "overlap" as closely as possible before they can be compared (see Figure 2.3).

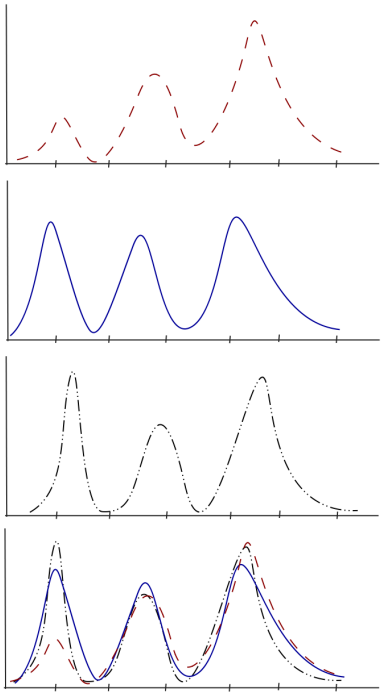


Figure 2.3: Simplified view of aligning peaks in different chromatograms.

# Chapter 3

# R and XCMS

The statistical component of QALM is based on the **R platform** and the package **XCMS**, which enables analysis of LC-MS-MS/MS data in R. This chapter describes R in general and more specifically the requirements for loading XCMS and carrying out certain analysis tasks on LC/MS data. Instructions for installing R and XCMS are included in apendix A.

## 3.1 The R-Project

The R Project is an open-source platform of software for statistics. It includes both a programing language and an environment for statistical computing. From the webpage[1]:

> *It is a GNU[2] project which is similar to the S language and environment (...). R can be considered as a different implementation of S. There are some important differences, but much code written for S runs unaltered under R.*
> *R provides a wide variety of statistical (...) and graphical techniques, and is highly extensible. The S language is often the vehicle of choice for research in statistical methodology, and R provides an Open Source route to participation in that activity.*

### 3.1.1 The R Environment

R contains software facilities for handling and storing data, and has a large collection of tools for performing data analysis, including graphical facilities for display.

R is not limited to statistics, but is an environment in which many classical and modern statistical methods have been implemented[RIntro]. In addition to the standard and recomended packages that are shipped with R, many more packages designed for various tasks are available through **CRAN**, the **C**omprehensive **R** **A**rchive **N**etwork[3]. XCMS and it's dependencies are examples of such packages, and will be described in section 3.2.

[1] http://www.r-project.org
[2] http://www.gnu.org
[3] http://cran.r-project.org

R differs from other platforms for statistical computing in one particular respect; instead of writing output to the screen the way many other systems do, R focuses on using and storing data in objects. The section on XCMS will include examples of this.

R runs on various systems such as Windows, Mac OS X, Linux and FreeBSD, and is available as free software under the terms of the GNU General Public License, version 2[4].

### 3.1.2 The R programming language

The R programming language may be considered an implementation, or a *dialect* of the S programming language. S was developed in the 1980's and has seen widespread use in statistical computing. It is a functional language, and though the syntax may resemble that from C and similar languages, it's semantics are more like those of Lisp, including the possibility of *"computing on the language, which in turn makes it possible to write functions that take expressions as input, something that is often useful for statistical modeling and graphics"*[RLangDef].

The R language has support for Objects of various basic types as well as special compound objects, and it is possible to do a variant of object oriented programming in R. QALM does not define objects of it's own, but makes use of objects defined in XCMS.

#### Running commands in R

It is possible run R commands directly through a command line interface once the R environment has been loaded. In many cases, such use alone may be sufficient, depending on the work being done. It is also possible to create functions to automate common tasks, or to group commands in text-files, and load and call these either from the command line of the Operating System, or from within the R environment.

One of the major tasks QALM performs is the grouping and execution of a number of such commands in two separate steps, thereby making access to the command line interface and any knowledge of R redundant for the typical user. This will be elaborated on in section 3.3 and in later chapters.

## 3.2 XCMS

XCMS is an R-package specifically developed for the analysis of data resulting from LC/MS experiments. It was developed at the Scripps Center for Mass Spectrometry, a section of the Scripps Research Institute[5]. From the XCMS website[6]:

> *A common goal of all of the metabolomics/proteomics bioinformatic platforms is to allow users to identify and statistically assess metabolite and peptide features that show significant change between sample*

[4] see: http://www.gnu.org/licenses/gpl-2.0.html

[5] The Scripps Research Institute is a private non-profit research organization, funded mainly by the National Institute of Health (NIH) and other federal agencies in the US. For more information, see http://www.scripps.edu/intro/overview.html

[6] http://masspec.scripps.edu/xcms/xcms.php

> *groups. (...) The success of XCMS has come from largely achieving the primary goal in identifying and statistically assessing feature metabolite and peptides.*

XCMS can read and process LC/MS data stored in several formats, including mzData, which will be described in chapter 6.6.1. It has many options for interacting with and visualizing such data, as well as for carrying out retention time alignment, relative qunantitation, and more. The details relevant to this thesis will be described in the following section.

## 3.3 Use of XCMS under QALM

Under QALM, XCMS is used to preprocess the data imported into a project (QALM-projects will be described in section 6.1) and search for peaks that may represent differentially abundant peptides. The result of these operations should be a peak-list ordered by statistical significance so the most clearly differentiated peaks are found at the top.

The following description is based on the manual *LC/MS Preprocessing and Analysis with xcms*[Smith09], which was also central during the development of QALM. Only functions directly relevant to the problems in the thesis are described. The main steps needed are in short:

1. Organizing the directories and data files.
2. Starting R and loading XCMS.
3. Specifying the data files, importing them, and doing filtration and peak identification (generating **xcmsSet**-objects).
4. Performing the first matching of peaks across samples (grouping).
5. Doing the first retention time correction or alignment, which is followed by a new grouping procedure.
6. Repeating the previous two steps a number of times (zero or more).
7. Filling in data for missing peaks.
8. Generating a report of the most significant differences.

### 3.3.1 Directories and files

QALM works with mzData-files. These contain LC/MS data in an XML format (described in detail in chapter 6.6.1). Files in this format should be placed in directories where they will remain available to XCMS. Files containing data from samples from the same situation should be in the same directory, while files from another situations should be in separate directories, such as

```
current-project/situation-a/
current-project/situation-b/
```

Within XCMS, each situation will later be referenced by the name of the directory it's files was located in.

It is also possible to divide groups of samples further by employing hierarchies of directories. If samples should be divided depending on the day they were taken for instance, one might use:

```
current-project/situation-a/day-1/
current-project/situation-a/day-2/
current-project/situation-b/day-1/
current-project/situation-b/day-2/
```

XCMS would then automatically classify the files in each such directory separately. Support for more than one level of samples is not yet supported in QALM however, and will therefore not be elaborated on here.

Once placed in directories, the files should remain there as XCMS will refer back to them throughout the analysis process.

### 3.3.2 Starting R and importing XCMS

On a Linux system, R can generally be started with the comand `R` at a terminal or command-line[7]. Once R has started, a copyright notice and introduction message should appear, and below this, the entry point for commands to R. To call XCMS, the package must be loaded into R. This can be accomplished with the command:

```
>library(xcms)
```

When an R session is started, data from that session will be stored in a hidden file called **.rsession** in the directory R was started from. When the command **q()** (quit; exit from R) is issued at the R command line, a prompt asking whether or not to save this data (the **workspace image**) will appear. The next time R is started from the same directory, the workspace will be reloaded automatically. This makes it easy to continue working on the same data in a later session.

In QALM, R is always started from the directory the currently open project is located in. This makes it possible to continue working with previously generated objects and enables "manual" inspections of the generated objects and data by starting R from the command line in the project folder if needed. The contens of an R workspace image can be listed using the command:

```
ls()
```

### 3.3.3 Importing files to XCMS

There are several ways to let XCMS know about the files it should read data from. Among other things, given one directory (the root of the previously defined hiearchy), XCMS can search through it and any sub-directories recursively, and read any encountered files having supported formats.

In QALM, the situation directories (directories containing sample files to be included) are added to a vector[8], then that vector is passed as an argument to the

[7] For information about how to obtain and install R and XCMS, see Apendix A.

[8] A vector can be thought of as a type of list in R.

function **xcmsSet()**. The result of this call is stored in a variable (`my.xcms.set` in the following example):

```
# NOTE: Lines beginning with # are comments.
# Store list of two directories with files in a vector:
my.collection <- c('/home/projects/current-project/situation-a',
                   '/home/projects/current-project/situation-b')

# Pass that vector to the function xcmsSet, and store
# the result in the variable "my.xcms.set":
my.xcms.set <- xcmsSet(my.collection)
```

*Note: This is a simplification of the code actualy used in QALM, for the sake of clarity, but the result of the code is essentially the same, if a little less flexible. The actual calls made by QALM will be described in chapter six.*

The last command will initiate the process of peak identification. During this process, XCMS will output pairs of numbers showing the m/z it is currently working on, and the total number of peaks identified so far (see Figure 3.1). Each sample will have its output written on a separate line. Note that the final number of peaks might be lower than those displayed throughout the process, as postprocessing will remove duplicate peaks.

The resulting variable, `my.xcms.set`, will be an object of the type **xcmsSet**. It contains the lists of peaks from the samples and related information, and provides methods for aligning and grouping those peaks, as discussed in chapter 2.4. A summary of the data can be output by entering the variable name by itself (see also Figure 3.1):

```
# Display output about the imported samples and peaks
>my.xcms.set
```

The method xcmsSet() accepts several arguments that optimize its function for particular instruments or groups of samples. In most cases however, the default values should work acceptably. In the case of QALM, support for alternative arguments were not deemed neccessary[9], and have therefore not been integrated into the main application. Changes may be made to the individual R-scripts however, as explained in chapter 6.7.3. Further details on **xcmsSet()** can be found in the various manuals that come with XCMS.

### 3.3.4 Groups: Matching peaks accross samples

Once an xcmsSet has been created, the peaks in it that represent the same analyte need to be **grouped**. This is done using the **group()** method:

```
# Group peaks in different samples representing the same analyte:
>my.xcms.set <- group(my.xcms.set)
```

[9]This decision was made based on discussions with a biologist with some previous experience with XCMS, and who would be a potential user of QALM.

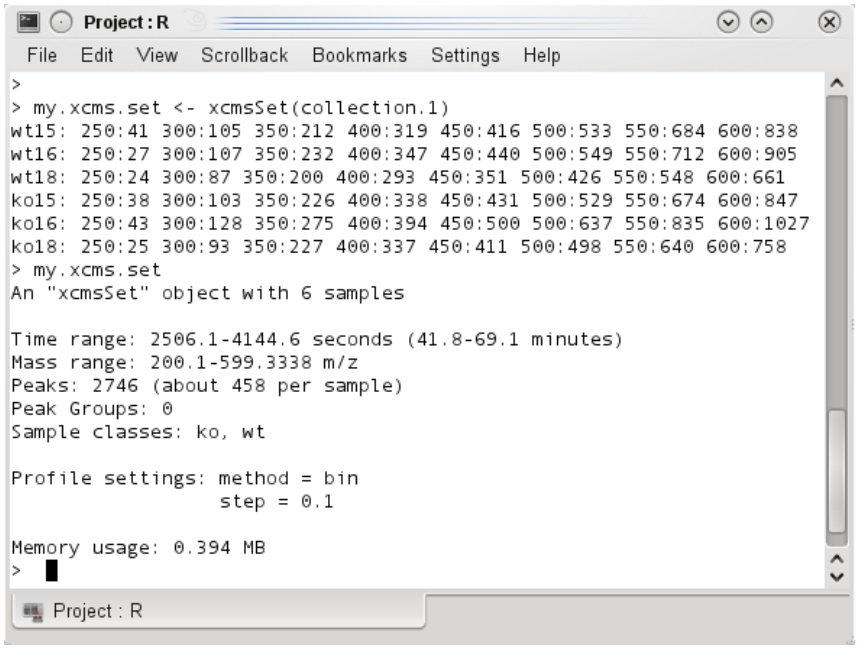



Figure 3.1: Output from creating an xcmsSet and displaying it's content ("collection.1" is a vector containing references to two directories containing in all 6 samples).

Note that this command stores the result of grouping **my.xcms.set** over the existing variable. This is acceptable here, as **group()** adds group information without destroying or changing the existing data (it is non-destructive).

As with **xcmsSet()**, arguments to **group()** are possible, but not supported yet in QALM.

### 3.3.5 Aligning chromatogram peaks

Peaks in different chromatograms representing the same peptides may have differing retention times. It is therefor neccessary to do **retention time correction**. XCMS attempts to align corresponding peaks in different chromatograms by using the group-information from the previous step. After correcting for drifts in retention time the group data is no longer valid, so a new grouping-step is needed.

Occasionaly, improper grouping will occur, resulting in groups that contain more than one peak from each sample, or groups that are missing peaks from some samples. Such data should not be used for retention time correction, so XCMS ignores groups with peaks missing from more than one sample or that have more than one extra peak.

The retention time correction procedure will calculate a median retention time for each group and a deviation from that median for each sample. The changes within a sample are aproximated using a local polynomial regression technique, and least squares regression is used for the curve fitting on all data

points.

Setting the argument **family** to **symetric** allows for outlier detection and removal. **Plottype** can be set to **mdevden** in order to display a plot of the retention times and what the algorithm is doing (neither are currently used in QALM). The retention time correction command:

```
# Plain / default retention time correction as used in QALM:
my.xcms.set.2 <- retcor(my.xcms.set)

# Alternative, with arguments as explained:
my.xcms.set.2 <- retcor(my.xcms.set, family = "symmetric", plottype =
"mdevden")

# Do a new grouping based on the new retention times.
my.xcms.set.2 <- group(my.xcms.set.2)
```

Note that this stores the result in a new variable. This is recomended, since the function **retcor** will change the retention times for the peaks in the object.

### 3.3.6 Extra iterations

The second grouping-run may result in more accurate data than the first. It may therefore be usefull to perform several rounds of retention time correction followed by new groupings. This can be repeated iteratively in the same manner as described in the two preceeding sections.

When doing so, it is advisable to use a new variable for the result of each iteration. In QALM this is done by using naming conventions for the variables that include the iteration-number, among other things.

### 3.3.7 Filling in missing peaks

Since peaks for certain peptides may be missing in some chromatograms even when observed in others, XCMS may sometimes produce groups with peaks missing from some samples even after retention time correction and regrouping. Using the **fillPeaks()** method, XCMS can re-read the original data files and integrate them, filling in the missing data points:

```
# Fill in missing peak data:
>my.xcms.set.3 <- fillPeaks(my.xcms.set.2)
```

### 3.3.8 Analyses: Generating peak reports

In the final step a report showing the most significant differences in intensity can be generated using the function **diffreport()**. When called from QALM, this step will be refered to as the *analysis*-step.

The arguments to this function, as it is used by QALM are:

- The xcmsObject to analyze data from
- The name of the first situation in the comparison (the name of the directory containing the files for the situation).

- The name of the second situation.
- The path and filename the report should be stored under. ".tsv" will be appended to this filename.
- The number of extracted ion chromatograms to produce[10] (zero in the following example).

```
# Generate peak report, to be stored as:
# /home/projects/current-project/reports/my_report.tsv
>my.report <- diffreport(my.xcms.set.3,
                 situation-a,
                 situation-b,
                 "/home/projects/current-project/reports/my_report,
                 0)
```

This function will calculate Welch's two-sample T-statistic for each analyte. The analytes will then be ranked by P-value and a report will be written to a tab-separated file which may be opened in Open Office.org Calc, Microsoft ExCell, or a similar application [XCMSAPI].

The data in the report is also available in the form of a **dataframe**; a set of tabular data in R. After running the previous example, the resulting data may be accessed as follows:

```
# Will write out all the result data:
>my.report

# Will write out the first three lines of the report:
my.report[1:3, ]
```

[10] Technicaly, this is included in the R scripts for QALM, but it is not used/supported in the main application.

## 3.4 Understanding the peak-report

Reports consist of the following columns (shown in Figures 3.2 and 3.3):

- **name**: A simple name to refer to the peak/analyte. It consists of the integer values of the analyte mass and retention time, prepended with M and T, respectively (for instance **M449T3288**).
- **fold**: The mean fold change between the situations. Refer to the next column (tstat) to see which was higher.
- **tstat**: Welch's two sample T-statistic. This number will be positive if greater intensity for the analyte was observed in the second situation, and negative if greater intensity was observed in the first.
- **pvalue**: The P-value for the t-statistic.
- **mzmed**: The median of the m/z values of peaks in the group.
- **mzmin**: The minimum m/z value of peaks in the group.
- **mzmax**: The maximum m/z value of peaks in the group.
- **rtmed**: The median retention time of peaks in the group.
- **rtmin**: The minimum retention time of peaks in the group.
- **rtmax**: The maximum retention time of peaks in the group.
- **npeaks**: The number of peaks assigned to the group.
- <**situation-a**>: The number of samples from the first situation represented in the group.
- <**situation-b**>: The number of samples from the second situation represented in the group.
- <**file-n**>: The integrated intensity value for each sample. There should be one column for each file / sample.

*Note: If there were several levels of situations (several levels of directories with data, as mentioned in section 3.3.1), then the p-value for anova statistics would follow after* **pvalue**, *under the column titled* **anova**. *This is not included in the list because multiple levels are not currently supported in QALM.*

| | A | B | C | D | E | F | G | H | I | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | name | fold | tstat | pvalue | mzmed | mzmin | mzmax | rtmed | rtmi |
| 2 | 1 | M301T3388 | 9.21 | -20 | 0 | 301.19 | 301.19 | 301.19 | 3388.45 | 3 |
| 3 | 2 | M300T3389 | 9.96 | -18.85 | 0 | 300.19 | 300.18 | 300.2 | 3388.92 | 3 |
| 4 | 3 | M423T3253 | 8.03 | -15.12 | 0 | 423.14 | 423.11 | 423.15 | 3252.57 | 3 |
| 5 | 4 | M548T3182 | 1.15 | -8.42 | 0 | 548.1 | 548.1 | 548.1 | 3182.11 | 3 |
| 6 | 5 | M348T3290 | 8.72 | -14.94 | 0 | 348.16 | 348.12 | 348.17 | 3289.85 | 3 |
| 7 | 6 | M298T3185 | 4.97 | -12.01 | 0 | 298.13 | 298.11 | 298.15 | 3185.14 | 3 |
| 8 | 7 | M491T3377 | 19.63 | -12.36 | 0 | 491.2 | 491.19 | 491.2 | 3377.49 | 3 |

Figure 3.2: An example of a portion of a peak-report (continued).

| I | J | K | L | M | N | O | P | |
|---|---|---|---|---|---|---|---|---|
| ed | rtmin | rtmax | npeaks | test_one | test_two | ko15 | ko16 | ko18 |
| 388.45 | 3386.89 | 3392.52 | 3 | 3 | 0 | 962353.43 | 1047934.14 | 11 |
| 388.92 | 3386.29 | 3392.52 | 6 | 3 | 3 | 4534353.62 | 4980914.48 | 52 |
| 252.57 | 3252.31 | 3253.78 | 3 | 3 | 0 | 236249.55 | 255168.62 | 2 |
| 182.11 | 3178.65 | 3185.43 | 4 | 3 | 1 | 758915.78 | 769992.38 | 7 |
| 289.85 | 3287.05 | 3292.91 | 3 | 3 | 0 | 165830.9 | 183665.01 | |
| 185.14 | 3180.95 | 3189.32 | 2 | 2 | 0 | 180780.82 | 195508.48 | 1 |
| 377.49 | 3369.07 | 3388.43 | 3 | 3 | 0 | 432037 | 332159.07 | 3 |

Figure 3.3: An example of a portion of a peak-report.

In the context of QALM, the most interesting pieces of information are the **pvalue**, **mzmed** and **rtmed**. The first value is used for selecting only the peaks with the highest potential; the latter two are then used to search the raw MS/MS data for matching spectrums-peaks, which will then be used to search for potential peptides, as will be explained in the next few chapters.

# Chapter 4

# Main goals for the thesis project

MS and MS/MS-experiments generate large amounts of data, and analysis of this data can be extremely timeconsuming, not only due to the amount of data itself, but also due to the overhead involved in preparing for and executing each step involved in such analyses. The fact that steps in such a process take part in and are related to different systems (XCMS, Mascot, protein databases, etc.), complicate things further.

QALM has been developed for a specific type of project. The assumptions for such a project have been that a set of results from MS and MS/MS experiments are given, and that a researcher would like to scan these results in order to identify proteins that may have a significantly different abundance in one sample compared to another. Ideally a report would be generated that would list such proteins along with their corresponding P-value or some other form of scoring.

Generating such a report would require the following main steps:

1. Scanning MS-data from two comparable situations with XCMS, and generating a report consisting of a list of differentiated peaks with corresponding retention-times, mass-over-charge values (m/z), and statistical data (P-values, etc).

2. The corresponding MS/MS-data would need to be sent to Mascot, which would identify the differentially expressed peaks with potential peptides and provide a corresponding score for each match. This is not a trivial step: Huge file-sizes would make such a search impractical, if not impossible to perform over the Internet. A dedicated Mascot-server would therefore be required, or the input-files (MS/MS data) would need to be reduced to only the most relevant data.

   The solution proposed in this thesis involves filtering out spectra by searching for and using only those that match peaks identified in step one.

3. For each peak in the generated peak-list (or a selection of only the most significant peaks from the list), corresponding spectra in the MS/MS data (spectra with similar retention times and m/z-values) would be used as input in a Mascot query, and the resulting Mascot .dat-file would be scanned

for matching peptides. Differentially expressed peaks would thus be identified with these peptides (and by extention, with the proteins in which they occur), and such a match would be given a scoring of some form. This however, would again involve sifting through large amounts of data (the Mascot result-file), only a fraction of which might be interesting for the purposes of the current experiment.

These main steps are a somewhat simplified summary of what would be required. To begin with, the first step would require R to be started, XCMS to be loaded, and a handfull of relatively complicated commands to be given through a command line interface. Given the setting, it would be important to understand the relations between the different commands, and how they affect the underlying data. This process may not only be time-consuming, but also error-prone, and unless repeated regularly, the different steps are easily forgotten.

The second step is by no means very difficult, but as mentioned, with large files, the operation may be impractical. Reducing the size of input files may have at least three benefits (besides the obvious need for less diskspace for storage): Smaller files may be submitted over the Internet, allowing for a centralized server to be used by distributed clients, and the actual search and processing may complete a great deal faster if as little irrelevant data as possible is included. Finally, the result from Mascot will consist of a smaller set of matches if the set of spectra in the query is reduced.

Although some data mining tools for Mascot result files do exist, and the web-based reports from Mascot may give some results, a satisfactory way to produce a specific report such as the one descriped previously does not appear to exist.

Finally, there is the issue of files and data: How should all input files be treated and stored? And what of any intermediate files and resulting report-files? Though it is entirely possible to define a set of rules to handle such things in an orderly fashion (specific directory-structures and naming conventions, etc), such a solution may lead to a host of problems in the long run (accidentally storing something in the wrong place, or duplicating or deleting data, to mention a few examples).

Automation of some or all of these tasks may not only simplify and increase the efficiency of the analysis process, but also supply a level of abstraction that may reduce or remove the risk of many of these problems.

## 4.1 Initial goals for the system

Initially, not very much was known about the project; the details were somewhat unclear, and the primary goal for the project was simply to automate the analysis process described in the previous section in a way that would be as simple and efficient for the end user as possible. In the very beginning, it was theorized that this could be achieved by use of the R programming language alone, by implementing a set of scripts to run the required tasks, but this proved to be inadequate as the requirements for the application evoled.

R would however be required to run the first step of the process (generating reports listing differentiated peaks). The result would thus have to consist of a GUI-application with the built-in capability of:

- Starting R, loading XCMS, and calling specific R-functions as needed to analyze and generate a peak report for a set of MS data.
- Reducing the corresponding MS/MS-data from the same situations until it only contained spectrum-data matching those peaks that would be of interest, based on the generated peak-report.
- Submitting the reduced MS/MS data to a Mascot server over the Internet, and recieving the results in the form of a plain-text DAT-file (a Matrix Science / Mascot proprietary format).
- Reducing the result file from Mascot, and formatting it into a final report listing the interesting matches.
- Importing, storing and handling files as needed to support these operations.

After some investigaton, a library for calling and running R-commands directly from Java, called RJava/JRI, was found. RJava/JRI has evolved from two separate projects, one with the goal of being able to call Java-applications from R, the other for calling R from Java. These have been joined together into a fairly coherent system, that it appeared would provide the needed interface between a Java GUI and scripts in R.

Java was thus selected as the main development platform, and a basic plan and set of goals for what to implement was defined as follows (translated from [Leroy09], see Apendix B):

1. A set of R-scripts to automate the handling and analysis of LC/MS data.
2. A GUI that simplifies the set-up and running of such analyses.
3. Some form of "link" between this GUI and Mascot which will make searching the database a natural next step, that can be done from within the same application / user interface as the analyses.

RJava/JRI was specified as the prefered option for the implementation of the R-related parts of the project, but only on the premise that the libraries proved to be mature and stable enough for the task at hand. Due to the uncertainty surrounding this, and the apparent lack of any other realistic alternative for the intercomunication between R and Java, a little extra development time was reserved for this, and the main goal was specified as steps one and two, with step three as optional if there was enough time.

## 4.2 Issues with RJava/JRI, and an alternative approach for interfacing with R/XCMS

After spending the time needed to get familiar with R and XCMS, basic scripts to automate the neccessary functionality were written. These were then called in succession from Java, and seemed to function correctly, but to run notably slower than when called directly from a command-line interface. Closer inspections revealed that when called from RJava/JRI, a batch of scripts would take aproximately eight times as long to complete as when called directly from R

or from the command line of the operating system. This was the case even for very small sets of data. For larger amounts of data, this would constitute an intolerable increase in time required to run the analyses. It was clear that a different approach would be needed.

The requirements for the first step of the analysis process was studied again, and a simple solution was found: Calling the R-scripts directly through the host system, by way of the Runtime-class in Java, which gives access to the environment a java application is running under. By dividing the neccessary calls to R and XCMS into two main groups, and requiring input data for each of these groups to be given at the same time, the whole process could be abstracted into two separate steps.

There were two potential disadvantages to this approach: Firstly, variables and objects created in R could not be referenced directly from Java, as they could using RJava/JRI. Such access however, turned out to be unneccessary, so this would not be a problem. The other potential problem was the risk that commands needed to run scripts might differ slighly depending on the host system. If so, keeping the application cross-platform compatible would require the implementation of different methods to call the R-scripts, depending on the host-system. In light of the huge gains in speed however, this was considered an acceptable tradeoff.

## 4.3 Evolving requirements

Due to the issues mentioned above, as well as uncertainties regarding the details of what exactly the final system should produce, and how it should function, a complete plan for what was to be done was not completed until several weeks into the project. Later, the plan was refined several times due to the complicated nature of the steps in the analysis process and the consequences changing one step would have on others.

An example of this is found in [QALM09] (Apendix C), where terms such as "situation", "collection", and "project" are given specific definitions within the context of the first step of the analysis-process (that in which R and XCMS is involved), and where some of the relations between these entities are touched upon.

A project may for instance contain several situations and collections. Each situation may be added to one or more collections, which may then be "preprocessed". If, however, a change was to be made to a collection after it has been preprocessed, the preprocessing would be invalidated, and so would any analyses and reports that had been generated based on that collection in subsequent steps. The final application would have to be able to handle or avoid this and similar potential problems, and handle the different steps in the analysis process so that no such conflicts could occur. This goes not only for the steps related to R/XCMS, but also to the steps following them, such as MS/MS-file reduction and generating the final report.

The next steps in the process, up to the production of a final report, was elaborated in [Eidhammer10] (Apendix D), which was used as a set of guidelines, although it was revised several times throughout the project.

### A single MS/MS file per analysis

One choice made along the way was to limit the number of MS/MS files per analysis to one. This choice was based on a recomendation to keep things fairly simple in QALM and focus on producing a functional proof of concept for the major tasks.

In practice, there will usually be more than one MS/MS file per analysis. The reason for this is the existance of multiple MS-files per situation; in the type of experiment for which QALM was developed, there will generally be one MS/MS-file for each MS-file.

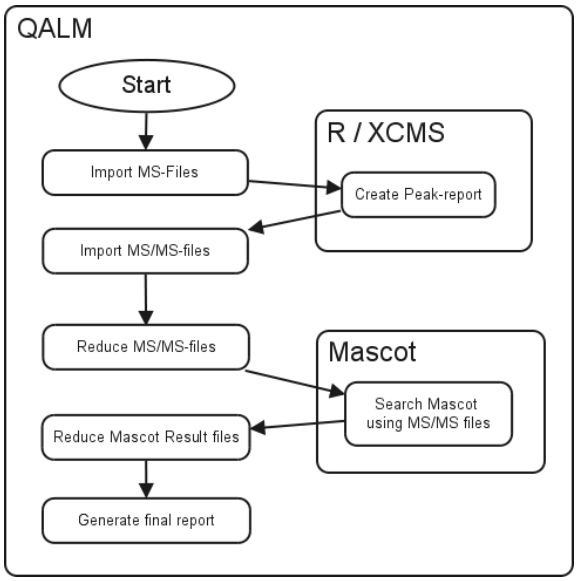



Figure 4.1: The envisioned function of QALM.

### Accessing Mascot

The most significant revision involved accessing Mascot: At the start of the project, it was believed that Mascot could be accessed programatically through some form of interface that QALM would be able to utilize. This turned out to be wrong however, as no such posibility currently exists[1]. The ideal solution (see Figure 4.1) with calls to Mascot integrated into QALM would therefore not be possible. The best alternative seemed to be to export the reduced MS/MS-files and leave the Mascot-search to the user. The resulting dat-file would then have to be imported "manually" into QALM after the search completed.

[1]The background for this belief was an unfortunate misunderstanding regarding the use of Mascot at the University of Bergen: When asked by e-mail whether it would be possible to access Mascot from an external program, an administrator confirmed this. He was however, refering to *Mascot Daemon*; a specific client program for Mascot that runs under Microsoft Windows, and not to a general interface that would enable other third-party applications to access the server.

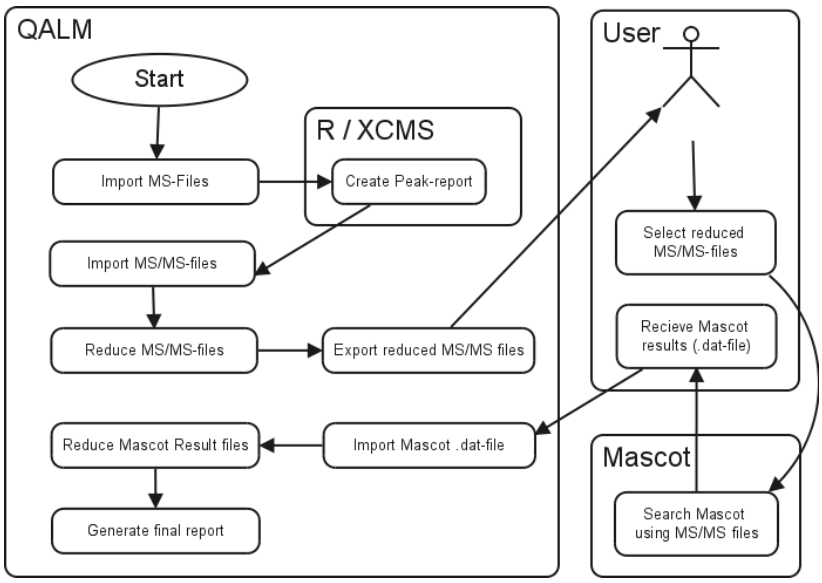



Figure 4.2: The revised solution.

Figure 4.1 illustrates how the tasks to be performed by QALM were originally envisioned: All tasks would be initiated and run from within QALM, including those requiring access to the external systems XCMS and Mascot.

Figure 4.2 illustrates the revised processes: Now the user has to select the files to use as input in a Mascot search, and import the resulting .dat-file from Mascot into QALM afterward to complete a project. Although not ideal, this solution was considered acceptable. The final solution and implementation details are described in the next two chapters.

# Chapter 5

# Running QALM

QALM is a Java desktop application. This chapter presents the main elements in the Graphical User Interface (GUI) and gives an introduction to the function of the application. The next chapter will go into the details of the implementation.

As this chapter does not provide any definitions or detailed explanations, it is recomended to read quickly through it at first, then refer back to it later while going through the technical details described in the next chapter.

This is primarily a guide to the user interface of QALM. For information on how to obtain and install a copy of the application, please refer to apendix A.

In the GUI, there are different "sections" for each of the main steps in a typical analysis process, each with specific buttons, text-fields, lists of data, etc. In the following description, each such section will be refered to as a specific **view**. n

## 5.1 An example run and tutorial

The first time QALM is started, it will connect to the database, create the necessary tables, and display the *project* view (Figure 5.1).

#### Managing projects and general navigation

Under the project-view, a user may create a new project or select and open an existing project from the list on the left hand side. On the bottom left hand side basic information about the selected project is displayed. The buttons on the right hand side are disabled until a project is opened. When a project is opened, they allow the user to navigate to other views for that project. At the top, the menu-bar button "File" has options for creating a new project, closing a currently open project, or exiting QALM. The menu-bar button "view" enables navigation to any view, like the buttons on the right in the project view (unlike these, the view-menu is available from all views).

The buttons near the bottom are also for navigation; these are shown in all views, and for each, allow a user to go to the "previous" or "next" view, or to return to the project-view. The button to the far right at the bottom toggles the processing panel, which displays information and output from some of the more time-consuming tasks performed in QALM.

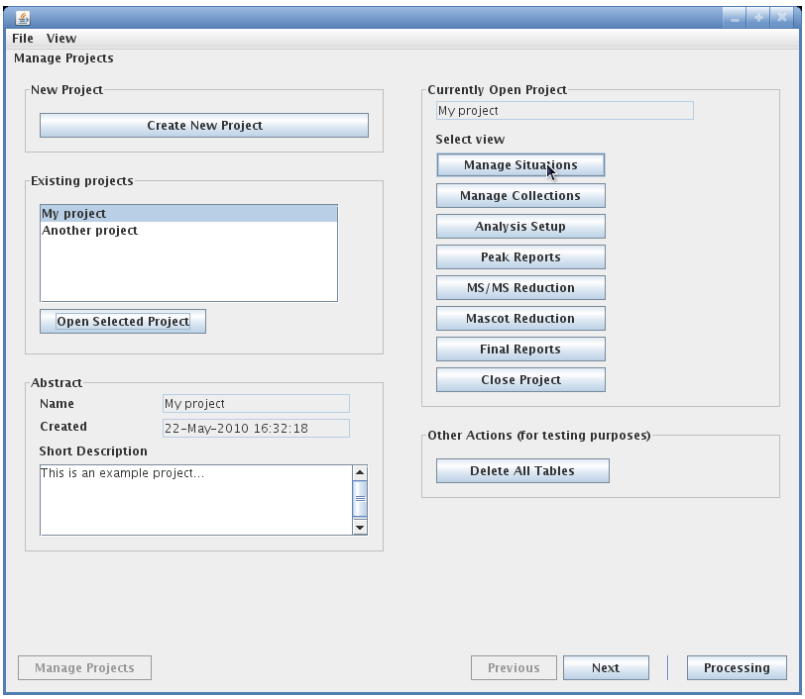



Figure 5.1: The project view in QALM, as shown on startup.

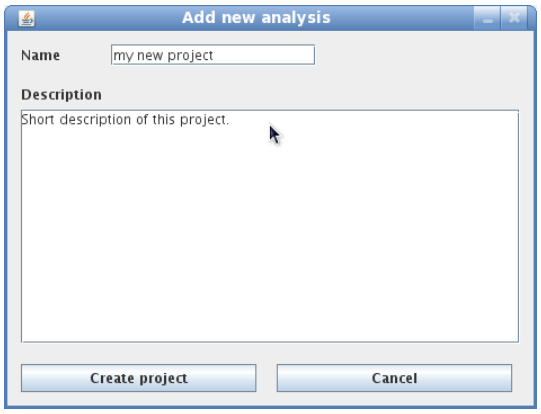



Figure 5.2: The panel for adding new projects.

When a new project is created, the user may enter a name and a description for the project (Figure 5.2). This will be displayed whenever the project is selected in the project-view.

### Importing files

The first step in a QALM-project involves importing files under the *file import* view (Figure 5.3). Here a user may add any number of situations (represented

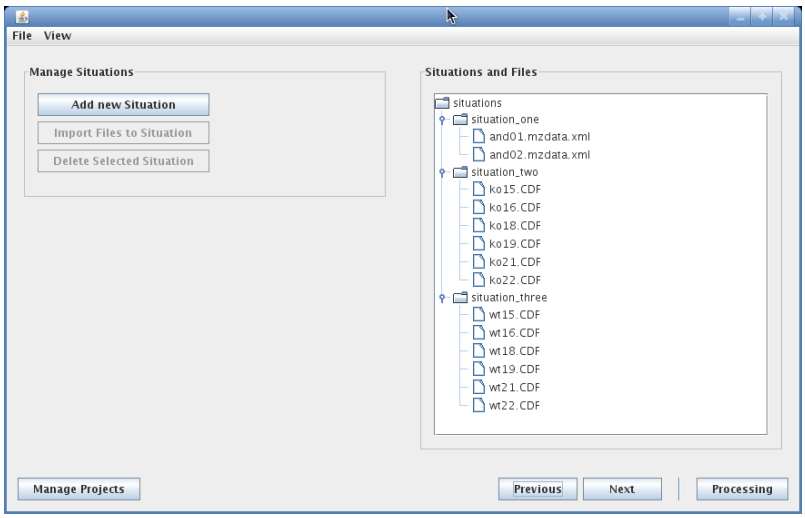



Figure 5.3: The file import view. This project contains three situations and a number of files under each of them.

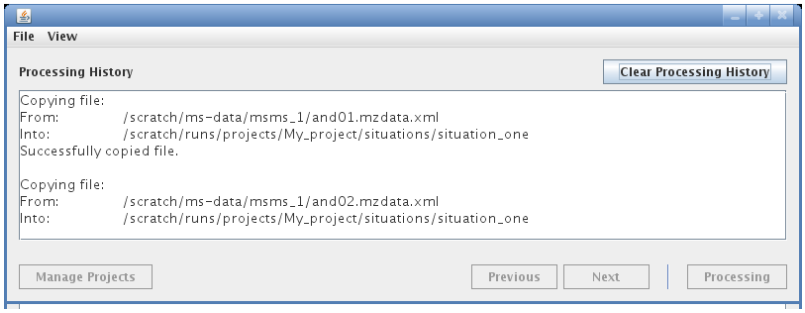



Figure 5.4: During importation, information about the copy-operation is displayed in the processing panel.

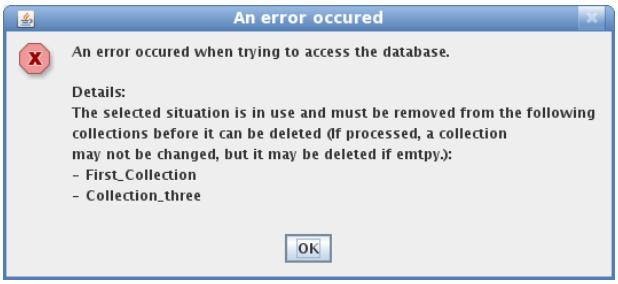



Figure 5.5: Deletion of a situation fails if the situation is in use in one or more collections. This error message lists two collections in which the situation is in use.

as directory folders) and import files into each of them. The results of the import-operations are displayed in the processing panel while they are carried out; when completed, the GUI automatically switches back to the file import view.

For reasons relating to data consistency, single files may not be deleted from situations. In stead, each situation must be treated as a single whole, and may be deleted as long as it is not in use in a collection. If included in a collection, an attempted deletion will cause an exception, and an error message listing the collections to remove the situation from will be listed (Figure 5.5).

### Managing collections

Figure 5.6 presents the collections view. Here a user may create collections under the opened project, or select previously created collections from the dropdown-list on the top left hand side. The list below it will contain all the situations under the currently open project. These can be marked and added to the selected collection using the buttons on the top right hand side. Situations added to a collection will be visible in the list on the top right hand side. After situations have been added, a collection can be preprocessed by clicking "Preprocess Collection". Before preprocessing is initiated however, a confirmation-dialogue will be displayed (Figure 5.7). Such dialogues are displayed before some of the most time-consuming tasks performed in QALM to ensure that they are not initiated by accident.

As long as the collection has not been preprocessed, it is possible to change the name, description, and number of retention-time corrections to perform for a collection using the controlls in the lower left corner of the collections view. Once the collection has been preprocessed however, all butons except "Delete Collection" will be disabled.

Collections may be deleted if they do not contain any analyses. If they do contain one or more analyses, a warning similar to that for situations used in collections will be displayed if deletion is attempted (see Figure 5.5).

### Managing analyses

Figures 5.8 and 5.9 show the analysis-view and the panel for adding new analyses, respectively. Under the analysis view, a user may select a collection, from the top left drop-down list, then either select an existing analysis under that collection from the drop-down list below it, or create a new analysis to add to that collection by clicking the button below it.

In the lower left area, the values for the selected analysis are displayed: It's name, two dropdown-lists for selecting the situations to analyze, and a text-field for adding a description. The list on the lower right side is the analysis list. This is a list of analyses from this project that are ready to be processed as a batch. Selected analyses can be added to or removed from this list with the buttons above it. The lower of the three buttons on the top right side initializes the analyses, which will result in a peak-report for each analysis in the peak-list to be produced.

Note that an analysis must contain two different situations. If the same situation is selected twice in the same analysis, then it will not be possible to add it to the analysis-list.

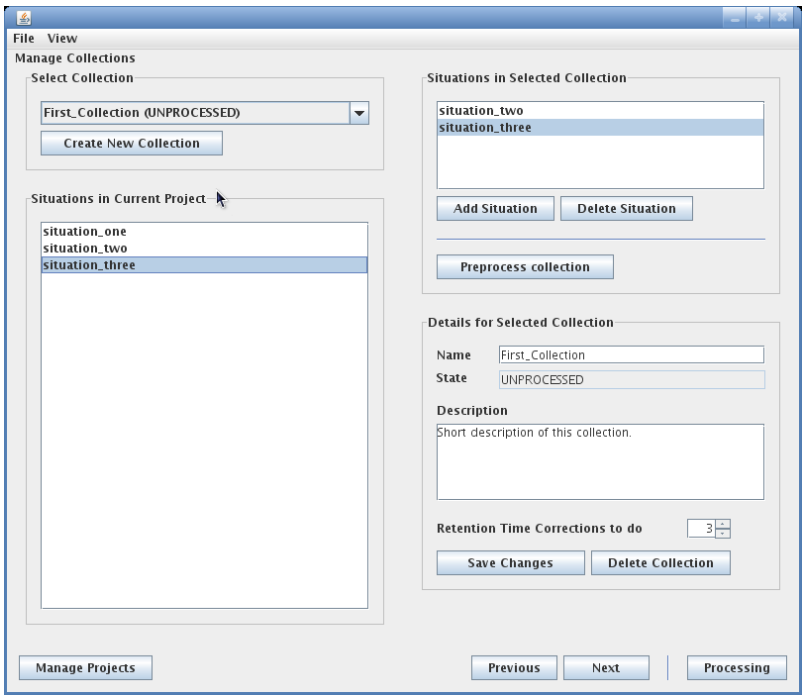



Figure 5.6: The collections-view. The currently selected collection is called "First Collection". It has not yet been preprocessed, but two situations have been added to it out of three possible. When it is preprocessed, retention-time corrections will be performed three times.


Figure 5.7: A confirmation dialogue appearing before some of the most time-consuming tasks performed in QALM.

The output resulting from both preprocessing collections and running analyses is displayed in the processing panel. In Figure 5.10 the last part of the output from a preprocessing-task and an analysis-run resulting in two peak-reports can be seen.

### Peak reports

The reports resulting from peak-analyses are available from the *peak reports*-view (Figure 5.11).

The options near the top left lets a user choose whether to display all analyses from a given project, or to filter the list and display only results from a specific

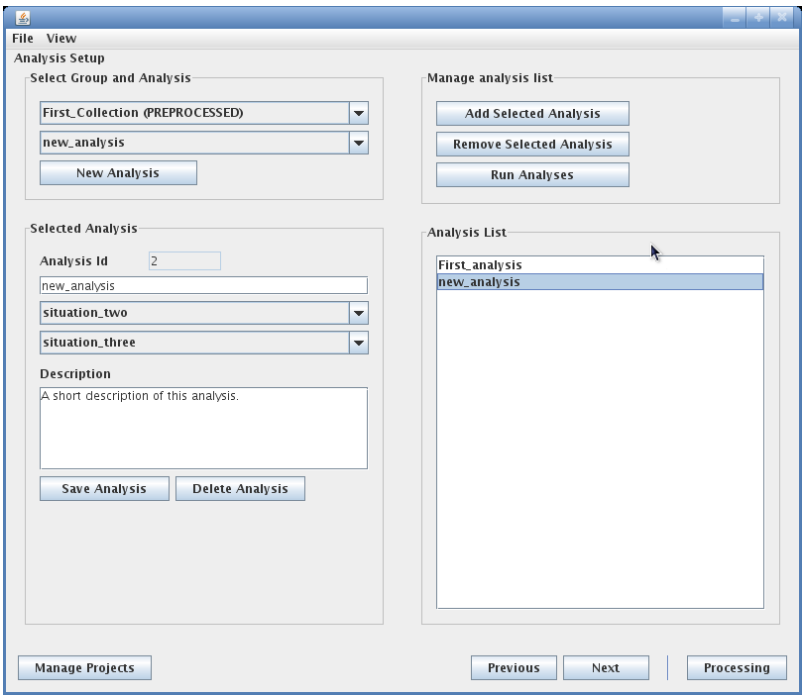



Figure 5.8: The analysis view.

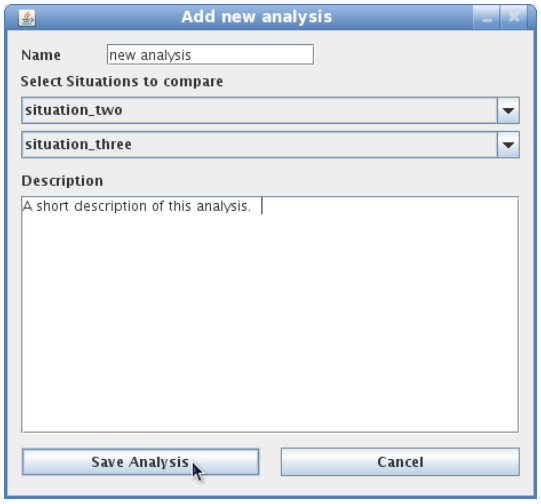



Figure 5.9: Creating a new analysis.

collection. The list on the right hand side contains the names of the peak report-file (typically on the format *<analysis-id-number>-<analysis_name>.tsv* ).

When an analysis in the list is clicked and marked, a summary with the name and P-value of the best peaks will be displayed in the tables on the lower

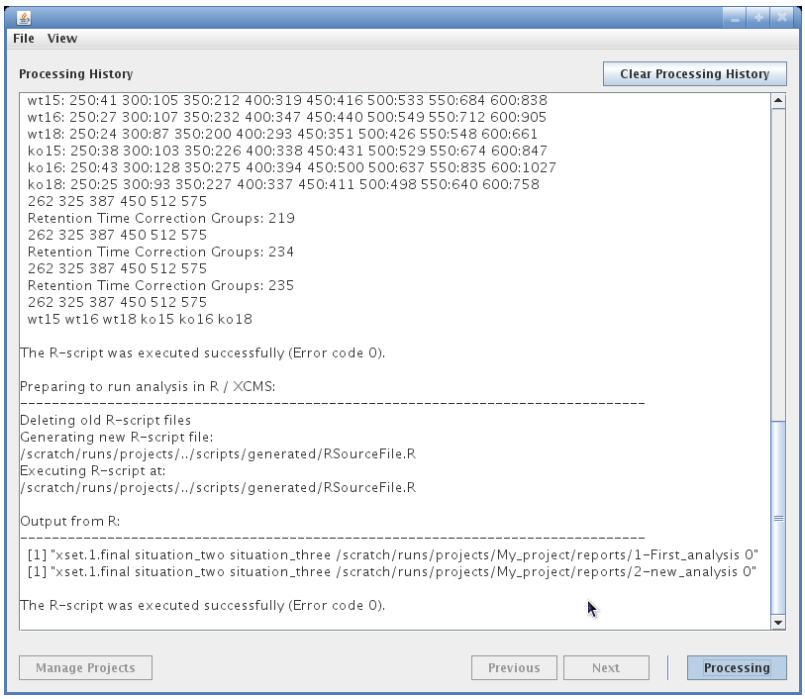



Figure 5.10: The processing panel with the output from preprocessing a collection and generating two peak-reports.

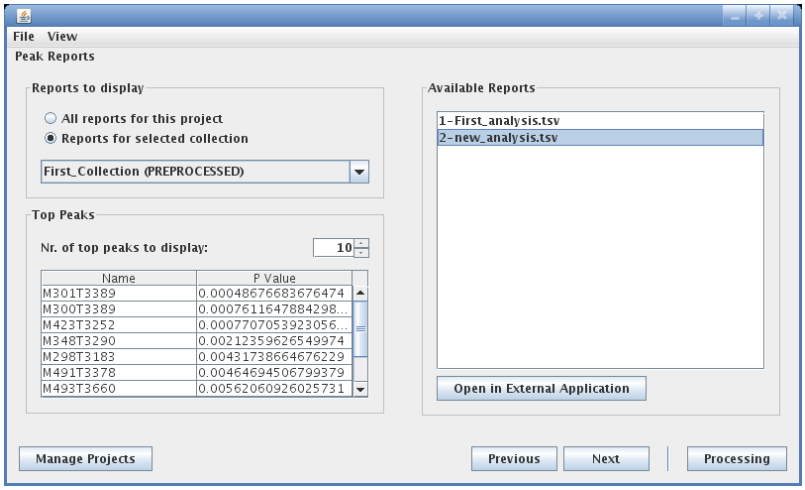



Figure 5.11: The peak reports view, currently displaying two reports from the collection "First_Collection", and the top ten peaks from the second one of them.

left hand side. The "spinner" above the table lets the user decide how many top peaks to display.

Peak reports may also be opened in an external program. Clicking "Open in external application" below the peak report-list will attempt to call an appropriate program for this through the operating system. If that does not succeed, an error-message will be displayed, and the path to the report-file will be presented in such a way that it can be copied. This should enable the user to open the file manually in another application.

### MS/MS file reduction

Once a peak report for a pair of situations has been created, a corresponding MS/MS-file may be imported and reduced. This is done from the *MS/MS reduction*-view, shown in Figure 5.12.

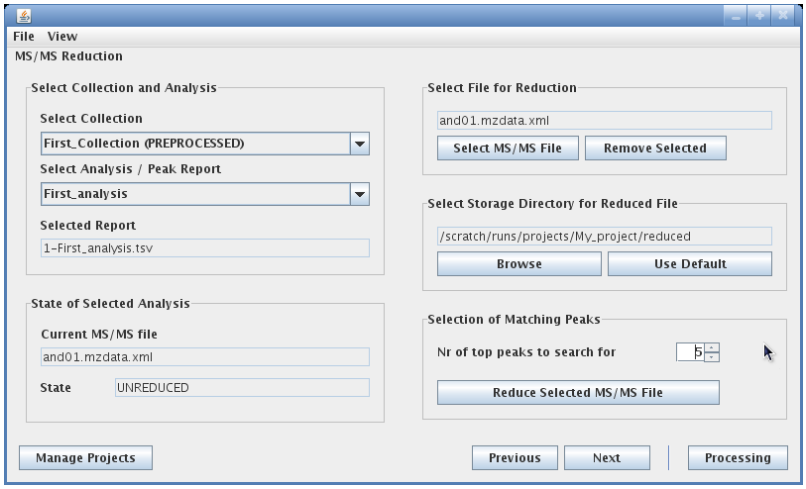



Figure 5.12: The MS/MS reduction view. An MS/MS file and the directory to store the reduced file in has been selected, but reduction has not been carried out yet. Reduction of mascot result files is performed from a similar view.

As under the analysis-view, a collection and analysis may be selected from the drop-down list on the top left hand side. The name of the corresponding peak-report is shown below the selected analysis upon selection. Below this, a summary shows the current state of the analysis with respect to MS/MS-files: If an MS/MS-file has been selected, it's name is shown along with information about it's state, which will be either `REDUCED` or `UNREDUCED`.

On the top right hand side the user may select the MS/MS-file to reduce, or choose to remove a previously selected MS/MS-file. The next two buttons allow a user to specify a directory in which to store the reduced MS/MS-file, or to "use default" directory, typically *<path to the project-catalog>/reduced/*. This is important since the MS/MS file is to be exported and used as input in a Mascot search which the user must perform manually.

The "spinner" on the lower right hand side enables the user to specify how many of the peaks from the peak report to use when scanning the MS/MS-file for matches, while the button below it initiates the reduction process.

### Mascot .dat-file reduction

The selection of data from Mascot result-files (.dat-files) is initiated from a view quite to that for MS/MS file reduction (see Figure 5.12).

The main difference is that only a file is selected; contrary to the case of MS/MS-files, the data from the .dat-files is stored in a database, and not in a separate file, so there is no nead for specifying a directory for storage. Also there is no spinner for selecting a number of peakd; instead, all matches with a score above a threshold specified in an external settings file[1] are automatically included.

### Selecting final reports

The last view in QALM is the *final reports*-view. From here a user may select a collection and an analysis, and output reports for that analysis in a variety of formats. Figure 5.13 shows the view.

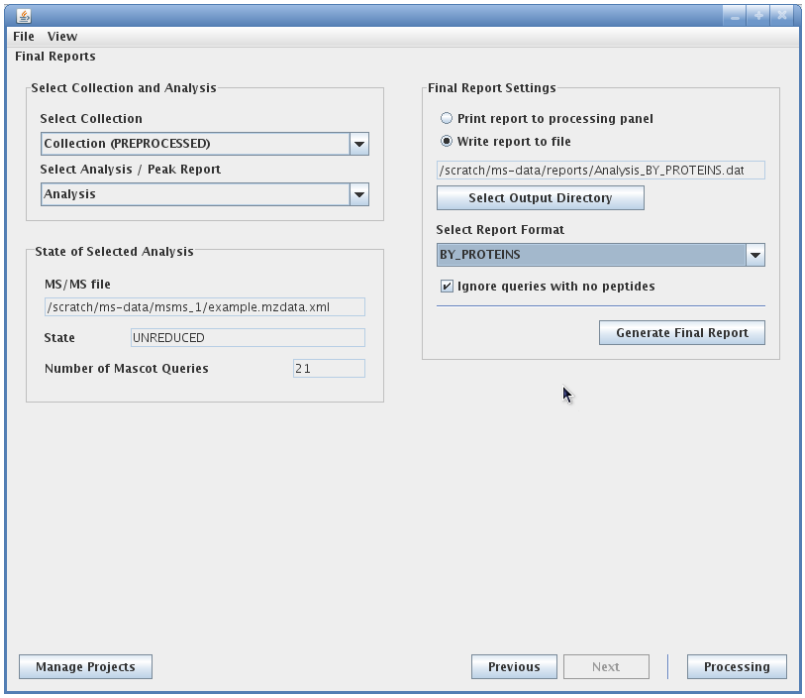



Figure 5.13: The final reports-view.

A collection and analysis may be selected from the top left. Below, some information about the data under the analysis is displayed. On the top right hand side the user can select whether to store the report to file or to output it directly in the processing-panel, allowing it to be viewd quickly and directly from within QALM. If "Write to file" is selected, the button below can be used to select the output-directory where the report will be stored (note

[1] For more information about the settings in QALM see chapter 6.7.5.

that the report-name will be generated automatically in the format *<analysis-name>_<format-name>.dat*).

If a report is to be written to a file, the button for generating the final report will be disabled until a valid output directory has been selected. If on the other hand "print report to processing panel" is selected, the button "Generate Final Report" will be enabled, while the button and text-field for selecting an output directory will be disabled.

The drop-down list on the right hand side allows a user to select which format the report should use. Below this list, a checkbox lets the user choose whether or not to include resulting Mascot-queries for which no peptides were found.

The main contents of each report-format is the same, although certain details may be excluded in some formats. This is discussed in more detail in chapter 6.3.

## 5.2 Notes on the Graphical User Interface

While the main priority in a prototype application such as QALM often tends to be placed on the handling of data in the lower levels of the architecture, the graphical user interface (GUI) also plays an important role. This is especially important considering one of the goals specified in the previous chapter: To "abstract" and simplify the analysis process; in other words, to make it as user friendly as possible.

For this reason, a lot of effort has gone into the design of the GUI, in an attempt to make it as clear and understandable as possible. Some principles that have been taken into account in this respect are:

- To divide the various steps in an analysis process in a clear and logical way, and attempt to hide the more complicated details.
- To strive for consistency and avoid confusion, for instance by making sure the various views and components within them work in similar fashion.
- To give meaningfull feedback as often as possible. This includes both visual and textual feedback. Examples of the former may be seen when adding or removing elements from a list, or in how certain buttons are disabled in specific situations (The buttons and situation-tree in Figure 5.3 are good examples). Examples of the latter may be seen in message dialogues and the processing panel, as shown in Figures 5.5 and 5.10.
- To make the consequences of actions clear to the user before they are executed, for instance by asking for confirmation before starting a time-consuming process or deleting something.
- To catch and handle all exceptions in a similar and reasonable manner, and explicitly inform the user when something goes wrong. There are a large number of specific, detailed error-messages that may be displayed in QALM, and should an error occur that is unknown, the user will still be informed of this through an information dialogue.

# Chapter 6

# Implementation details

In this chapter an overview of the architecture of QALM will be provided along with more technical descriptions of selected components and algorithms.

## 6.1 Definitions

Throughout the development of QALM, several terms have been used to discribe the processes and elements that it works with, and the uses of these have varied over the course of the project, as needs have changed. The following list defines terms as applicable to the final software and the overall process it handles.

### 6.1.1 Projects

A project is the context under which all operations in QALM are done. One project may be open at a time, and opening a new one will automatically close any currently open project. Situations and Groups belong to a given project.

### 6.1.2 Situations

Situations are directories created from within the application, to which files with MS-data from sets of samples are imported. To compare samples from two different sources,such as cells from the liver of a healthy cod, and corresponding cells from the liver of a fish exposed to some form of treatment, one would create two situations. In the first would be placed files with data from experiments on the healthy cod, while data from the other cod would be imported to the second situation. The imported files would need to be MS-data in the **mzData**-format, as explained in chapter 3.2. XCMS will group the data and enable subsequent comparisons between different situations. Situations are imported direcly under a project, and will belong to that project. (It is possible to import the same situations into different projects, but this will result in some duplication of data, as each project has it's own directories with it's own copies of the supplied files and information about these in the database).

Note that for two situations to be comparable in the "analysis"-step follow, they have to contain data from at least two samples each (ie, there must be at lest two files with MS data in each situation).

### 6.1.3 Collections and preprocessing

A collection represents a set of related situations under a project and corresponds roughly to an `xcmsSet` object[1] in XCMS. Several collections may be added to a project, and the same situation may be included in several different collections. For each collection, XCMS will create an `xcmsObject`, attempt to identify peaks representing the same analyte accross samples, and group these together in order to make them comparable in subsequent steps (see chapter 3.2 for details on this process).

The actual invocation of the XCMS-function for grouping ( *group()* ) is just one call in a batch of commands, collectively refered to as **preprocessing** in the application (see chapter 3.3 for details). To the average user, these will appear as a single operation resulting in a change in state for the collection from `unprocessed` to `preprocessed`.

The state of a collection has important implications for the analyses run on it later, and any other elements dependent on those analyses. For a collection to be deleted after having been processed, any analyses and peak-reports belonging to that collection will therefore need to be deleted first. A collection may not be changed after processing, as this would invalidate any previous analyses and results under it.

Furthermore a collection can only be processed once at least two separate situations have been added to it.

### 6.1.4 Analyses

In the context of QALM, an Analysis is a selection of two situations from the same collection, that are to be compared when the analysis is run. An analysis can be created under a processed collection. Several analyses can be created under the same group, and when ready to be run, one or several analyses from one or several different groups can be added to the analysis list (see Figure 5.8), and run in a batch. When an analysis is run, a corresponding peakreport will be generated.

### 6.1.5 Peak reports

A peak-report is generated by XCMS and describes the statistically significant differences in expression of analytes from the two situations in an analysis, as explained in chapter 3.4.

To recapitulate, the values that are of greatest interest here are the mass over charge (m/z), retention time (RT), and the P-value. The first two identify the analyte, while the latter is used to select the most significant peaks for reducing MS/MS-files in the next step.

### 6.1.6 XML

E**X**tensible **M**arkup **L**anguate (XML) is a plain-text format with a hierarchy of elements or "nodes", possibly with attributes. In the following example, the first element (which is called "firstElement") has an attribute ("typeOfAnimal"), the

[1] In short, an xcmsSet contains sets of sample-data in various classes, peaks and peak-groups.

value of which is "fish". The first element also contains two other elements ("secondElement" and "third"), which contain the text "Atlantic Cod" and "Arctic Cod".

```
<firstElement typeOfAnimal='fish'>
   <secondElement>Atlantic Cod</secondElement>
   <third>Arctic Cod</third>
</firstElement>
```

Elements and attributes may be named however needed to represent a set of data, and the hierarchy may have any number of nodes, sub-nodes, and attributes.

An **X**ML **S**chema **D**ocumentation (**XSD**) is a document that specifies how an XML-file representing a specific type of data may be formated, i.e. which elements it may contain, in what order they may appear, etc.

The main advantages of XML are that it is relatively easily read both by humans and computers, and that it's simplicity and flexibility enables it to represent almost any form of data. The mzData-format which will be described in section 6.6.1 is an XML-based format adhering to a specified XSD.

### 6.1.7 Peaks and spectra

A **Peak** generaly refers to a row in the peak report for an analysis (see Figure 6.1). Such a row consists of data such as peak-name, retention time, mass-over charge (m/z), and the P-value for the peak.

The term **Spectrum** will generaly refer to the XML-representation of an MS- or an MS/MS-spectrum in MS- or MS/MS-files (In this context, a spectrum is specific type of XML-element, that may contain subelements containing the MS-data and various metadata; see Figure 6.2 and the description in section 6.6.1). In some contexts however, this may refer to a record in the table "spectrum" in the database.

| | A | B | C | D | E | F | G | H | I | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | name | fold | tstat | pvalue | mzmed | mzmin | mzmax | rtmed | rtmi |
| 2 | 1 | M301T3388 | 9.21 | -20 | 0 | 301.19 | 301.19 | 301.19 | 3388.45 | 3 |
| 3 | 2 | M300T3389 | 9.96 | -18.85 | 0 | 300.19 | 300.18 | 300.2 | 3388.92 | 3 |
| 4 | 3 | M423T3253 | 8.03 | -15.12 | 0 | 423.14 | 423.11 | 423.15 | 3252.57 | 3 |
| 5 | 4 | M548T3182 | 1.15 | -8.42 | 0 | 548.1 | 548.1 | 548.1 | 3182.11 | 3 |
| 6 | 5 | M348T3290 | 8.72 | 14.84 | 0 | 348.16 | 348.12 | 348.17 | 3289.85 | 3 |

Figure 6.1: A peak (highlighted) in a peak-report.

### 6.1.8 The database

For the context of this thesis, a database may be considered to signify a relational database; a solution that allows various types of data to be stored in a set of tables that have clearly defined relationships to each other. The word "database" generally refers to the files or datastructures in which such tables and related data are stored, while a "Database Management System (**DBMS**)" is the software used to create, control, and access a database.


Figure 6.2: A spectrum (highlighted) in a file with MS data. The parameter *msLevel="1"* states that this is an MS spectrum (an MS/MS spectrum would have a msLevel of 2). Note also the m/z-range and retention time.

Generally, any mention of "the database" in this thesis will refer to the local database included with QALM, described in section 6.5. This is a facility for storing data within QALM, and should not be confused with any protein-databases mentioned in earlier chapters, or the Mascot database, which will only be referenced explicitly by that name.

### 6.1.9 MS/MS-file reduction

MS/MS-file reduction is the process of scanning an MS/MS-file for spectra matching the peaks in the peak report from an analysis. The results of such a scan are:

- A new MS/MS-file into which all matching spectra are written (this will be used as the input when performing a Mascot search later).
- The insertion into the local database of a representation of the match (including spectrum, report-peak, and a match between them - see section 6.5).

### 6.1.10 Results from Mascot: ".dat-files"

When Mascot is searched, the result may be retrieved as a text-file with the file-ending ".dat". The terms ".dat-file" and "Mascot result file" will be used interchangably to signify such files throughout this chapter.

### 6.1.11 Final reports

"Final reports" refers to the text-files generated as the final step in QALM, and should not be confused with the peak-reports generated by XCMS. Various formats exist, which essentially display the same data in various ways. These are described in section 6.3.

### 6.1.12 The background processor

The background processor is a solution that allows separate "threads" or a program to perform two separate tasks at once; some form of work is done in the background, while related output is printed in the processing panel (see for instance Figure 5.4). The solution will be discussed in more detail in section 6.7.4.

## 6.2 Overview of the architecture

Many different components are integrated in QALM. This section gives an overview of how they work together. The most important components are discussed in more detail in section 6.7. Instructions for obtaining a copy of the source-code and API documentation for QALM are included in Apendix A.

### 6.2.1 Layers and division of responsibilities

The design of the application has four layers, each of which has separate areas of responsibility. In addition, there are a few components that do not fit clearly into any layer, as they are called from and used by other components from several layers. These are displayed in a separate box near the bottom right corner in Figure 6.3.

Although it does not correspond exactly to classes or software-packages, the following diagram is usefull for getting an overview of the application. The arrows show the main ways in which each component makes use of other components.

#### The presentation layer

This contains the Java GUI. With the exception of the reports, all things related to graphics and presentation is handled here. The main Java Form, which contains all the other panels (and is also the entry point for the whole application) is the class `no.kjartanleroy.codproject.gui.QALMGui`.

#### The Control layer

As the name implies, the main logic, or control of the application runs through this layer. All major operations are called from here after being initiated through the GUI.

There is one main Java Class here (`Controller.java`, technicaly under the package `no.kjartanloery.codproject.gui`), which forewards calls to other parts of the system or carries out operations on behalf of the GUI.

#### The Data layer

The `Adapter` class (under `no.kjartanleroy.codproject.persistence`), acts as a go-between for components that need to access the or the `StateManager`. In addition, some data-validation is performed here.

The other components in the data-layer are responsible for handling various types of data, as corresponding to their names. They may read from or write

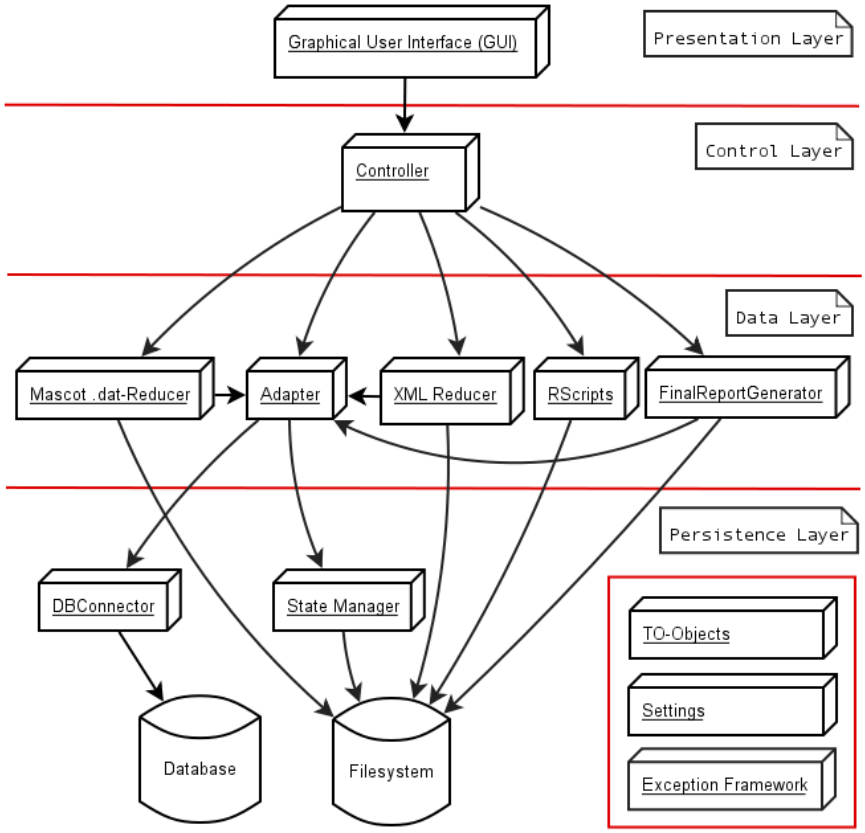



Figure 6.3: Basic architecture of QALM.

to files on the host system that are relevant to their function, or access the database through the Adapter.

### The Persistence layer

The `StateManager` class responsible for creating, storing, and reloading information about the file-structures of an open project (the situation-directories and files).

`DBConnector` is responsible for connecting to the database. The methods for executing each command is in that class, while the SQL-code is located in the class `SQLScripts`.

### Other components

**Transfer objects** are classes that are made specifically for carrying information of specific types. With a few exceptions, there are TO-objects under QALM corresponding to each of the tables in the database (see Figure 6.9 and 6.10). Each of these can contain information corresponding to that in a database table, and may be used in all layers, from persistence to presentation.

A host of different exceptions can be thrown if some task fails in QALM. The **excepion framework** ensures that all of these are caught and handled in an orderly fashion; when something goes wrong, an error-message is displayed, if possible with information specifically relevant to the error.

The text displayed after exceptions and in other information-messages are stored in a java *properties*-file, and technical settings are stored in another. The classes `Settings` and `TextStrings` are responsible for importing these settings and texts.

### 6.2.2 More on the function of QALM

Table 6.1 provides some short descriptions of functionality in QALM that may not be obvious from the preceeding section. More detailed explanations of selected components and solutions are found in section 6.7. In the table **DB** refers to the local database and **OS** to the underlying Operating System. "**Bold**" text refers to class names or Java objects, while "*emphasized*" text refers either to directories or filenames, or variables within classes, as should be clear from the context.

## 6.3 The final reports

QALM can create final reports in four different formats. This section gives a brief description of each of them, then summarizes what each of the data-fields in the reports mean.

#### Full

This may in a sense be consedered the "raw" format of the reports. It displays each match between a report peak and an MS/MS spectrum separately, and the elements under it as a hierarchy: Each match contains a mascot query, and each mascot query may contain a number of matching peptides. Each of the matching peptides may furthermore match a number of proteins (see Figure 6.4).

#### Compact

This contains essentially the same data as the "full" format, but the contents has been organized differently so as not to take as much vertical space: Each query and it's data is shown on the left hand side and each peptide under it in a list towards the right. The protein-list under each peptide occurs at the far right (see Figure 6.5).

#### By Proteins

This report reorganizes the data and presents it by proteins: Each line starts with a protein accession (a protein ID) and data from the peptide it was included for. Farther towards the right, there is one row for each of the matches the protein occured in (see Figure 6.6).

Table 6.1: Functionality in QALM

| **Action in QALM** | **Tasks performed in the background** |
|---|---|
| Starting QALM | DB connection is established and tables created if neccessary. |
| Creating a new project | The name is validated, the project inserted to the DB, and a directory for it created under *projects*. |
| Opening a project | *currentProject* in the **Adapter** is set to the project id. |
| Selecting a new view | A method for the selected view loads relevant data and displays it in a manner dependant on the data's state (eg. disallowing editing an analysis that has run). |
| Creating a new situation | The name is validated and a directory created under *situations*. The situation is added to the DB, and a **TreeModel** for storing file-structure is created (or updated) by the **StateManager** and stored in *project.qpj* in the project directory. |
| Importing files to a situation | Using the background processor, the selected file(s) are copied to the situation directory and *project.qpj* is updated. |
| Creating a new collection or analysis | The new item is validated, then inserted into the DB. |
| Preprocessing a collection | Using the background processor, the collection is verified to be ready for preprocessing. A set of R-script commands are prepared passed to **RScriptMaster**, which creates an external R-script and executes it by calling the OS. If it completes successfully, the collection state is changed to "`preprocessed`". |
| Running analyses | Again, **RScriptMaster** is used through the background processor. A script with a command for creating a peak-report for each analysis is generated and executed. If it completes successfully, the state of each analysis is changed to "`completed`". |
| Deleting a situation | If it is in use in any collection, deletion is canceled, otherwise the situation is removed from (a) the DB, (b) the file-system, and (c) the TreeModel. |
| Deleting a collection | If it contains any analyses, deletion is canceled, otherwise the collection is deleted from the DB. |
| Deleting an analysis | All related data is deleted: Mascot data (queries and matches), reduced MS/MS-files, the peak-report, and the analysis itself. |
| Reducing an MS/MS file | After asking for confirmation, any existing (previously reduced) MS/MS-file for the analysis is deleted. The supplied MS/MS file is then itarated over, and spectra matching any report-peaks are copied to a new MS/MS file, and the matches are registered in the DB. |
| Reducing a Mascot result file | For each query in the file, a query is inserted into the DB. For each peptide under the query and each protein under those peptides, a corresponding item is inserted (this hierarchy is most evident in the report format "FULL"; see section 6.3). |
| Generating a report | A list of all the data to output is fetched from the DB. The list is passed to **FinalReportFormat**, which formats the report as selected, and either displays it or saves it as a file in the selected directory. |
| Closing a project | The value of the `currentProject` variable in the **Adapter** and any existing instance in the **StateManager** (which manages the TreeModel of files and directories) is set to `null`. |

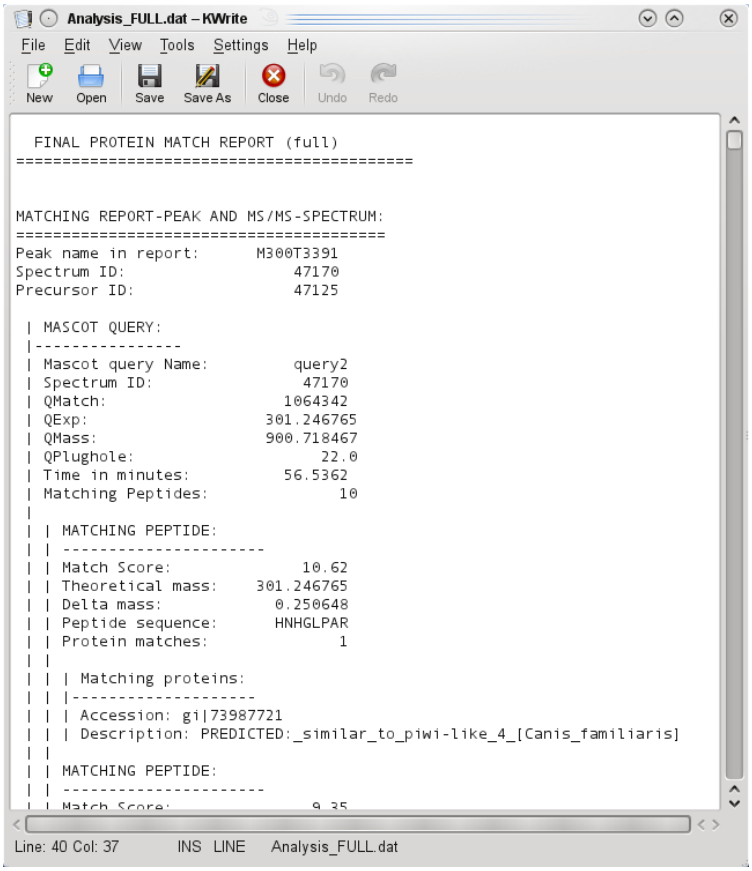



Figure 6.4: The *Full* report format.

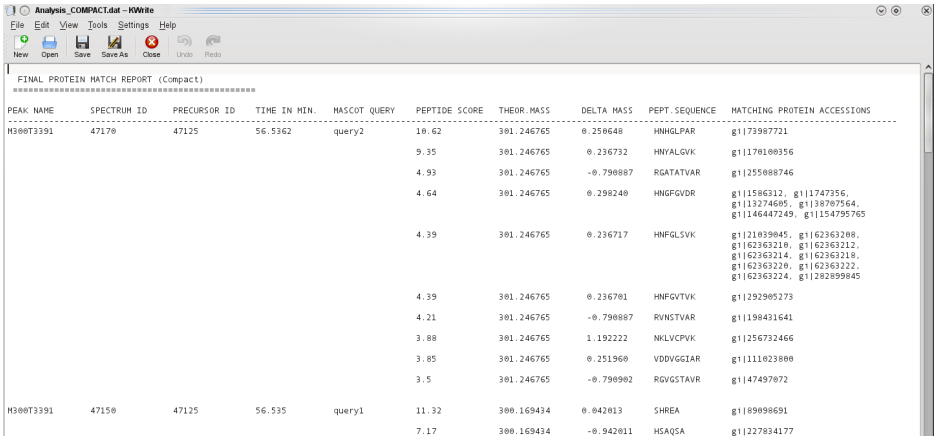



Figure 6.5: The *Compact* report format.

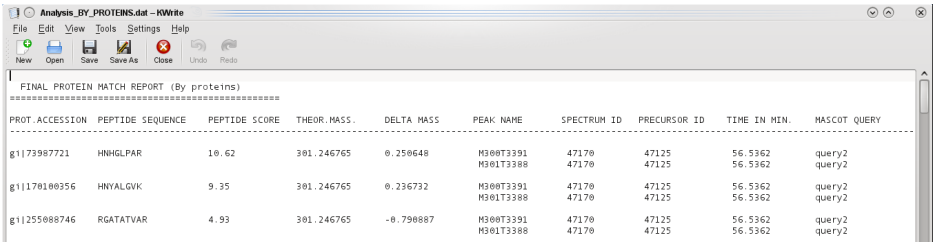



Figure 6.6: The report format *By_ proteins*.

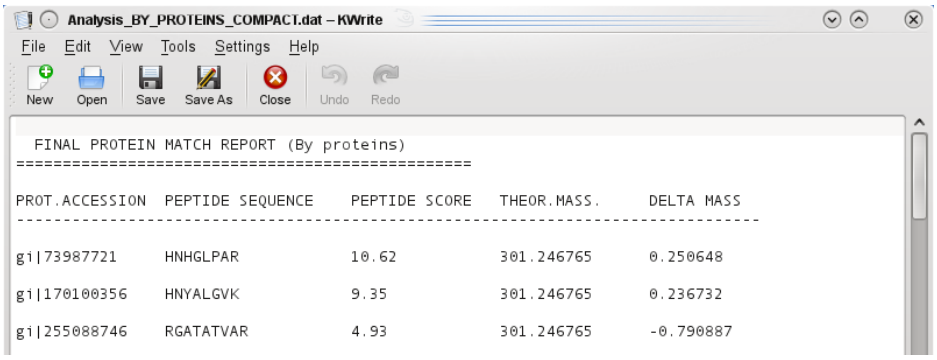



Figure 6.7: The report format *By_ proteins_ compact*.

### By Proteins Compact

The compact proteins-format is similar to the previous format, but includes less data, and only a single line per protein (the matched peaks are not mentioned here; see Figure 6.7). Often biologists will only need to identify differentially observed proteins. In such cases, the information provided in this report will be sufficient.

## 6.3.1 Data in the reports

The data fields in the reports are:

- **Peak name in report:** Name in the peak-report generated by XCMS.
- **Spectrum ID:** The spectrum ID-number in the reduced MS/MS file.
- **Precursor ID:** The ID of the precursor in the reduced MS/MS file.
- **Mascot query name:** Given by Mascot when it performs the search.
- **QMatch:** The number of peptides in the searched protein-database with theoretical masses matching the query mass.
- **QExp:** The m/z-value of the query

- **QMass:** The mass of the query. This is calculated from the m/z-value and the charge (note that the charge is not given explicitly).
- **QPlughole:** A threshold for homology; an empirical measure of whether the match is an outlier[MascotFAQ, p.4-5]).
- **TimeInMinutes:** The retention time in minutes.
- **Matching peptides:** The number of peptides matching a given Mascot query.
- **Match score:** The match-score for a peptide; a value given when Mascot performs the search.
- **Theoretical mass:** The theoretical mass of a peptide.
- **Delta mass:** The difference between the theoretical mass and the observed mass for a peptide.
- **Peptide sequence:** The sequence of amino acid residues in a peptide. Each letter represents one of the 20 regular amino acids.
- **Protein matches:** The number of known proteins in which a peptide occurs.
- **Accession:** A number that identifies a protein uniquely within the protein-database Mascot has searched.
- **Description:** A description of a protein.

## 6.4 File and directory structure under QALM

After instalation, all files and data relating to QALM will be in a directory named *QALM*. This section describes the contents of the various folders and files under that directory.

Figure 6.8 shows an example of a typical directory tree under the QALM directory. The database is stored in *database*, and *lib* is used for various libraries that QALM makes use of. All projects are stored under *projects*. The sub-directories under a project will be created when needed. In the example, under *situations* there are sub-folders for two situations. These correspond to the directories in the file tree in QALM when creating new situations. The actual filetree used to represent the directories within the GUI are stored in *project.cpj*.

*reports* contains peak-reports for analyses. The reports consist of the name of the analysis, with its' database-id prepended to ensure uniqueness.

Unless another directory is selected, reduced MS/MS-files are stored under *reduced*. They are named as the files they are based on, but like analyses, have the analysis-id prepended.

*scripts* contains the various R-scripts that are executed by QALM. The sub-folder *generated* contains the initiating script-file (*RSourceFile.R*), through which the others are called. This file should not be changed manually, as it will be replaced the next time QALM runs; any R-code that may be required to run before executing the other scripts should instead be placed in RPreparations.R; this script is allways called before the others (see section 6.7.3 for details).

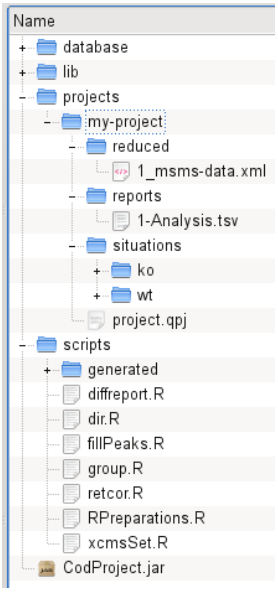



Figure 6.8: Structure and files under the main directory *QALM*.

## 6.5 The database

In the early stages of planning for the application, there was no specification for a database (for example, see [QALM09], apendix C). In stead, files or some form of data structure (e.g. a 2-dimensional array) was specified as requirements for the storage and handling of the information resulting from analyses and file-reductions. For some of this data, the format was a given (peak-reports are generated as stand-alone files by XCMS, and can be used as they are; MS/MS-files must be in their given XML-format to be used as input in a Mascot-search). However, to be able to use the various pieces of information from these files in a reliable and effective way, it would be a great advantage to be able to enforce stricter rules on their interrelationships. This section describes how the database used in QALM enables this.

### 6.5.1 Motivation for a database

A database model lets one define the relations between different things, such as a collection, an analysis, and an MS/MS-file. Once these relations are defined, the database management system (DBMS) does the job of enforcing them. If for example one attempts to delete a collection under which there exists an analysis, the DBMS will hinder this, and instead return an error message describing why the operation can not be completed: The collection has a related analysis; removing that analysis is required before the collection can be removed, otherwise, the remaining data would be inconsistent.

In similar fashion, searching, retrieving, sorting and filtering data is made easy by use of SQL (Structured Query Language). This is especially usefull when it is not known what needs may arise at a later stage. When unforseen

changes or requirements occur, manipulating SQL-queries (and possibly tables and relations) may be an effective and simple way to adjust, and new types of data and new relations can be added easily.

Also, there is the subject of standardization, both on the technical level, and on the usability level, for documentation purposes: If one would prefer to use a different DBMS, it may be switched with only minor (if any) adjustments in the SQL-code, and a diagram such as the ones below (Figures 6.9 and 6.10) immediately give an overview of the different types of data and their relations in a format that is well known and understood.

To recapitulate, some of the major advantages of using a database, as opposed to developing specific formats and data structures are:

- Reliability on defined relations for a number of operations
- The ability to expand organicaly as new needs arise
- Standardization and simplicity through use of a technology that has been in use throughouth the industry for decades.

### 6.5.2 Apache Derby / JavaDB

QALM currently uses Apache Derby, a small Java based DBMS, available under the Apache License, version 2[2]. It was chosen for it's simplicity of use; Derby can be run seamlessly from inside a java application, and can be deployed with it without requiring any installation or setup.

Later, with a more mature version of QALM ready for production, it would be advisable to switch to a more advanced and more thoroughly tested DBMS, which could be administered separately from QALM itself (such a system might also function over a network).

### 6.5.3 Data and the relations between them

Figures 6.9 and 6.10 give an overview of the data in the database, and the relations between them. Note that for presentational purposes, the model has been divided in two; the table *analysis* is displayed in both diagrams, and acts as the "link" between the two sections.

Some of the elements in the diagrams will be discussed in detail. The meaning an functions of others should be clear from the context, as they make use of fairly standard practices and have more or less self-explanatory names. The "*<something>_id*" columns for instance, serve either as primary keys identifying a record uniquely within the table, or as foreign keys linking two tables together[3], while *description* in most tables means exactly that; a description of the respective entity, as entered by the user in through the GUI[4].

In the following discussion, table names are in **bold**, while names of columns in them are *emphasized*. `Typewriter` text indicates values of data, either in a table column or in a Java class. Whenever names are used without such

[2]See: http://www.apache.org/licenses/LICENSE-2.0

[3]There are two exceptions to this rule: *precursor_spectrum_id* in the table **spectrum**, and *spectrum_id* in **mascot_query** refer to an id-number in a file, as discused later in this chapter.

[4]**protein_match** is an exception. Here *description* is obtained from a mascot result file.

formating, they will be refering to real-world instances of the items and the relationships between them, and not the objects in the database which are attempts to model those relationships.

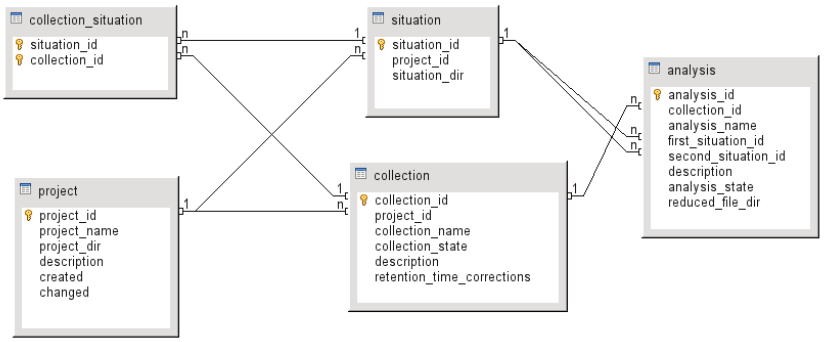



Figure 6.9: The database (project-related tables)

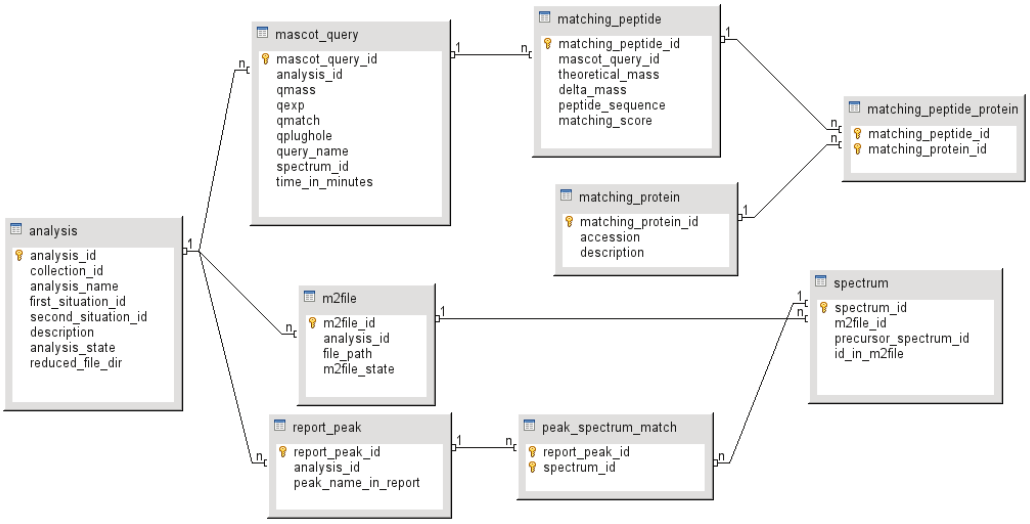



Figure 6.10: The database (tables with data from experiments and searching).

### Details concerning tables in Figure 6.9

The first Figure shows tables with data relating to the structure of projects in QALM. The relationships between the central "objects" are in brief:

- A project may contain any number of situations and collections
- Each situation may be added to a number of collections (but only once to each collection).
- An analysis consists of two situations and belongs to a specific collection.

In the **project** table, *project_dir* should contain the name of the directory under which the data from the project should be stored. This should be a single directory name, not a path; when the project is created, the directory will be created under the projects-directory, as mentioned in section 6.4. *created* and *changed* are timestamp-values that were intended to keep track of when a project was created or last changed respectively (currently, support has only been implemented for the former).

In the table **situation**, *situation_dir* is the name of the directory where the situation is located (see section 6.4).

In **collection**, *collection_state* is a string-representation of a value from a Java Enum type[5]. It can be either "`processed`" or "`unprocessed`". *retention_time_corrections* is an integer that defines how many times a collection should go through the cycle of retention time corrections and regrouping during preprocessing (see chapter 3.3.5).

Like *collection_state*, *analysis_state* is a string from an Enum type. It can be either "`ready`", if the analysis has yet to be run, or "`completed`" if it has been completed. *reduced_file_dir* is the directory in which a reduced MS/MS file for the analysis will be stored.

### Details concerning tables in Figure 6.10

The tables in the second Figure contain data from experiments and analyses. Following roughly the order in which the various data are used throughout a typical project in QALM:

- A **report_peak** corresponds to one row of data from a peak-report (as generated by XCMS) for which a matching MS/MS-spectrum has been found. Any number of **report_peak**-records may be linked to a given **analysis**.
- A **spectrum** is related to a record in **m2file** (representing MS/MS-files), and may match one or more **report_peak**-records.
- An MS/MS file (**m2file**) is related to an analysis (**analysis**), and can contain many spectra (**spectrum**).
- A **mascot_query** contains the results of one of the queries from a mascot search (it therefore corresponds to a specific **spectrum**-record; that from the spectrum used as input for the search).

[5]Simplified, an "Enum" is an object that may take on values, one at a time, from a list of specific, immutable "states" defined in the Enum itself.

- A mascot query (**mascot_query**) may include a number of peptides (**matching_peptides**); potential peptides in that they were found to match the criteria from the spectrum during the Mascot search.

- A peptide may be found in several different proteins, so **matching_peptide** may refer to several **matching_proteins** (note that different proteins may also contain several of the same potential peptides).

Each peak for which a matching MS/MS spectrum is found is inserted into **report_peak**. This feature exists mainly to make it possible to refer back to the report-peak from a matched spectrum. Some formats of the final report generated by QALM include the names of the peaks/rows (such as eg. "M501T3371") in the peak-report for the same analysis. Such names are stored in *peak_name_in_report*. Note that the report itself is not referenced here, nor anywhere else in the database. A unique filename for the report is created based on the name and id of the analysis, and it is left to other components of QALM to handle any references to this.

As mentioned, a **spectrum** refers to a spectrum in an MS/MS file. A spectrum will be inserted to the database when a match occurs during the reduction of an MS/MS-file, and will therefore be related to the **m2file** representing the file it was found in, and the **peak_report** it matched. Note that the table does not contain any data from the MS/MS-file, but only the ID-number the spectrum has in the MS/MS file (*id_in_m2file*), and the ID of the precursor-spectrum (*precursor_spectrum_id*. See section 6.6.1 for more information on the format of MS/MS files).

In **m2file**, *file_path* stores the path to the MS/MS file that will be reduced (not to be confused with the final reduced MS/MS file, which may be stored elsewhere). In *m2file_state* the state of the MS/MS-file is stored. It can be either `unreduced` or `reduced`. This state relates only to QALM, and not the file itself, which is never actually changed; `reduced` simply means that a new file containing portions of the data from the original MS/MS file (ie, a "reduced copy") has been created.

**mascot_query** stores selected data from each query in a Mascot .dat-file. Each query in such a file corresponds to a specific spectrum from the MS/MS-file used to initiate the Mascot search. This means that once a .dat-file has been reduced, there should be one record in the **mascot_query** for each record in **spectrum** under the same analysis (remember a spectrum belongs to an analysis by way of its relation to an **m2file**). This also means that although there is no direct relation between them in the database, the columns *spectrum_id* in **mascot_query**, and *id_in_m2file* in **spectrum** refer to the same field in the same file. For corresponding records, these id's should therefore be the same.

This solution may perhaps appear problematic at first, since the same ID is stored twice, in two different places, with no relation between the two instances. Remember however, that although related[6], the two instances originate from two different sources. It is conceivable that a different database design would handle this issue better. This might be required if QALM is to be expanded further, for instance to support several MS/MS-files and corresponding mascot

[6]The mascot query being the result of a search with the spectrum as input.

results in each analysis. For the current state of QALM however, the issue does not appear to pose significant problems.

As a sidenote to this, it should be pointed out that if a user should accidentally choose to import the wrong .dat-file before attempting to generate the final report, this is likely to be detected, since it is highly unlikely that all pairs of values for *id_in_m2file* and *spectrum_id* will be the same. This will result in an exception being thrown and a warning about a "mismatch between MS/MS data and mascot data". Even so, the addition of a more explicit check to ensure that the correct files are used is recomended for any future implementation.

The remaining columns in **mascot_query** correspond to similarly named fields in Mascot .dat-files. The same is true for non-key columns in **matching_peptide** and **matching_protein**. All these values are used in the final reports generated by QALM, and are defined under section 6.3, where the various report formats are described.

### 6.5.4 Issues with the current database

For practical reasons relating to time and efficiency, the early test-runs of QALM were carried out using relatively small data-sets and MS/MS-files (typically between 1 and 100 megabytes). This appeared to be sufficient to demonstrate the function of the application, and for a long time, no problems relating to the database were observed.

Later, when QALM was tested with bigger files (MS/MS files ranging from a few hundred megabytes up to several gigabytes in size), timeouts started to occur when a lot of data was inserted into the database[7]. The reason for this error has not been identified, but it may be related to the strain put on the database when large amounts of data are being inserted in a short period of time. During reduction of .dat-files for instance, a large number of calls to the database may occur in rapid succession.

In any case, the result of such an interruption will be that not all data for a given MS/MS-file or .dat-file is inserted into the database. When this occurs, the only current solution is to repeat the process; the existing data for an analysis will then be deleted[8], and hopefully a new, successfull operation will replace it.

For the future, there are several actions that may be taken to avoid this error:

1. Currently each piece of information is inserted into the database individually (each Matching Protein, for instance). An alternative approach grouping such pieces of data and inserting it in batches (so-called "prepared statements") would significantly reduce the number of calls to the database and might reduce the risk of such "timeouts".

2. Replacing the Derby-database with a bigger, more heavy-duty DBMS with better support for such intensive usage (if an application like QALM is to be used in practice, this should be done regardless of this specific time-out error, as discussed in the introduction to Derby in section 6.5.2).

[7] Technically, the error states that "*a lock could not be obtained within the time requested*".

[8] When reducing an MS/MS-file or a .dat-file, it is standard procedure to check for any existing data and ask for confirmation before replacing it.

3. In addition, the code for connecting to the database and performing the insert-commands from QALM should be scrutinized again for potential weaknesses that may be contributing to this problem, or alternative solutions that may result in better performance.

## 6.6 File formats

In this section, the basic formats of the mzData-files and the mascot result files will be described. Algorithms for mining such files for information and reducing their sizes are discussed in section 6.7.

### 6.6.1 Initial MS and MS/MS data: The mzData format

The reason for limiting support to one format is the need to mine the files for data later (see section 6.1.9). For the first version of QALM, adding support for more than one format would be timeconsuming and impractical. The selection of the mzData format was not a decision taken during the implementation of or for the sake of QALM - it was simply the format in which test data was supplied[9].

mzData is an XML-based format adhering to certain rules and specifications, as defined in it's XSD. Citing the abstract from [mzMemo]:

> *(..) mzData is an XML format for representing mass spectrometry data in such a way as to completely describe the instrumental aspects of the experiment. The key feature of the format is the use of external controlled vocabularies to allow data from new instruments and new experimental designs to be shared in a common format.*

In spite of this aim of being able to support future instruments and new data, the mzData format is currently deprecated by the Proteomics Standards Initiative. The reason for this was the parallel existance of another such format; mzML, developed at the Seattle Proteome Center at the Institute for Systems Biology. The two formats have now been replaced by a single new format, mzML, which incorporates the best ideas from each of the two. mzML was released in June 2008, and updated in june 2009.

Support for the new format has not been added to QALM yet for two reasons:

1. The test-data supplied was in the mzData format, so this is what was available from the onset of the project.

2. QALM is primarily a proof of concept, and although it would be an advantage, using the new format is not a crucial requirement at this stage. In any case, adding support for mzML in the future should only require the replacement of a single component of QALM.

[9] This may have depended on the instrument used; the format a given mass spectrometer may export data to may vary from instrument to instrument.


Figure 6.11: An mzData file: Root node and description.

### The contents of an mzData file

Figures 6.11 to 6.14 show the structure of an mzData file. The data is from a set of samples provided for the tesing of QALM.

Note that the "+" and "−" symbols are not part of the file, but rather controlls in the program displaying the file. They allow sections of the file to be "collapsed" or hidden from view while navigating it. In Figure 6.11 for instance, the elements **instrument**, **dataProcessing** and **spectrumList** have been collapsed. Figure 6.12 shows the **spectrumList** element and one MS-spectrum from the list under it expanded (note the attribute *count* in **spectrumList** - this is used by Mascot during the search to follow). Figure 6.13 shows an MS/MS spectrum with a **precursorList**. Two **data** elements under these spectra are collapsed; Figure 6.14 shows what they look like when expanded.

The first element, **mzData**, is the root-node or element, which contains all the other nodes in the document. The **cvLookup** element contains the name, location, version of a controlled vocabulary source, and a label used to reference it from within the document. **description** (Figure 6.11) includes various descriptive information about the sample, it's source, the instrument used, and more. All this information is kept as-is when QALM reduces an MS/MS-file. In practice, the section is copied as a whole into the new, reduced document.

Figure 6.12 shows four collapsed spectra under the spectrum list, followed by one expanded MS-spectrum while Figure 6.13 shows an MS/MS spectrum. Of special interest in the context of this thesis are:

- The attribute `id`, identifying each spectrum uniquely within the file. The columns `id_in_m2file` and `spectrum_id` in the database tables **spectrum** and **mascot_query** respectively, refer to this id.

- The attributes `mzRangeStart` and `mzRangeStop`, which identify the m/z-range for an MS-spectrum.

- The attribute `msLevel` under each **spectrumInstrument**-node. These


Figure 6.12: An mzData file: The spectrum list with one MS-spectrum expanded.

identify the spectra as either MS-spectra (if msLevel is 1), or MS/MS-spectra (if msLevel is 2). is marked with a red

- The **cvParam** nodes that have the `name`-attribute "timeInMinutes". It's `value`-attribute as a number; this is the retention time of the spectrum.
- The **cvParam** nodes that have the `name`-attribute "MassToChargeRatio". The number under `value` is the m/z-value of an MS/MS spectrum.

The last three are marked with red circles in Figure 6.13.

The nodes **mzArrayBinary** and **intenArrayBinary** contain base64 encoded binary data. **mzArrayBinary** contains the list of m/z-values for a spectrum, while **intenArrayBinary** stores the intensities for each member of that list.

Note the various types of content represented in the XML format. The start-elements (like <**spectrum**>) , end-elements (like </**spectrum**>), and attributes have already been mentioned. In addition there can be characters (plain text), whitespace, comments, and more. This is important because the algorithm used to traverse the XML data must be able to identify and classify each piece of the document and handle it correctly, a process which will be detailed in section 6.1.9.

More information about the mzData format may be found at the Proteomics Standards Initiative website[10].

### 6.6.2 Mascot Results: .dat-files

When a mascot search is carried out, a plain text file containing the results is produced. From this file, various reports may be genarated an exported. The

[10]Description and references for mzData from the Proteomics Standards Initiative: http://www.psidev.info/index.php?q=node/80#mzdata


Figure 6.13: An mzData file: An expanded MS/MS-spectrum.


Figure 6.14: An mzData file: A field containing binary data.

contents and formats of these reports may vary depending on what type of search is performed (Peptide Mass Fingerprinting, Sequence Query, or MS/MS Ion search, as in the case of QALM) and the result one is looking for.

In stead of making use of one of the exported formats, QALM mines the original .dat-file itself for the requested data. The reason for this is mainly to ensure that all the data is available in case it should prove neccessary later.

### The Contents Of a .dat-file

Before the list of results in a .dat-file, there are various settings, definitions and other data. This data is currently not used in any way by QALM, and will not be discussed here. Instead, focus will be on the sections of the .dat-files from which QALM imports data to the database.

The input for a Mascot search consists of an MS/MS file containing a number of spectra. Each of these spectra effectively becomes a separate search query,

and the .dat-file from a search will contain separate results for each such query. The overall data is divided into sections with various types of information, and results from each of the queries may be represented in each of these sections.

The sections of importance for the function of QALM are:

- **The summary section**, which starts with:
  `Content-Type: application/x-Mascot; name="summary"`
  For each query, there are four lines; one for each of the following:
  - qexp: The mass over charge and charge, separated by a comma.
  - qmatch: The number of peptides in the database matching the mass.
  - qplughole: The threshold for homology.
- **The peptides section**, which starts with:
  `Content-Type: application/x-Mascot; name="peptides''`
  This contains a list of all the matching peptides for each query, and for each such peptide, a string with data separated by commas. This data includes:
  - It's theoretical mass
  - The delta mass (deviation from the theoretical mass)
  - The peptide seqence.
  - A score for the match.
  - A list of matching proteins (proteins in which the peptides occur).
- The input sections, with information about each query, begining with:
  `Content-Type: application/x-Mascot; name="query17"` Among other things, this section contains information about the queries stored in a single line (the "titleString") in the following format:

  ```
  spectrumId=47062 TimeInMinutes=7.844 somethingElse=(...)
  ```

  The most important attributes for QALM here are the `spectrumId` and the retention time (`TimeInMinutes`).

What the various values mean is described in the section discussing the final reports generated by QALM (section 6.3). The process for getting the required information from .dat-files will be described in section 6.7.2.

## 6.7 Selected algorithms and other solutions

This section describes some of the technical solutions in QALM in more detail. Note that some familiarity with general programming-terms and algorithms are assumed. Instructions for obtaining a copy of the source-code and API documentation for QALM are included in Apendix A.

### 6.7.1 Reducing MS/MS Files

To reduce MS/MS files, QALM makes use of Java classes and objects that represent the various types of XML elements that occur in the mzData files. The classes used for this were generated using the tool **xjc** under the Java Architecture for Xml Binding (JAXB). JAXB enables the "marshalling" of Java objects

into XML objects for the purpose of storage, then "unmarshalling" them back into Java objects. Given a reference (URI) to an XML schema (an XSD document) and the name of an existing directory, the xjc-tool produces a set of Java classes containing members corresponding to the XML-nodes and attributes defined in the schema[11]. Figure 6.15 shows a repetition of the procedure used to generate the classes used by QALM. (The classes used in QALM are located in the java package `no.kjartanleroy.codproject.xmlbindings`).

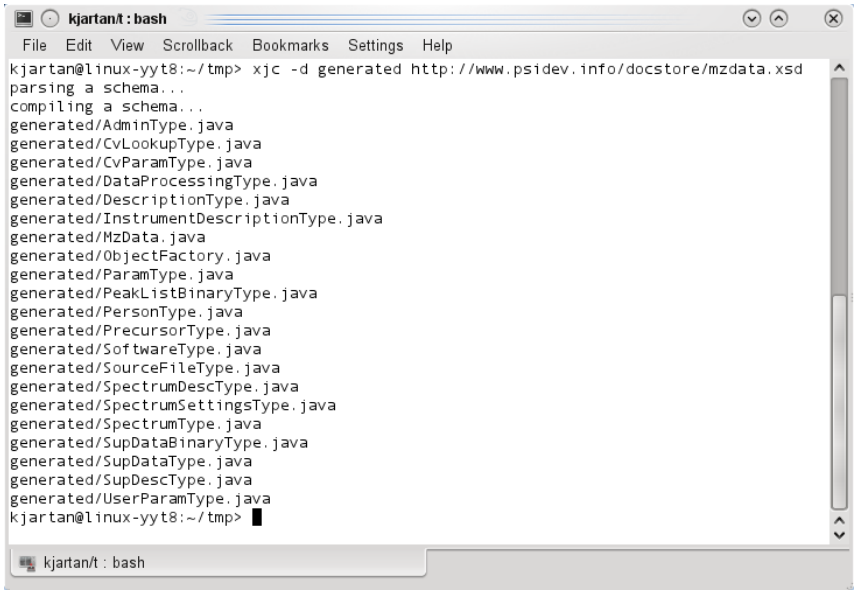



Figure 6.15: Demonstration of how XJC was used to generate Java classes used by QALM: Java-files corresponding to elements in the XML schema *mzdata.xsd* are stored in the directory *generated*.

### Overview of the algorithm

QALM reduces MS/MS files through four main steps:

1. Iterating over elements in the original file before `<spectrumList>`, and outputting each of them to a temporary XML file.

2. Once `<spectrumList>` is reached, do the following for each `<spectrum>` under it while keeping track of the number of matches:

   - Go through the spectrum recursively and populate the **SpectrumType**.[12]

[11] More information about JAXB may be found at:
http://java.sun.com/xml/jaxb
For an overview, the relevant Wikipedia article is also useful:
http://en.wikipedia.org/wiki/JAXB

[12] Recursively here means that all elements in the hierarchy under a spectrum should be read and represented by an object of the appropriate Java class which will be made accessible through the **SpectrumType** object.

- Append matching spectra to the temporary file (see below)

3. For any encountered elements not under `spectrumList` (in practice `</spectrumList>` and `</mzData>`), append them to the temporary file.

4. Finally, go over the temporary file again and rewrite it to the final file, making sure to fill in the attribute *count* under `spectrumList`.

The reason for performing the reduction in this manner has to do with the options for handling XML data in Java. There are two main models for working with XML:

1. The Document Object Model (DOM), in which an entire XML document tree is loaded into memory. This enables arbitrary parts of the document to be accessible whenever they may be needed, but may be inefficient for large documents.

2. Streaming model, in contrast, uses some form of "cursor" which reads one XML element at a time from the start to the end of the document. At any given time, only the element currently pointed to by the cursor may be accessed and the cursor may only be moved foreward. This provides a very efficient way to scan through large XML documents, but limits the options for navigating them.

In addition to these, JAXB provides the option of marshalling and demarshalling Java objects. Although this is quite different from DOM, it suffers from the same issues in the context of QALM: Loading an MS/MS file potentially several gigabytes in size into memory would require large amounts of memory, and would most likely be a prohibitively slow process. Marshalling smaller objects into XML on the other hand, would not pose any problems, and would simplify a part of the process.

QALM therefore uses a combination of the streaming model and JAXB marshalling: The Java Stream Reader API for XML (StAX) provides a quick way to scan through the original XML document, and is used in QALM to quickly generate java-objects representing each of the `spectrum`-elements (**SpectrumType**, and output directly the elements that are not part of spectra (in essence everything before and after the spectrum list). Each of the **SpectrumType**-objects are checked to see if it matches any of the report peaks, and appended to the temporary file using JAXB. This process is explained in more detail in the following section.

### Checking for matches and adding spectra

Checking if a spectrum matches any report peaks is done by the following algorithm (simplified / abstracted), which uses an array-list of **SpectrumType**-objects to keep track of the spectra to output:

Data:
**spec**: The new spectrum to check (either MS or MS/MS)
**rt**: Retention time from **spec**
**msl**: MS-level from **spec** (MS = 1, MS/MS = 2)
**sList**: An array list containing matched spectra

Procedure; For each encountered spectrum **spec**:

1. If(**msl** = 1):
   if (**sList**.length > 1){ marshallSpectrumList() }
   Clear **sList**
   **m1MatchList** = getMatchingM1Peaks( **spec**, **rt** )
   if(**m1MatchList**.length > 0){ sList.add(**spec**, **rt**) }
2. Else, if(**msl** = 2):
   if(**sList**.length = 0){ **rt** mismatch; do nothing! }
   else:
   **mz** = **spec**.getMzValue()
   **preId** = **spec**.getPrecursorId()
   **m2MatchList** = getMatchingM2Peaks(**spec**, **rt**, **mz**, **preId**)
   if(**m2MatchList.length** > 0):
   **sList**.add(**spec**)
   saveDataToDB(**m2MatchList**)

The method *getMatchingM1Peaks()* scans the list of peaks selected from the peak report (from XCMS) and returns any peaks that match the retention time of the current spectrum. *getMatchingM2Peaks()* does the same for MS/MS-peaks; it updates and returns a list of matches between spectra and peaks (these are stored in the database by the call in the last line).

*marshallSpectrumList()* appends all spectra in **sList** to the new, temporary XML file. Notice that the sets of spectra in the **sList** is added to the new file if and only if a new MS-spectrum (level 1) is encountered, and there are two or more previous spectra in the list. The reasoning behind this is as follows:

- The targets of the search are level-2 spectra and their precursors under the conditions that:
  - The level-2 spectra have **mz**-values matching a report peak
  - The precursor has **rt** matching that of the same report peak.
- All level-2 spectra have a level-1 precursor spectrum, so the first spectrum in the list will always be a level-1 spectrum.
- If level-2 spectra occur in the list, it must therefore have length greater then 1, and conversely, if there are more than 1 elements in the list, all except the first must be level-2 spectra.

- All spectra in the list must be matches; a level-1 spectrum must have an **rt** value matching that of one or more report peaks, while level-2 spectra following it must have **mz**-values matching those of the same report peaks.
- The last instance in a set of level-2 spectra with a common precursor must have been included once a new level-1 spectrum is encountered, since all MS/MS spectra in the original XML-file follow directly after their precursor (MS) spectrum.

This ensures that all matching MS MS/spectra are included, and that they will be preceeded by their respective precursor spectra. Two other points worth noting to better understand this algorithm:

- It allows small chunks of output to be made many times throughout a reduction.
- If the **rt** for a level-1 spectrum is a mismatch, **sList** will be empty, so all level-2 spectra encountered are ignored until a new, matching level-1 spectrum has been encountered.

#### Binary data and the Apache Commons Codec

The information stored under the `data`-elements (see fig.6.14) is stored in base 64 binary. In order to unmarshall this into a Java object and Marshall it back to XML later, the Commons Codec from Apache Commons[13] is used to decode and encode the data (The library is in the Java JAR-file `commons-codec-1.4.jar` in the /lib folder under QALM).

### 6.7.2 Mining Mascot: Retrieving data from .dat-files

To retrieve the data needed from Mascot result files, QALM makes use of Mascot Parser[14], a library developed by Matrix Science (the creators of Mascot) specifically for the purpose of accessing such files. Mascot Parser is "*(...) a package that provides an Application Programmer Interface (API) to the Mascot result and configuration files.*" [MascotParser].

Use of Matrix Parser relies on the availability of a Java JAR-file (msparser.jar) and a software library (libmsparserj.so), both of which are found in the lib-catalog under the QALM root directory.

Note that on 64-bit computers, a different version of libmsparserj.so is needed. This is included in QALM, but needs to be renamed; please refer to the installation instructions in Apendix A for details.

[13] http://commons.apache.org/codec/

[14] Mascot Parser is available from:
http://www.matrixscience.com/msparser_download.html

### Usage

The following is a simplified example of how Matrix Parser may be used:

```
// Load file and prepare for reduction:
String filePath = "/path/to/mascot-file.dat";
ms_mascotresfile file = new ms_mascotresfile(filePath, 0, "");

// Loop over queries in file:
for (int i = 1; i <= file.getNumQueries(); i++) {
   // Instantiate a TO-object to store Mascot Query information:
   query = new MascotQueryTO();

   // Get input query and title string:
   inputQuery = new ms_inputquery(file, i);
   String titleString = inputQuery.getStringTitle(true);

   // Get the start and stop-positions of the value of spectrum id
   //  in the title string (between "spectrumId=" and " ")
   int specIdStart = titleString.indexOf("spectrumId=") + 11;
   int specIdStop = titleString.indexOf(" ", specIdStart);

   // Fetch and store the value as an integer in the query object:
   query.setSpectrumId(Integer.parseInt(
       titleString.substring(spectrumIdPos, spectrumIdEndPos)));

   // For each query i, fetch the value of the mass-field from
   // the summary section, and store it in the query-object:
   query.setQmass(file.getSectionValueDouble(
           ms_mascotresfile.SEC_SUMMARY, "qmass" + i));
}
```

Similar code is used to retrieve all the relevant information from mascot files. In broad terms, the steps in the overall algorithm involve doing the following for each query in the .dat-file:

1. Create a **MascotQueryTO**-object
2. Set the variable "analysisID" in that object to the id of the current analysis.
3. Populate that object with query-level values (id, mass, score, etc)
4. Insert the information in the MascotQueryTO into the database
5. Create an **ms_peptidesummary** (**PS**) to access the peptide list
6. Create an **ms_proteinsummary** (**RS**) to acces protein information
7. Iterate over the peptides using **PS**, and for each of them:
   (a) Create a **matchingPeptideTO**-object to represent it
   (b) Populate that object with data fetched from the peptide

(c) If the peptide scores above the threshold[15]:

i. Insert it into the database

ii. Get the list of proteins in which the peptide ocurs

iii. For each such protein:
- Create a **matchingProteinTO**-object
- Fetch and store the accession (id) of the protein in it
- Fetch it's description using the **RS**-object
- Insert the protein into the database

Whenever something is inserted into the database throughout this process, a reference to the element it belongs to is always included (for instance, a mascot query refers to the id of the analysis it is included under, while a peptide fetched from that query refers to the id of the representation of that query in the database, etc). The result is a set of records in the database that will show the relationships between the various pieces of data from each .dat-file, and where that data will correspond to the data in the MS/MS-file that was used to generate the .dat-file through the Mascot search.

The relevant tables in the database were described in section 6.5.3 and illustrated in Figure 6.10.

### 6.7.3 R Scripts And How To Call Them

The R-procedures needed in the context of QALM were described in chapter 3.3. This section describes how scripts to execute such procedures are generated and called from the main application.

#### Scripts

The use of R under QALM may be divided into two main steps and several substeps, each of which involves executing a separate R-script. Preprocessing is performed once for a collection, while analyses may be performed several times for each collection; usually once for each pair-combination of situations under the collection. The set of scripts and results for each of the main steps are:

1. Preprocessing:

   **prepareFileList.R** supplies the files to include in the group.

   **xcmsSet.R** creates the xcmsSet-object from the files.

   **group.R** groups the peaks in the xcmsSet-object.

   **retcor.R** performs retention time correction.

   **fillPeaks.R** fills in missing peaks.

2. Analysis / peak-reports:

[15]The threshold is set in the file QALMSettings.properties. See section 6.7.5

**diffreport.R** generates peak-reports for two specified situations under a specified collection.

In addition to these scripts, two special scripts are used: **RPreparations.R**, which loads the XCMS package and the other scripts, and **RSourceFile.R**, which is generated anew before each main step and is responsible for calling all the other scripts in succession.

Any environment-settings needed in R may be added to **RPreparations.R**, as this script will always run before the others. **RSourceFile.R** on the other hand, should not be changed, as it will be overwritten the next time QALM makes any calls to R. The rest of the scripts may be changed if necessary, for instance to set different arguments to a command under certain conditions. This solution allows for some flexibility in the way R and XCMS is called; The actual R-calls may be changed a great deal without having to make any changes to the Java source code.

Although the generation of peak-reports (step 2) depends on the existance of data resulting from preprocessing, the calls to each of the two steps run independantly of each other. The way they are called is a illustrated in Figure 6.16, which displays the first three scripts needed for the preprocessing-step.

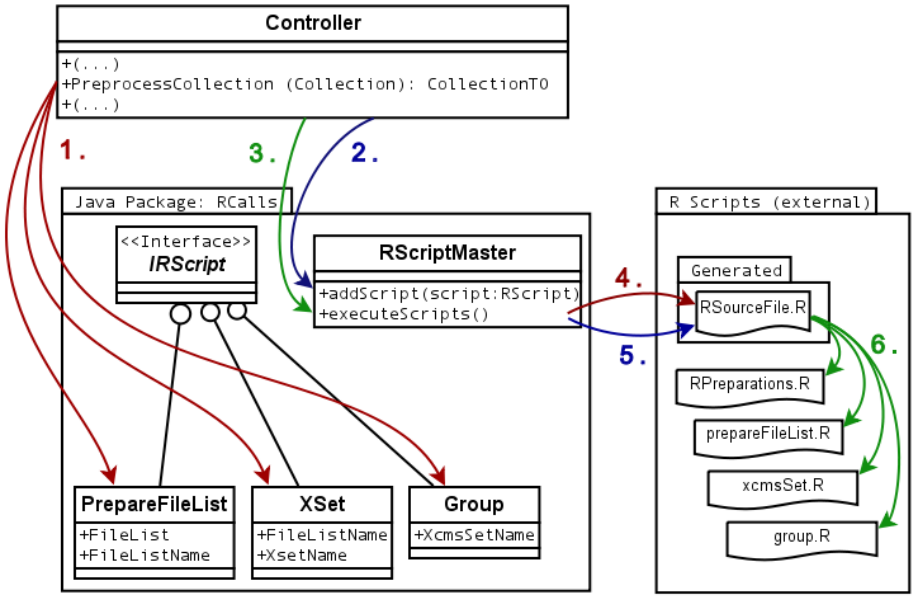



Figure 6.16: Overview of the procedure for producing and calling R-scripts. Elements to the left represent Java-objects, while "R Scripts" and "Generated" toward the right represent directories containing individual R-script files. See the description in the text.

### Calling R

The task of executing a set of R-scripts (either for preprocessing or analysis) involves the following calls (corresponding to the numbers in Figure 6.16):

1. The Controller creates a set of Java objects to represent each of the required R-commands. These objects are implementations of the IRScript interface[16], and correspond to specific R-Script files.

2. Each of the RScript-objects are added to a list in an **RScriptMaster**-object in the order they should be executed in using the method *addScript()*.

3. When all the script-objects have been added, the controller calls the method *executeScripts()* in **RScriptMaster**.

4. The **RScriptMaster** deletes any existing file `RSourceFile`, then creates a new, empty file, and places a call to execute `RPreparations.R` in the first line. It then iterates over the **RScript**-objects received from the controller, and adds one line to `RSourceFile` for each object.

5. Finally, **RScriptMaster** calls R through the operating system and requests it to run the script `RSourceFile.R`.

6. When `RSourceFile.R` is executed, it calls `RPreparations.R` and the other scripts in succession.

To recapitulate: The main script for initiating either prepcoessing or analysis is generated by QALM, then run through a call to the the operating system. In the case of an analysis, this would only include one call to the script `diffreport.R`. For preprocessing however, a number of scripts would be called, and two of them (`group.R` and `retcor.R`) may be called several times with different arguments each time.

The output from R is usually sent to the command line from where the scripts were executed. In the case of QALM, this output is caught and redirected to the Processing Panel using the **Background processor**, as described in the following section.

### 6.7.4 The Processing Panel And BackgroundProcessor

Some of the tasks performed by QALM take significantly longer time to complete than others. Such operations may also produce output that could be usefull or informative throughout the process. If such operations were started without any thread-handling the result would be a "frozen" GUI, with no possibility of outputting any information or results. The user would simply have to wait until the operation either completed or failed and an error message was shown before being able to continue.

To avoid such problems and to enable relevant output to be displayed within the application, QALM runs certain tasks in a separate thread, and provides the option of outputting data from tasks run in that thread in the "processing panel".

The main class in the GUI-component of QALM is **QALMGui**. Within this class, there is an inner class called **BackgroundProcessor** which is responsible for the thread-management. This class extends the abstract class

[16] That is, they are specializations of the type of object, "RScript". Simplified, one might say that **Xset** and **Group** are specific types of **RScript**'s, similarly to how **cat** and **dog** are specific types of **mamal**'s. The point of this is to ensure that the RScriptMaster accepts all objects that are valid **RScript**'s, but no objects that are not.

**javax.swing.swing.SwingWorker** to manage threads. The constructor in **BackgroundProcessor** takes two arguments:

- `operation`: An instance of **BackgroundProcessEnum** that identifies the operation to run. This lets all background-processes to run through the same class. The alternative would be to create a separate class for each background-operation.

- `param`: An Object-array for passing arguments to the method to call. Since the required type is **Object**, all Java objects can be passed here.

When **BackgroundProcessor** is instantiated, relevant parameters are passed in the array `param`. These are stored as object-variables in the **BackgroundProcessor**-instance, and when the method *execute()* is called on that object, they are used to run the specified operation in the a separate thread.

In the case of preprocessing for instance, the following method in the GUI is called. It instantiates the **BackgroundProcessor**, which in turn calls the **Controller** (in the new, separate thread):

```
/**
* Method for initiating preprocessing in the background:
* (The parameter "c" is the collection to process)
*/
protected void runPreprocessing(CollectionTO c){

   // Store the parameter "C" in the Object-array "paramList":
   Object[] paramList = new Object[]{c};

   // Instantiate the BackgroundProcessor, specifying "PREPROCESS"
   // as the operation, and "paramList" as the arguments:
   BackgroundProcessor preproc = new BackgroundProcessor(
                  BackgroundProcessEnum.PREPROCESS, paramList);

   // Start preprocessing in the background:
   preproc.execute();
}
```

Running the code snippet above causes the following two methods in **BackgroundProcessor** to be called. The first is the constructor, responsible for instantiating the class. It causes the processing-panel to be set to visible, and creates a **PrintStream**-object that refers to it. The parameter passed to **PrintStream** is an object of type **TextAreaOutputStream**; this is a class specially written to forward and output text to a specified **JTextArea**-component[17]. Here, the processing panel text-area (called `txtAreaProcessingOutput`) is passed to the **TextAreaOutputStream**, causing any text that is sent to the **PrintStream** `output` to be forewarded to that text-area.

The constructor also registers which operation is to be performed, stores the relevant arguments from the array `param`, and does any other necessary preparations for the main call that will follow.

```
/**
 * Constructor.
 * @param operation Constants representing the operation to run.
 * @param param Any data the current operation needs to run.
 */
private BackgroundProcessor(
           BackgroundProcessEnum operation, Object[] param){
    super();
    // Change cursor to a "sandglass" or other "busy"-icon:
    setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));

    // Show the processing-panel in the GUI:
    setProcessingView();

    // Create a reference to a PrintStream-object that will
    // send text to the JTextArea in the ProcessingPanel:
    this.output = new PrintStream(
        new TextAreaOutputStream(txtAreaProcessingOutput));

    // Set the type of operation to be called
    this.currentOperation = operation;

    // Any preparations for the specified operation:
    switch(operation){
        case PREPROCESS:
            // Store the collection in the object:
            this.collection = (CollectionTO) param[0];
            break;
        case (...)
    }
}
```

[17]This solution is adapted from a solution proposed by Ranganath Kini in an online discussion in 2006. See the post dated `06 Feb 2006 05:58 GMT` at: *http://www.javakb.com/Uwe/Forum.aspx/java-programmer/24140/Redirecting-System-out-println*. A link is also available from the API for the **TextAreaOutputStream**-class.

The second call to **BackgroundProcessor** in turn calls the method in **Controller** for preprocessing the specified collection.

```
/**
* Run the operation specified in the call to the
* constructor in a background thread.
*/
@Override
public Void doInBackground() {
   try {
      // Run whichever command this instance was instantiated to run:
      switch(this.currentOperation){
         case PREPROCESS:
            guiCtrl.preprocessCollection(collection, output);
            break;
         case (...)
(...)
}
```

Notice the argument `output` in the call to *preprocessCollection()* here. `output` is the variable defined in the previous code snipet (the constructor). It is passed on to the **Controller**-method so that output from the preprocessing can be output to the JTextArea in the processing panel.

Once the process has completed, the method *done()* is automatically run. In this case it updates the relevant information-fields in the GUI, hides the processing panel, and changes the cursor back to it's regular icon.

```
/*
* Executed in event dispatching thread when doInBackground() completes.
*/
@Override
public void done() {
   switch(this.currentOperation){
      case PREPROCESS:
         // Update the GUI-elements relevant to this operation:
         panelManageCollections.updateCollectionLists();
         break;
         (...)
   setProcessingViewOff(); // Hide the processing panel
   setCursor(null); //turn off the "busy" cursor
}
```

The same procedure is used for all processes that require use of a separate thread and the background panel. This includes all the major tasks such as preprocessing, analysis/peak-report generation, file-imports and reductions, and generation of the final reports.

Note that the output itself is left to the methods implementing each of these tasks. The **BackgroundProcessor** simply provides each of them with a **PrintStream**-object that they may choose to make use of.

### 6.7.5 External Settings, Text And Validation

Technical settings and text that is not integrated directly into the GUI are stored in two Java Properties files in the package `no.kjartanleroy.codproject.settings`. These are plain-text files that Java has special methods for reading from with ease and efficiency.

#### Settings

The first file, `QALMSettings.properties`, holds information such as the name of the database, the database driver and the path used to access it, as well as a few path-strings. In addition, there are three settings that affect computations:

- `PeptideMatchingScore`: The minimum score a peptide needs to have to be considered a match when retreiving data from a Mascot .dat-file.
- `MzLim`: The limit for matching m/z-values during MS/MS file reductions.
- `RtLimInSeconds`: The retention time limit for MS/MS file reductions.

The file and the values are loaded when QALM starts, and can be accessed with a call to the method *getInstance()* in the class **Settings**, which is an implementation of the interface **ISettings**.

In addition to this, **ISettings** has three noteworthy methods. If the user enters a name such as *Peter's collection* for a collection, the apostroph may in certain cases cause problems, for instance when inserted into the database. To be safe, this and other unsafe characters should in certain situations be replaced by some other character that is known to be safe. This requires a comparison to a list of characters that are known to be safe. What characters are safe however, may depend on the situation. For instance, an analysis-name, which is used to generate the file-name for a report, may need to conform to stricter rules than text in the description for the same analysis (there may be symbols that are acceptable in the database, but that may not be permitted in file-names). For this reason, there are three separate methods for accessing character lists in **ISettings**:

- *getStrictLegalCharacters()* returns an ArrayList of all alpha-numeric characters (from a-z, A-Z and numbers 0-9), as well as the underscore (_) and dash (-). These are characters and symbols that will be acceptable in most cases.
- *getLaxLegalCharacters()* returns a less strict set of characters. In addition to the previous list, this includes the characters: `æøå, ÆØÅ,.;:@!/?*()[]`.
- *getForbiddenCharacters()* returns an array of characters that should not be used in the database.

These methods are used in various combinations, mainly by the **Adapter**. In some cases, strings that contain illegal characters (or are too long) will cause an exception to be thrown, and an error message explaining the problem to be displayed. In other cases, illegal characters will simply be replaces with an underscore, "_". The above example would then result in a collection called "peter_s_collection".

### Text strings

The other properties file, `ApplicationText.properties`, contains a long list of words, sentences, headings for reports, and more. These are pieces of text that may be output when running QALM, but are not directly integrated into the GUI (unlike titles, text on buttons, etc).

In the same way as for settings, this text is loaded when QALM starts, and accessible through a call to a Java object; in this case, an instance of **TextStrings**, which implements **ITextStrings**, an interface defining a single method: *getText(String key)*.

### Techincal motivation and choices

Storing text and settings in files separate from the source code has several benefits. The primary motivations are to make the them more accesible for editing and to simplify the code by separating content from logic.

"Hiding" the implementation of classes behind an interfaces has one major benefit: The implementations can be changed completely without having any adverse effect on the rest of the application. If for instance one wanted to store all the text and settings in a database, or in several different files in stead of a single properties-file, the implementation could be changed to do that; as long as the interface is still the same, there will be no difference to the rest of QALM.

Both **Settings** and **TextStrings** uses the "**singleton-pattern**": Instead of creating a new instance each time access to one of them is needed, a call to the static class *getInstance()* in the respective class is called. *getInstance()* returns a single instance of the class, through which the needed method may then be called. Next time a similar call is made, the same instance will be returned, even if the call is made from a different component of QALM (a new instance is only created the first time each class is used). The following code, for instance, will return the text "`Scheduling file for deletion:`", which has the key "*SchedulingFileForDeletion*" in the file `ApplicationText.properties`:

```
TextStrings.getInstance().getText("SchedulingFileForDeletion")
```

To ensure that the classes are only used in this way, and never instantiated unnecessarily, they have *private* constructors, and *private static* fields that refer to instances of themselves, once instantiated:

```
class TextStrings implements ITextStrings {

  // Reference to the instance, only reachable
  // through the method getInstance():
  private static TextStrings instance = null;

  // private consstructor, so "new TextStrings()" can
  // only be called from within the class itself:
  private TextStrings() {
     // Do nothing, just avoid instantiation!
  }

  /**
```

```
     * Get singleton.
     * @return
     */
    public static ITextStrings getInstance(){
      // If instance has not been created
      // yet, create it now:
      if(instance == null){
        instance = new TextStrings();
      }

      // Return the instance:
      return instance;
    }
  }
```

### 6.7.6 A Framework For Exceptions And Messages

The class **MessageManager** under the GUI has methods for handling exceptions and displaying other messages and warnings as needed.

#### Messages

There are several different methods for displaying general "popup-messages". Generally, each of these accept various arguments, most of which are forewarded to a Java JOptionPane method, which causes an appropriate dialogue to be displayed. The text displayed in the dialogue will be retrieved through the **ITextStrings** interface described in the previous section. The following example (where `msgMngr` is an instance of **MessageManager**), would cause a dialogue or popup-message containing the text "*Please select a situation to import the files to.*" to be displayed.

```
this.msgMngr.showPopup("NoSituationSelected");
```

#### Exceptions

The file `ApplicationText.properties` contains keys matching the names of most of the exceptions that may be thrown in the context of QALM. Each key has a corresponding message that describes the given problem. If an exception that does not have a corresponding key and message should be thrown, the default message will be shown in stead: "*An unknown error occured.*"

In addition to the default text for each *type* of exception, details for each specific instance in which the exception is thrown can be added using the *message* field in **Exception**. Unless the value of *message* is `null` (empty), the message will be apended to the default text under the default text and the heading "Details".

Figure 6.17 illustrates how an exception might occur and be handled:

Assume some data is to be inserted into the database, but the operation fails because the data contains invalid characters. The operation is started from a calling method in the GUI, and would if successful, be forwarded through the **Controller** and **adapter** to the **DBController**. The **DBController** would insert the data, and the program flow would return to the calling method in

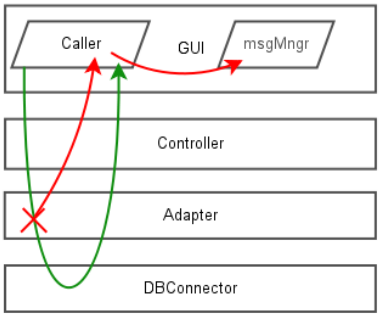



Figure 6.17: Simplified illustration of exception handling.

the GUI, allowing the application to continue running as normal (the green arrow in Figure 6.17). Since illegal characters are noticed by the **Adapter** however, an exception of the type **IllegalCharsInStringException()** would be thrown from there, and it would be propagated back up through each of the other classes, until it reached the calling method in the GUI. Here the exception would be caught in a *try-catch block* (see the code snippet below) and sent to an instance of **MessageManager**, which would identify the exception type and display an appropriate message explaining the problem and listing the illegal characters that were encountered. The code involved would be similar to the following:

```
/**
 * Code in the Adapter. "observedIllegalChars"
 * is a String containing the illegal characters.
 * It will be appended to the general error message
 * under ‘‘Details’’.
 */
throw new IllegalCharsInStringException(
   "The illegal symbols are: " + observedIllegalChars);


/**
 * Code in the calling GUI-class. Only the code
 * between "try{" and "}catch" would be run if there
 * were no errors. When an exception is thrown, it
 * is caught and passed to msgMngr, which handles
 * it appropriately.
 */
try{
    // Code for initiating the process
   (...)
} catch (Exception e) {
    this.msgMngr.handleException(e);
}
```

### 6.7.7 FileTreeNode and the StateManager

**FileTreeNode** is a class that extends **DefaultMutableTreeNode** to add support for file-trees. It adds fields for:

- Keeping track of a file or directory which the **FileTreeNode** will represent.
- Sub-files in it if it represents a directory.
- The id of the situation under which it belongs

Methods in the superclass (i.e. the extended class, **DefaultMutableTreeNode**) are used to add new **FileTreeNodes** below an existing one (which must be a directory for it to be able to accept sub-nodes).

The main use for FileTreeNode is in the class **StateManager**, which keeps track of the directories and files under a project. When a new project is created, **StateManager** creates a Java **DefaultTreeModel**. Objects of this type may contain a hierarchy of **TreeNode**-objects[18]. The **DefaultTreeModel** is serialized and stored to a file (`<project-name>.qpj`) when the project is closed. Next time the project is opened, the file is deserialized and used to display the folders and files under the project (see Figure 5.3).

Although **FileTreeNode** is not displayed in the architecture diagram (Figure 6.3), it may be considered something akin to the TransferObjects (TO-objecs in the diagram), since it is mainly used in the persistence layer, but also occasionally in others (due to it's flexibility, **FileTreeNode** is in some instances used as a form of "wrapper-object" to modify how information such as file-names are displayed in the GUI).

## 6.8 Known issues and potential improvements

QALM is currently a functioning application, yet there are still some issues that should be improved for it to be truly usefull. Time-outs when working with the database have already been mentioned; this section explores a few other potential improvements that might move it from a "proof of concept" to a truly usefull application.

### 6.8.1 Multiple MS/MS files per analysis

QALM currently only supports importation and reduction of a single MS/MS-file for each analysis. While this does supply a "proof of concept" for the importation and reduction of such files, it does still reduce the usability of the application, as there will often be several MS/MS files for each situation.

The choice not to support multiple MS/MS files was made to save time and avoid too much complexity when implementing QALM. The focus was instead on completing a functional application where MS/MS-files could be imported and reduced, and relevant information from them stored in the database. Such functionality is now supported, and by using it in a certain way, it is possible to acquire results from several different MS/MS files related to the same situation:

[18] **TreeNode** is an interface which is implemented by **DefaultMutableTreeNode**, and thus by extention also by **FileTreeNode**.

It is possible to create two identical analyses under the same collection, and import different MS/MS-files into these. The generated peak-reports would be identical, but the data retrieved from the two MS/MS files would differ, and the two final reports generated by QALM would give the required results.

Althogh possible, this is by no means a good replacement for support for multiple files within each analysis. If QALM was to be developed further, such adding such functionality would therefore be a natural first milestone[19].

### 6.8.2 Importing files

The component responsible for importing files into QALM currently uses a simple call to the underlying operating system. The reason for this solution was twofold: Firstly, it was quick and simple to implement; relevant points considering that QALM was primarily a proof of concept, and the copying-component just a small (and easily replacable) part of it.

Secondly, there is currently no simple way to perform file-copying in Java that works acceptably in all cases. Typical examples involve opening byte-streams and transfering a file piece by piece using a byte-buffer. This requires a specific buffer size to be specefied, which might affect the speed of the operation, especially for large files.

In any case, it is reasonable to assume that the operating system will generally perform copy-operations faster than Java will be able to. In stead of spending too much time on optimizing the latter, the former was therefor chosen for QALM.

For the future, it would be advisable to investigate alternatives to the current solution, primarily because it's functionality depends on the underlying OS and may limit cross platform functionality. One viable option may be to make use of the **Path** class under the package `java.nio.file`, which will simplify such operations greatly, and which is to be released with the next version of Java (JDK 1.7[20]) sometime in the near future (QALM currently uses Java 1.6).

### 6.8.3 Support for mzML

QALM currently only supports MS/MS-data in the mzData format, in accordance with specifications of the project it was developed for. Since that data was produced however, mzData has been deprecated due to the existance of another similar format. The two formats have now been combined into the new mzML, and for the future it would be advisable to implement support for this, as recomended by the Proteomics Standards Initiative. For more information on mzML, see: http://psidev.info/index.php?q=node/257

### 6.8.4 Other things

Some other smaller changes that would contribute to the improvemet of QALM might be:

- Adding support for multiple levels of situations (see section 3.3.1).

[19] This would also require adding support for several Mascot result-files.
[20] See: http://java.sun.com/docs/books/tutorial/essential/io/pathClass.html

- Moving settings such as the threshold for .dat-file reduction from the external settings and into the GUI, thereby allowing users to specify them while running the program.
- Adding some more "presentable" formats for the final reports, for instance HTML and PDF.
- Improving support for arguments in the calls to XCMS.
- A solution for storing the data in final reports, thus avoiding having to re-run the calls for generating them unless something has changed.
- Adding support for running QALM under Windows and other platforms.

# Chapter 7

# Final results and concluding remarks

When this thesis was started, very little was known about what the result would look like, or even which main processes and tools would need to be included. Throughout the development process, the components involved have grown from just R and XCMS to a host of technologies including various Java components, a database, several software libraries for traversing and analyzing files in different formats and more. A solution for organizing data in hierarchies of projects, collections and analyses has been designed, and this has been connected to third party applications for analysis and data storage. Algorithms have been developed for traversing various files and extracting and storing relevant information, while complexity has been reduced by removing irrelevant data.

As it stands today, QALM is primarily an experimental application. Though it does work as intended and may well be usefull as a tool to automate analyses, it is important to remember that it was developed as a proof of concept, and that it is not yet a stable, release-ready application. What QALM does provide is a useful example of how files and data from mass spectrometry experiments may be handled, and how the various tasks performed on such data may be integrated and executed from a single consistent framework. A future application, based either directly or indirectly on QALM, is likely to benefit from the experience gained throughout the work done in this thesis.

## 7.1 Visions for the future

Beyond adding support for the functionality described at the end of chapter six and possibly revising QALM to make it more robust, there are several posibilities for expansions in the future:

1. Closer Mascot-integration: If either QALM (or some other application with an interface that QALM could connect to) were to run on the same computer as a Mascot-server, then it might be possible to send commands directly to Mascot. If possible, this would remove the need for exporting MS/MS-files and requiring the user to perform the Mascot-search manually, and let the whole process be controlled directly through QALM.

2. Enabling use of an external DBMS, possibly one available over a network. Such a system might also be expanded to support different users with different sets of data, sharing of that data.

3. A client-server solution: When an LC-MS-MS/MS experiment is run, the resulting data is currently stored on one system, and needs to be transfered and imported into a system running QALM to run the analyses described in this thesis. If QALM could instead be divided into a "server" component where most of the functionality would run, and a "client" component which could issue commands to the server-component, and the server-component could be granted access to the experimental data, this would further improve and simplify the analysis process. Combined with the previous point, this might make up a usefull framework which would not only allows different researchers to carry out their analyses quickly and effectively on a common system, but also to share results with each other.

   In addition to technical challanges, such an approach would likely raise questions related to security and how experimental data is accessed, among other things. Barriers might exist that would render such a solution difficult or unrealistic, but this will not be known for sure unless the concept is explored more thoroughly.

# Bibliography

[Eidhammer08] Ingvar Eidhammer et al. *Computational Methods for Mass Spectrometry Proteomics*. Wiley-Interscience, first edition, 2008.

[Eidhammer09] Ingvar Eidhammer. Protein Quantication by Mass Spectrometry - A Compendium for the Course INF389. Referenced with permission from the author, 2009.

[Eidhammer10] Ingvar Eidhammer. Finding the proteins of the differentially abundant peaks. Description of functionality of QALM (included in apendix), 2010.

[Leroy09] Kjartan Lerøy. Beskrivelse av masteroppgåve i Bioinformatikk. Priliminary description of Masters thesis, 2009.

[MascotFAQ] Matrix Science Limited, London. Mascot Search Results FAQ. PDF document / presentation from the developers of Mascot, available online at:
http://www.matrixscience.com/pdf/2005WKSHP4.pdf, 2005.

[MascotParser] Matrix Science Limited, London. Matrix Parser Download. Webpage with lisence and download information for the Mascot Parser package:
http://www.matrixscience.com/msparser_download.html, 2010.

[NelsonCox08] David L Nelson and Michael M. Cox. *Lehninger Principles of biochemistry*. W.H. Freeman and company, New York, fifth edition, 2008.

[QALM09] Ingvar Eidhammer. QALM - Quantitive Analysis of LC-MS/MS-data. Description of functionality of QALM (included in apendix), 2009.

[RIntro] The R Development Core Team. An Introduction to R / Notes on R: A Programming Environment for Data Analysis and Graphics, version 2.11.0. Manual (Dated 2010-04-02) available from the R-project website: http://www.r-project.org, 2010.

[RLangDef] The R Development Core Team. R Language Definition, version 2.11.0. Manual (Draft, dated 2010-04-02) available from the R-project website: http://www.r-project.org, 2010.

[Smith09] Colin A. Smith. LC/MS Preprocessing and Analysis with xcms. Manual for XCMS, dated May 8. 2009. Available online at: http://www.bioconductor.org/packages/release/bioc/html/xcms.html, 2009.

[XCMSAPI] Colin A. Smith et al. XCMS API Reference manual. Manual for XCMS, dated September 2., 2009. Available online at: http://www.bioconductor.org/packages/release/bioc/html/xcms.html, 2009.

[mzMemo] Jr. Randall K. Julian and Puneet Souda. Mass Spectrometry Data Representation (mzData) 1.05. Proposed recomendation / Memo from the Proteomics Standards Initiative (PSI), dated 1. August. Available online at:
http://psidev.info/files/mzData_1.05_spec.doc, 2006.

# Apendix A: Obtaining and running QALM

# Obtaining and running QALM

QALM currently runs on the Linux platform. It has been developed and tested on a 32-bit system running OpenSuse 11.2 and on a 64-bit system running Fedora. In addition, an early version of QALM has run on Ubuntu. This chapter describes how to obtain a copy of QALM, and the requirements for installing and running it on a platform similar to these.

## Requirements

QALM integrates several different applications and technologies, and it is generally assumed that the following will already be installed in the host system QALM is to run on. Details regarding the installation of these applications may vary depending on the platform, but relevant installation guides for most systems are available from the vendors.

- **Java, version 6** or newer. This can be obtained from: http://www.java.com/en/download/index.jsp
- **The R environment** is available from the CRAN-network:
  - http://cran.r-project.org/mirrors.html
  - The University of Bergen hosts a mirror for CRAN. Linux packages are available for some of the major Linux distributions from: http://cran.ii.uib.no/bin/linux/
- **The XCMS package**. As this is a more specialized package, some installation details are described below.

### Installing XCMS

The XCMS package for R can be downloaded from the bioconductor.org website. If the required libraries are available, the package can be installed from within R. To do this, start R by opening a terminal and entering "R". When the program is started, enter the following:

```
source("http://bioconductor.org/biocLite.R")
biocLite("xcms")
```

If this succeeds, XCMS is installed and QALM should be able to make use of it. If an error-message results however, one or more libraries may be missing.

XCMS requires the libraries NetCDF and zlib-devel, which may be acquired as follows:

### NetCDF

The NetCDF-library may be downloaded from (in one line):

```
http://www.unidata.ucar.edu/downloads/
netcdf/ftp/netcdf-4.0.1.tar.gz
```

Download the package and save it in a directory, then enter the following commands in a console to install it under the directory */usr/local*:

```
tar -xzf netcdf-4.0.1.tar.gz
cd netcdf-4.0.1
./configure --prefix=/usr/local
make
sudo make install
```

### zlib-devel

The best way to obtain the `zlib-devel` package may depend on the platrom running. During development under Open Suse 11.2, the following package, available from the Open Suse repositories has been used (in one line). The package was installed and configured automatically by the package manager.

```
http://download.opensuse.org/repositories/openSUSE:/
11.2/standard/i586/zlib-devel-1.2.3-140.2.i586.rpm
```

Note: On a 64-bit system, the following package should be used instead:

```
http://download.opensuse.org/repositories/openSUSE:/
11.2/standard/x86_64/zlib-devel-1.2.3-140.2.x86_64.rpm
```

### Sample data and examples for XCMS

For sample data and examples in XCMS, the following packages are also recomended. They may be installed directly from R, just like XCMS itself:

```
source("http://bioconductor.org/biocLite.R")
biocLite("faahKO")

source("http://bioconductor.org/biocLite.R")
biocLite("multtest")
```

## Installing QALM

QALM itself can be downloaded in the form of a zip-archive from:

```
http://qalm.googlecode.com/files/QALM.zip
```

The archive should be stored in a directory and unzipped there. As long as the requirements specified above are met, the QALM can be started by running the java JAR-file `QALM.jar`, found under the new directory, *QALM*.

In some cases the file can be started simply by clicking the icon marked QALM.jar. Otherwise, it can be called from a terminal with the command:

```
java -jar QALM.jar
```

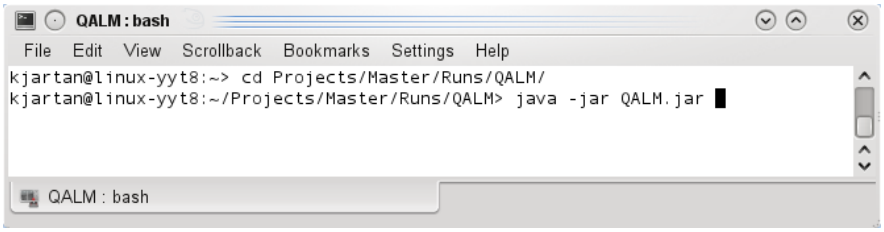



Figure 7.1: Starting QALM from a terminal (the archive has been "unzipped" in the directory *Runs*).

### Usefull notes and troubleshooting

- The library used for Mascot file reduction is available in two versions, one for 32-bit systems and one for 64-bit systems. Both libraries are included with QALM (under the directory *lib*), but only the file named `libmsparserj.so` will be used. **For 64-bit systems, it is therefore necessary to rename the file `libmsparserj.so` to something else (e.g. `libmsparserj_32.so`), and to rename the file `libmsparserj_64.so` to `libmsparserj.so`.**

- When started for the first time, QALM will create the database and the required tables and store it in a directory *QalmDB* under the main directory *database*. If the directory *QalmDB* is removed or renamed, QALM will create a new, empty database with the original name the next time it is started.

## NetBeans project and source code

QALM has been developed using the NetBeans IDE[1]. A NetBeans project which includes the source code for QALM is available from a Subversion (SVN) repository under Google Code. The command needed to access and check out the project is:

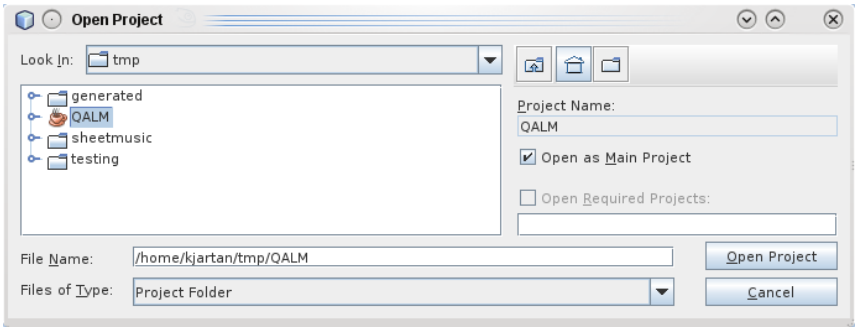
```
svn checkout http://qalm.googlecode.com/svn/trunk/ QALM
```

This will place all the files in a directory *QALM* below the the directory in which the command was issued. Once this process has completed, NetBeans

[1] Integrated Development Environment

should be able to find a project named **QALM** in that directory (Clicking *File*, then *Open project* should display the dialogue shown in the figure below).

While it is possible to access the source code of QALM using other IDE's or editors, using NetBeans is highly recomended due to the way it handles information about the GUI[2].




Apendix A, Figure 1: Opening the project in NetBeans
(`svn checkout` was issued from within the directory *tmp*)

### Just the source code

It is also possible to download just the Java source code and R-scripts used in QALM. These are available for download in a zip-archive from:

`http://qalm.googlecode.com/files/QALM-src.zip`

Note however, that this archive does not contain any of the libraries or other things QALM may require, and is only meant to provide a simple way to obtain the source code directly. To get a better overview of the project and to be able to compile and run it, checking out the project from the repository is highly recomended.

## Javadoc API

API[3] documentation for QALM is available in the form of a zip-archive from:

`http://qalm.googlecode.com/files/QALM-javadoc.zip`

Once the archive has been extracted into a directory, the file `index.html` in that directory should be opened in a web browser. The packages under QALM will be displayed in the top left hand frame, while the contents of each package (such as interfaces, classes and enums) will be displayed below. When an item on the left hand side is clicked, the corresponding documentation will be displayed in the main frame to the right.

The documentaion was generated using the Javadoc, and includes comments for all packages, interfaces, classes, enums and methods in QALM.

[2]For more information, see http://wiki.netbeans.org/FaqFormFormFile and other NetBeans documentation.

[3]Application Programming Interface

# Apendix B:
# Beskrivelse av masteroppgåve i bioinformatikk

# Beskrivelse av masteroppgåve i Bioinformatikk

for Kjartan Lerøy, haust 2009 – vår 2010

## *Utgangspunkt*

"R" er ein programpakke for statististisk analyse. XCMS er eit bibliotek / tilleggspakke til R utvikla spesielt for behandling av data frå proteinanalyser, som LC/MS og LC/MS/MS[1]. Når rådata (MS-spektra) produsert ved slike forsøk vert behandla i R og XCMS gjev dette nye datasett som kan brukast til søk I proteindatabaser (t.d. Mascot-databaser) for å identifisere og anslå mengda av bestemte protein.

Sidan R er komandobasert og ikkje har noko grafisk brukargrensesnitt eller direkte kobling til mascot er det noko komplisert og tidkrevande å gjenomføre analyse og proteinidentifikasjon.

I vårt tilfelle er det fisken torsk som skal forskast på og karakteriserast. Det er i den samanhengen ønskeleg med eit system som automatiserar analyse-funksjonaliteten frå R og XCMS mest mogeleg, og som kan bruke desse resultata direkte i søk mot ein **Mascot-database[2].**

## *Målsetning*

Det er tre hovedpunkter som er aktuelle å implementere under dette prosjektet:

1. Eit sett med R-Script som automatiserer behandling / analyser av LC/MS-rådata.
2. Eit grafisk brukergrensesnitt (GUI) som forenklar oppsett og køyring av slike analyser.
3. Ei form for kobling mellom brukergrensesnittet og Mascott, slik at databasesøk vert eit naturleg neste steg som kan gjennomførast innanfor same applikasjon / grensesnitt som analysene.

Som nevnt finnst det inga innebygd grafisk brukargrensesnitt for R og XCMS. Det finns derimot bibliotek[3] som gjer kommunikasjon mellom programmeringsspråket Java og R mogeleg. Per dags dato er desse biblioteka ikkje fullstendige, men det er sansynligvis langt nok utvikla til å kunne nyttast i dette prosjektet for å utvikle eit grafisk brukargrensesnitt i Java.

På grunn av usikkerheten rundt dette vil det i første omgang være naturleg å ha hovedfokus på punkt ein og to, sidan desse vil henge tett saman, og i alle høve være ei forutsetning for å realisere punkt tre.

1 **LC/MS: «**Liquid chromatography» og «mass spectronomy».
2 **Mascot:** Proteindatabase utvikla for å identifisere protein ut frå data frå MS/MS-forsøk. Mascot-databasen som er aktuell i dette prosjektet ligg i Høgteknologisenteret og vert drifta av CBU.
3 **Rjava og org.rosuda.JRI:** Dette var opprinnleg to ulike Java-bibliotek som har vorte slått saman til eit, og som gjer toveis-kommunikasjon mellom R og Java mogeleg.

# Apenxic C: Quantitive Analysis of LC/MS/MS-data

# QALM - Quantitative Analysis of LC-MS/MS-data

Ingvar

October 19, 2009

QALM is a tool for quantitative analysis of proteins. It is based upon the programs XCMS and Mascot. A goal is that *the average user should not be aware of the underlying programs/tools R, XCMS and Mascot.* The basic input to the tool is raw data LC-MS- and LC-MS/MS-spectra.

For using QALM a *project* should be defined. Thus a project consists of the spectra constructed in the project, and all the results, partly and final, produced by QALM.

## 1 Some definitions

**Protein profile** is a list of proteins and their amounts (abundances).

**Situation** is a specification for which we want to determine the profile and compare the profile to other situations. An example is liver of cods living in a polluted environment of a specific pollutation.

**Quantitative comparison** is the process of comparing the profiles from two situations. **Note**: this is a pairwise comparison.

**Collection** is a set of situations. Any two situations in a collection can be compared by the quantitative comparison. When a collection is defined and has been *combined* (Step 4) it can not be changed. This means that if one wants to add a situation to a combined collection, a new collection must be defined. This is due to how the program XCMS works, in that the files of all situations of the collection must be processesed (combined) together in a first phase of XCMS.

## 2 A project

A project consists of:

**A set of situations** Each situation consists of files with raw data LC-MS or LC-MS/MS spectra. Typically a situation has spectra form biological and technical replicates.

**A set of collections** Each collection consists of:

- A set of situations
- The state of the collection, which means steps executed so far.
- Part results calculated so far.
- Final results.
- A set of quantitative comparisions. Each comparison consists of:
  - The (two) situations in the comparision.
  - The state of the comparison.
  - The part and final results so far.

More?

# 3 The steps

Performing a quantitative analysis by QALM consists of executing a set of *steps*. Between some of the steps there exists an ordering, in that one of them must be executed before another. Most of the steps calculate partly (or final) results, and these are stored. This means that after execution of a step the user can take a pause, and then continue at another time. This again means that when QALM is called, one must specify which step to execute, and depending on the step which collection to use. However, when inside QALM one can perform several steps.

The user can perform the following steps:

**Step1** Define a *project*. When starting any of the following steps from "outside", the project must be given.

**Step2** Define a *situation*. The parameters are:

1. Name of the situation
2. The files of the situation
3. Other

**Step3** Define a *collection*. The parameters are:

1. Name of the collection
2. The situations of the collection
3. Other

**Step4** Perform a *combination*. A combination is processing each of the raw data files of a collection for peak detection, matching peaks and retention time aligning. The XCMS operations are: *xcmsSet, group, retcor, fillPeaks*. The parameters are:

1. Name of the collection
2. Other

**It might be that this step should be divided into two or several steps**.

**Step5** Define a *quantitative comparison*. The parameters are:

1. Name of the collection
2. Name of the quantitative comparison
3. Name of the (two) situations
4. Other

**Step6** Perform a *peak comparison*. This means comparing the matching peaks of the two situations of a quantitative comparison. This is performed by the XCMS operation *reporttab*. The parameters are:

1. Name of the collection
2. Name of the quantitative comparison
3. Other

**Step7** Identify *interesting peaks* from the peak comparison. These are peaks with different amounts (intensities). The parameters are:

1. Name of the collection
2. Name of the quantitative comparison
3. Other

**Step8** Identify differentially abundant *proteins*. This is done by searching by *Mascot* by using LC-MS/MS-spectra of the interesting peaks (and also possible using non-interesting peaks). The mass and retention time of the intereting peaks are used to first locate MS/MS spectra for these peaks. The parameters are:

1. Name of the collection
2. Name of the quantitative comparison
3. Name of the database (filesystem) to search in
4. Other

**Step9** Result assembling. This means collecting information from several quantitative comparisons. The parameters are:

1. Name of the collection
2. Name of the quantitative comparisons
3. Other

**Step10** Result presentation

# Apendix D:
# Finding proteins of the differentially abundant peaks

# Finding the proteins of the differentially abundant peaks

Ingvar

March 4, 2010

This document describes the procedures for finding the proteins of the differentially abundant peaks found by XCMS from two situations, *sit1* and *sit2*. Assume there are totally $m$ MS/MS-files for the two situations.

## 1 Inputs

The inputs are

- The report(s) produced by XCMS: *rtab<-diffreport(object, situation1, situation2, "report", parameters)*
  - *rtab* is an object in R, getting the report about the differentially abundant peaks
  - a directory *report_box* is constructed with plots of the intensities for the differentially abundant peaks
  - a directory *report_eic* is constructed with the Extracted Ion Chromatogram for the differentially abundant peaks
  - a file *report.tsv* is constructed, showing the *differentially abundant peaks* (probably same as *rtab*)
- MS/MS-files (.xml) denoted as $M2$ files, from the situations. We also denote them as

Denote the differentially abundant peaks as $D = \{d_i\}, i = 1, ..., n$. The content for each peak is:

- identification by mass-to-charge $Mz$, and retention time $T$
- contain data about fold, P-value, Mz and T for each of the MS-files for the two situations

## 2 Overview

Figure 1 shows an overview of the procedure with the different steps. For the first four steps it illustrates the procedure for one MS2-run. Step five will then combine the results for all the MS2-runs for the two situations under consideration.

Considering one M2-file, we can conclude:

- one peak can have several potential M2-spectra (step1)
- two different peaks can share a potential M2-spectrum (step1)
- an M2-spectrum can match several database peptides, in the same or (most often) different proteins (step2-4)

The last step (step5) is then to combine the results from the analysis of each M2-file, and present the recognized proteins with some scores.

# 3 Step1 - File reducing

For performing the most reliable analysis, all the M2-spectra in the situation files should be used. However, in this first analysis we will only use spectra that may correspond to one of the differently abundant peaks $d_i$. We therefore have to extract those spectra for each file. Considering a spectrum in a M2-file with precursor mass $Mz_p$ and retention time $T_p$, the spectrum will be part of the new (reduced) file if

$$\exists d_i : |Mz_p - Mz_{d_i}| < \delta Mz \wedge |T_p - T_{d_i}| < \delta T$$

where $\delta Mz$ and $\delta T$ are the accuracies. Note that $Mz$ and $T$ of the peak ($d_i$) is not the integer value of the identification, but rather *mzmed* and *rtmed*.

The results of this step are (for each M2-run):

- A new (reduced) situation file (RM2-file)
- An 2D array *PS* (for PeakSpectra) with one row for each of the peaks ($d_i$), with the fields
  - name
  - fold
  - tstat
  - p-value
  - mzmed
  - rtmed
  - *count* the number of potential spectra in the M2-file for the peak
  - *spec* the identification of the potential spectra in M2-file, the identification is *spectrum id*

# 4 Step2 - Mascot search

For each of the (reduced) situation files (MR2), a search is performed against a protein sequence data base. A result, *MascotRes2.dat*-file is constructed for each situation file.

## 4.1 Mascot

Mascot is reached from http://mascot.bccs.uib.no/

The results are accessable from http://mascot.bccs.uib.no/mascotdata, where the result files are saved under subdirectories for each day (year, month, day).

To find how to call Mascot from a program, go to mascot.bccs.uib.no. You will see the Welcome message and a little below that there is a topic called *Mascot Daemon*.

Daemon is a software that can connect to the server and submit/queue multiple searches from the user. I.e. if you have many searches to do, you dont need to submit them individually, you can queue all at once at Daemon and he will sort it out.

If you click in Install option there, you will be directed to a nice walkthrough of how install and setup.

# 5 Step3 - Mascot result extraction

Step3 extracts from the MascotRes2 file the data that is going to be used for the further analysis for the protein matches.

The structure of the .dat-file determines how to extract the protein matches. Each M2 spectrum in the situation file (RM2) is denoted a *query* in the MascotRes2 file. The queries are sorted on increasing precursor mass, not mass-to-charge. They are denoted $q_1, q_2, .....$

The MascotRes files are divided into sections. The sections to consider here are:

- The *summary section*, containing for each query:
  - qmass: the mass ($Mr$) of the query (precursor), calculated from the $Mz$ and the charge
  - qexp: $Mz$ and the charge
  - qmatch: number of matched peptides (to the mass $Mr$) in the data base
  - qplughole: threshold for homology
- The *peptides section*, containing the queries in increasing mass order. For each query:
  - for each matching peptide is (the most important):
    - the theoretical mass ($Mr$) of the peptide
    - the mass (delta) difference to the query mass
    - the peptide sequence
    - the matching score
    - the matching protein(s)
- The *"input" section*, containing (mostly from the reduced M2-file) data about the queries in increasing mass order, especially:
  - name="queryi"
  - spectrumId
  - TimeIn Minutes
  - (acqNumber, seems as same as spectrumId)

A new file is constructed, where the extracted data from the three sections are combined in records. There are then one record for each query. The new files are denoted *MascotRed* files.

## 5.1 An extract from an MascotRed file

An example of three records from an MascotRed file is shown below.

| ----------summary section---------- | | | | "input" section | | ---------- peptide section ---------- | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| query | mass | exp | qplughole | spectrumId | Time | pep.nr | peptide mass | delta | sequence | score | protein | position |
| 50 | 1020.500402 | 511.257477,2+ | 19.020084 | 36114 | 6.019 | 1 | 1020.523987 | -0.023585 | LAEEFAVSR | 6.38 | YIDP_ECOLI | 31:39 |
| | | | | | | 2 | 1021.559647 | -1.059245 | WYKIGLDK | 5.67 | CR8AA_BACUK | 248:255 |
| | | | | | | 3 | 1020.524017 | -0.023615 | TGFPTTAQAK | 3.85 | YNR6_YEAST | 503:512 |
| | | | | | | 4 | 1020.571609 | -0.071207 | TGNKVYAIR | 3.18 | DECR_RAT | 107:115 |
| | | | | | | 5 | 1020.535248 | -0.034846 | TGQLVQYGR | 3.18 | RPOB_BACAN | 2:10 |
| | | | | | | | | | RPOB_BACC, RPOB_BACCR, RPOB_BACCZ RPOB_BACHK, | | | |
| | | | | | | | | | RPOB_BACLD, RPOB_BACSU, RPOB_OCEIH 2:10 | | | |
| 51 | 1020.500402 | 511.257477,2+ | 19.692263 | 35943 | 5.991 | 1 | 1021.446503 | -0.946101 | AGDTFPSDGR | 7.02 | GLNE_STRAW | 950:959 |
| | | | | | | 2 | 1020.560379 | -0.059977 | GPSFKASSLK | 6.26 | INP4A_HUMAN | 317:326 |
| | | | | | | | | | INP4A_MOUSE, INP4A_RAT 317:326 | | | |
| | | | | | | 3 | 1021.512711 | -1.012309 | YLSAGPCRR | 5.95 | BRWD1_HUMAN | 25:33 |
| | | | | | | | | | | | BRWD1_MOUSE | 25:33 |
| | | | | | | 4 | 1019.518188 | 0.982214 | ESLRTMQR | 5.85 | UVRC_PSEA6 | 142:149 |
| | | | | | | 5 | 1021.534500 | -1.034098 | ESLRGFWK | 4.82 | POLN_HEVBU | 1507:1514 |
| | | | | | | | | | POLN_HEVME 1505:1512:2, POLN_HEVMY 1507:1514, | | | |
| | | | | | | | | | POLN_HEVPA 1507:1514 | | | |
| | | | | | | 6 | 1019.536819 | 0.963583 | IMLRTQCR | 4.82 | CYOE_BUCBP | 71:78 |
| | | | | | | 7 | 1021.545258 | -1.044856 | MLIRMYPV | 4.82 | YEB4_YEAST | 94:101 |
| | | | | | | 8 | 1021.427353 | -0.926951 | DELEEEMK | 4.59 | SYD_PYRAE | 363:370 |
| | | | | | | 9 | 1021.551773 | -1.051371 | IMIEEFIK | 4.59 | HTPG_THIDN | 418:425 |
| | | | | | | 10 | 1019.467209 | 1.033193 | DTLHSEYR | 4.53 | ALDO3_ARATH | 498:505 |
| 52 | 1020.533544 | 511.274048,2+ | 13.435090 | 35469 | 5.912 | 1 | 1020.435959 | 0.097585 | SEDQEQASK | 3.81 | ANK1_MOUSE | 870:878 |
| | | | | | | 2 | 1019.470596 | 1.062948 | ECLGGVGTER | 0.21 | MTNN_TREPA | 18:27 |

# 6 Step4 - Creating final result file for the M2-run

The input to this step is the *PS* array and the *MascotRed* file. A new file *Result* file is constructed, with one record for each peak ($d_i$). The record for $d_i$ contains the data from the MascotRed file for all the potential spectra found for $d_i$.

Can also be implemented as part of the data base.

## 6.1 Example

Suppose that peak $d_i$ matches spectra with spectrumId 35943 and 36114. Then the record for $d_i$ will contain the two records in MascotRed for those two query spectra (see example above).
△

The way this is done is to scan the PS array, and for each peak ($d_i$) for each potential spectrum (see Step 1) find the spectrumId in the reduced file (MascotRed-file). **It might be that it is here enough to save a pointer to the record in the MascotRed-file.** See how it is to be used in Step 5.

## 6.2 Combining the PS array and MascotRed file

The key for combination is *spectrumId*, which exists in both PS and MascotRed. However, none of them are sorted on this term. For fast combination it may be appropriate to first sort one of them, probably it is best to sort MascotRed. Then the new file *Result* is constructed by scanning *PS*, and then use binary search in MascotRedSorted for each spectrumId.

# 7 Step5 - Final calculations

Before this step, Step1-4 are performed for all M2 runs of the two situations, creating a *Result* file for each of the runs. In this step the final matching (peak, protein) is assessed by combining the resulting information from all the M2 runs.

In this preliminary version we restrict us to the result from only one MS/MS-file. The report should be constructed from the Result file constructed in Step 4. The goal was to calculate a report where the main content is a list of proteins, a calculated fold and a score or probability. The last will be a bit complicated, since I cannot find that there is any P- or E- value corresponding to the peptide matches in the resulting .dat-files from Mascot, only scores. So I think we should try to make a list of *proteins*, where the record for each identified protein should contain:

- the name of the protein
- the peaks ($d_i$), that are found to may have produced a spectrum that has a match to the protein. For each peak:
  - fold
  - P-value
  - the score of the (highest scoring) MS/MS-spectra

# 8 Additional

It should be possible to search in the Cod database at CBU.

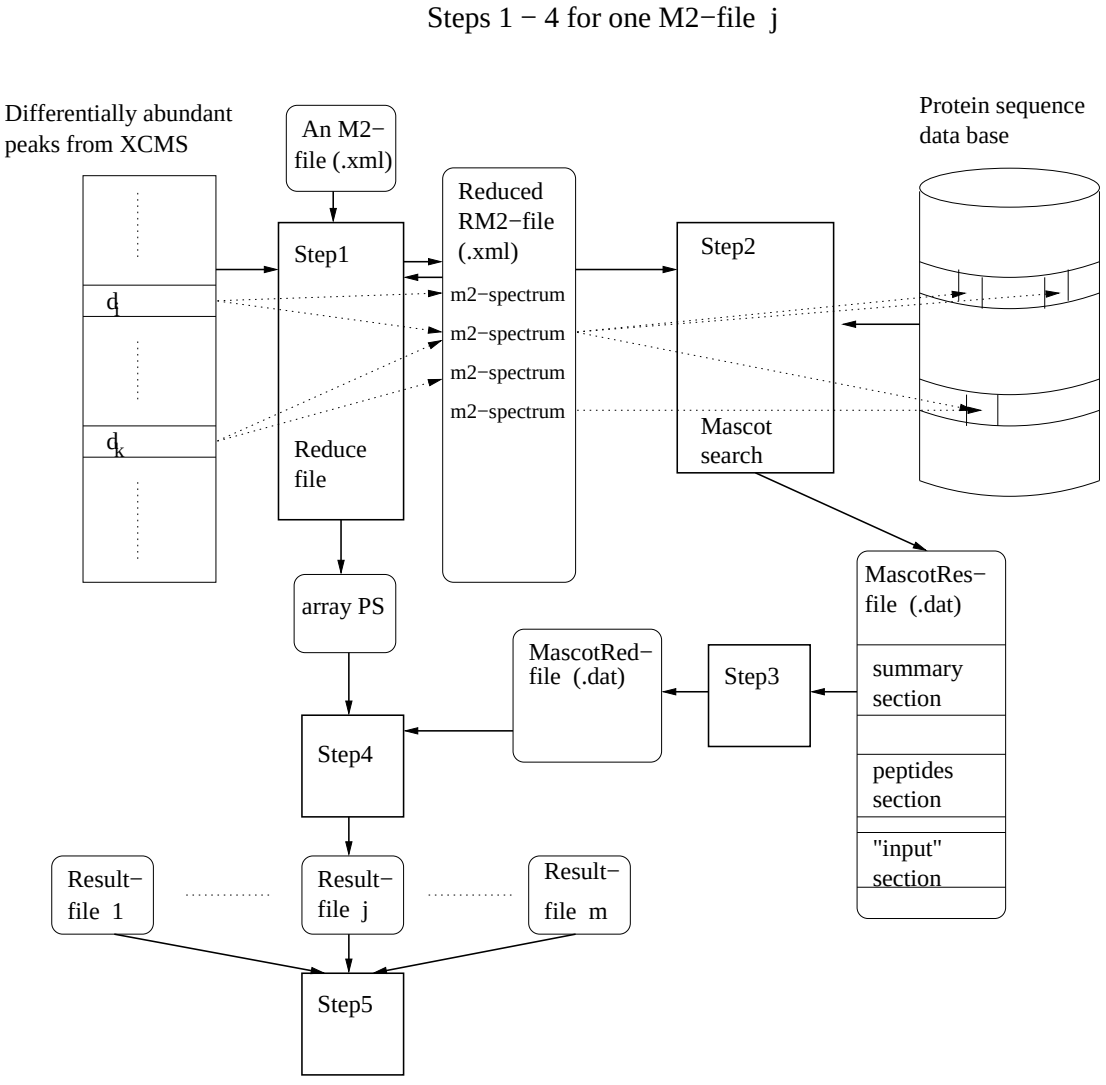



Figure 1: Figure illustrating the first three steps for one m2-file.

# Index