# Automated Analysis of Complex Oligosaccharides

# *via*

# Ion Mobility-Mass Spectrometry

Master Thesis

of

Christian Manz

submitted to the Department of Biology, Chemistry and Pharmacy

of Freie Universität Berlin

28th of February 2017

The work presented here was carried out from 29th of August 2016 to 28th of February 2017 as a collaboration between the research groups of Prof. Dr. Kevin Pagel at the Institute of Chemistry and Biochemistry of Freie Universität Berlin and Dr. Gert von Helden at the Department of Molecular Physics of the Fritz Haber Institute of the Max Planck Society.

First Examiner: Prof. Dr. Kevin Pagel

Second Examiner: Prof. Dr. Beate Paulus

# Acknowledgments

An dieser Stelle möchte ich mich bei Prof. Dr. Kevin Pagel für die Vergabe des interessanten Forschungsthemas und dessen Betreeung bedanken. Zusätzlich möchte ich mich bei Prof. Dr. Beate Paulus für die schnelle Bereitschaft bedanken, diese Arbeit als Zweitprüferin zu bewerten. Außerdem danke ich Dr. Gert von Helden für die Erlaubnis zur Nutzung der Messinstrumente, ohne die eine Anfertigung der Arbeit nicht möglich gewesen wäre.

Ein besonderer Dank geht an alle Mitglieder der Arbeitsgruppen von Dr. Gert von Helden und Prof. Dr. Kevin Pagel, die mir stets mit Rat und Tat zur Seite standen. Insbesondere danke ich Johanna Hofmann für die Einweisung am Synapt und das stets offene Ohr bei Problemen und der positive Rückhalt während der Anfertigung dieser Arbeit. Desweiteren möchte ich mich bei Ph.D. Mateusz Marianski für die Einarbeitung in den Informatikteil dieser Arbeit bedanken, ohne den die Fragestellung nicht hätte bearbeitet werden können. Desweiteren möchte ich mich bei Melanie Göth, Sebastian Malerz und Waldemar Hoffmann für die vielen Diskussionen und dem Korrekturlesen dieser Arbeit bedanken.

Weiterhin möchte ich bei Patrick Schlecht und Christian Teßmar aus dem Elektroniklabors des Fritz Haber Instituts für die Hilfe beim Umbau und der Installation der Messgeräte bedanken.

Bei meiner Familie und insbesondere meiner Freundin möchte ich mich für den sozialen Rückhalt und für die ermutigenden Worte bedanken, ohne die die Anfertigung dieser Arbeit nicht möglich gewesen wäre.

## Statutory Declaration

I herewith confirm that I have prepared the master thesis entitled “Automated Analysis of Complex Oligosaccharides *via* Ion Mobility-Mass Spectrometry” exclusively with the help of the sources and resources specified in this work.

Berlin, 28th of February 2017

_ _ _ _ _ _ _ _ _ _

Christian Manz

# Abstract

Glycosylation is the most common post-translational modification in eukaryotic cells, making glycans ubiquitous in nature. They play a key role in a variety of physiological functions such as serving as recognition sites in molecular recognition or cell-cell communication. Due to their complex regio- and stereochemistry carbohydrate characterisation remains one of the greatest challenges in modern glycoproteomics.

The frequent presence of isomers requires multidimensional analysis techniques to resolve the composition, connectivity and configuration of complex oligosaccharides. Recently, an orthogonal approach combining liquid chromatography (LC) and ion-mobility-mass spectrometry (IM-MS) emerged as a promising tool for carbohydrate analysis. The complementary separation techniques enable the differentiation of isomers based on polarity (LC), molecular size/ shape (IM), and mass-to-charge (MS). However, the combination of methods leads to an enormous increase of multi-layered datasets and processing tools to support such data collections are missing. Furthermore, existing databases lack the required quantity of reference data, therefore preventing the further utilization of this approach as routinely applied analyse tool.

Within this thesis, this problem is addressed by using an instrumental setup of LC IM-MS to enable automatic online measurements in combination with a self-designed **a**lgorithm for **pr**ocessing **i**on mobility **d**ata (Aprid). For this purpose, the well-known oligosaccharides dextran and raffinose were used to characterize the automated setup based on sample consumption, time, robustness and accuracy in comparison to offline measurements. Furthermore, the biologically relevant milk sugar lacto-*N*-hexaose (LNH) as well as the blood group antigen Lewis Y were used to rate the automatic processing. The evaluation revealed the enormous potential of the automated setup. The processing time decreased by multiple orders of magnitude, while sample consumption and accuracy of the results stayed comparable to offline measurements and manual data processing. The so-obtained results demonstrate the potential of this approach as high throughput method to obtain reference data for future databases.

# Content

# 1 Introduction

Carbohydrates represent one of the four major classes of biomolecules along nucleic acids, peptides and lipids. As a product of the photosynthesis, carbohydrates make up most of organic matter on earth and therefore represent an important biological and chemical class of molecules.[1] Beyond their function as an energy source in the metabolism and as a scaffold component in organic structures, carbohydrates can be attached to proteins or lipids on the cell membrane and act as recognition sites in biological signal processes.[2, 3] It is estimated that more than 50 % of proteins and lipids have glycosylation as post-translational modification.[3-5] A well-known example can be found on the surface of the red blood cells as small sugar antennas, serving as blood antigens for molecular recognition.[6, 7] Besides that, carbohydrates are involved in innate and adaptive immune responses and many more biological systems with a wide variety of physiological functions.[8-11]

Despite their diverse biological role in nature, an assignment of functions to specific carbohydrate sequences remains elusive. This is due to their non-template driven formation and especially due to a vast diversity in their architecture caused by their branched and complex stereochemistry.[12, 13] The detailed structural analysis of carbohydrates is very challenging and usually techniques such as liquid chromatography (LC)[14, 15] or nuclear magnetic resonance (NMR) spectroscopy are employed.[16] However, LC methods require a minute to hour timescale and often lack to resolve stereoisomers, and while NMR approaches provide detailed structural information, they require a high amount of sample material, which is often not available. Gas-phase techniques like mass spectrometry (MS) have a high sensitivity and allow the fast investigation of samples in absence of solvent molecules.[17, 18] MS experiments can provide information about the sample composition by measuring the mass-to-charge ratio (*m*/*z*), but lack to distinguish between same-mass isomers.

Multidimensional analysing approaches were proven to be useful for a more defined characterization. As such, fragmentation techniques are often used in tandem MS experiments, which in some cases allow the identification of stereoisomers based on specific fragments.[19, 20] Alternative approaches combine established techniques such as LC and MS.[21-24] Here, individual species in a complex mixture can be separated (LC) and further analysed in the gas phase (MS). A similar separation of isomers can also be performed in the gas phase *via* ion mobility-mass spectrometry (IM-MS).[25-28] In IM-MS, ions are guided by a weak electric field through a cell filled with an inert buffer gas such as helium. During their migration ions with an extended structure collide with the buffer gas more often than compact ions. For that reason, compact ions traverse the cell faster and therefore have shorter drift times. This principle allows a separation of isomers based on their charge, mass, shape and size.[29, 30] The resulting drift time of ions can be converted into a rotationally averaged collision cross-section (CCS), which represents a molecular property.[31] Furthermore, the CCS of fragment ions can be utilized to identify specific carbohydrates motifs. The successful application of IM-MS for carbohydrates was demonstrated in many recent studies.[7, 27, 32]

The complementing two dimensional dataset of *m*/*z* and CCS allows a more reliable and faster identification of carbohydrates when comparing with reference data.[33] These database approaches promise a full scale characterization of oligosaccharides in the near future. Similar to top-down proteomics, which is a method that allows high-throughput MS experiments with an associated database analysing process[34], glycan database approaches could help to significantly increase the analysing speed and quality of carbohydrate identification. The scores of such a glycan database could also be further increased by adding the highly complementary information of other separation techniques as HPLC to the multidimensional dataset. However, processing tools to support such data collections are missing and existing databases lack the required quantity of reference data, therefore preventing the further utilization of this approach as routine analyse tool.[35]

The aim of this work is to establish a high throughput method using an experimental setup that combines HPLC and IM-MS with software supported data processing to enable automated measurements as well as automated data interpretation. The automated setup has the potential to increase the analysing speed and thus allow the accumulation of reference data for future database applications. Furthermore, the standardized procedure increases the reproducibility, therefore leading to more confidence in obtaining structural information of carbohydrates.

# 2 Fundamentals and Methods

## 2.1 Carbohydrates

The term carbohydrates goes back more than 100 years and is literally derived from naturally occurring "hydrates of carbon", expressed in the general chemical formula $C_x(H_2O)_n$. Nowadays the term also involves the derivatives of the originally formula which contain other functional groups and heteroatoms like nitrogen and sulphur. They may be classified based on their degree of polymerization (mono-, di, trisaccharides) or on bigger subclasses like oligo- (3-9 monosaccharides) or polysaccharides (>9 monosaccharides). Today the names glycans, sugars and carbohydrates are used for this class of biomolecules and no sharp distinction in nomenclature exists.[13] Therefore, the different terms will be used interchangeably in this work.

The composition of glycans is determined by the type of connected monosaccharides. The most common monosaccharides are composed of four to seven carbons and therefore grouped as tetroses, pentoses, hexoses or heptoses. In nature, they are present in a ring conformation as the result of an enthalpically favoured cyclization reaction. For hexoses, this reaction includes the binding of the hydroxyl group at position 4 or 5 with the aldehyde at C-1, leading to a five membered ring, called furanose, or a six membered ring, called pyranose, respectively (see Figure 1).[36] Pyranose rings are usually structurally preferred due to less torsional strain. Furthermore, the cyclization reaction leads to the creation of a new stereogenic centre at C-1, called the anomeric centre.[37] Depending on the configuration of the reference atom at C-5, the resulting monosaccharide configuration is either called alpha for a *trans*-relationship or beta for a *cis*-relationship as shown in Figure 1. Monosaccharides are often isomeric and only differ in the stereochemistry of a single hydroxyl group.

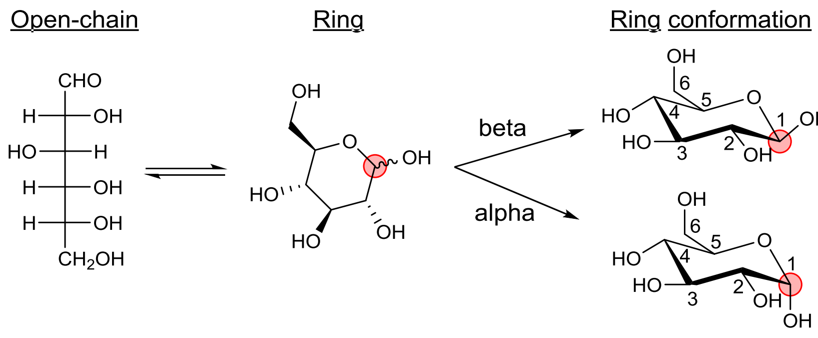


**Figure 1:** Structural relation between the open chain form and the ring conformation of glucose as example for a hexose. The first formula depicts glucose as open chain in a Fischer projection, after a cyclization reaction glucose can occur as closed ring as shown in the middle scheme. C-1 can adopt the two configurations alpha and beta.

The structure of carbohydrates is defined by a building block chemistry, where each carbohydrate unit is connected to a second sugar via a glycosidic bond. These bonds are formed by a condensation reaction between two building blocks and vary in either bond position or anomeric state, which leads to a significant number of different oligosaccharides that differ in either composition, connectivity or configuration (see Figure 2).

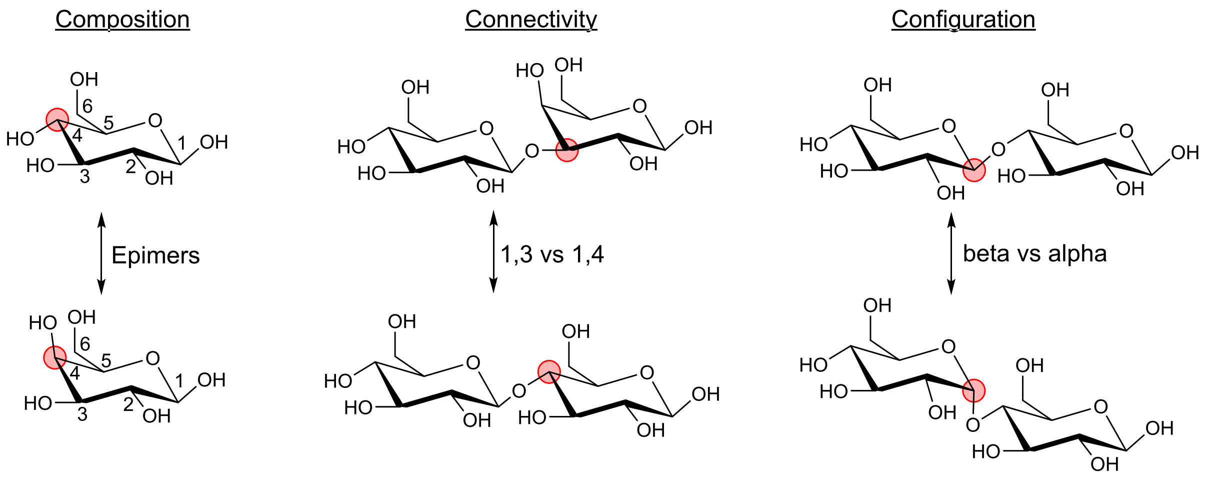


**Figure 2:** Main characteristics that define a carbohydrate structure. Carbohydrates can differ in their composition, e.g. the type of monosaccharides, in the connectivity between two monosaccharides, as well as in the configuration of the anomeric centre resulting in a large variety of isomers.

The complex structure of sugars becomes even more obvious when compared to the linear structure of other biomolecules like peptides. The connection of three amino acids leads to six possible isomers, while the connection of three different hexoses could lead to several thousands of possible isomers.[38, 39]

A simplified visual representation of the complex structure of carbohydrates is provided by the symbol nomenclature for glycans (SNFG) as presented in Figure 3. The SNFG depicts monosaccharides in different geometrical shapes and colours to illustrate same-mass isomers and epimers. The reducing end of a glycan is drawn on the right side and different regio- and stereochemistry is indicated by the type and angle of the lines that connect the monosaccharides.[40]

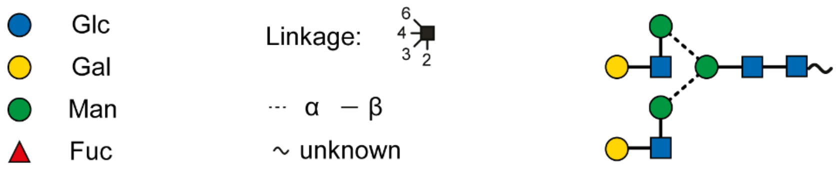


**Figure 3:** SNFG nomenclature of carbohydrates. Sugars are represented by different geometrical shapes and colours, while regio- and stereochemistry is indicated by the type and orientation of the linkage.

Up until now it remains difficult to fully characterize the complex structure of carbohydrates. As mentioned before, an established method to separate and identify components of complex carbohydrate mixtures is LC, but despite the enormous resolving power of current LC systems, the separation of isomers is still not possible in every case and requires a timescale of minutes to hours. Multidimensional tools like sequential MS$^n$ or combined LC-MS/MS methods provide profound characterisation of sugar composition and connectivity, but still lack full separation for configurational isomers.

## 2.2 Ion Mobility-Mass Spectrometry

Recently, IM-MS emerged as a promising addition to established MS methods for carbohydrate analysis. Ion mobility spectrometry (IMS) is a gas-phase separation technique to characterize and separate ionized molecules based on their drift time through a buffer gas. It has a variety of civil and military applications ranging from the detection of explosives at airports[41] to diagnostic functions in hospitals[42], while scientific applications use the method to study and characterize molecules in the gas phase.[43-45]

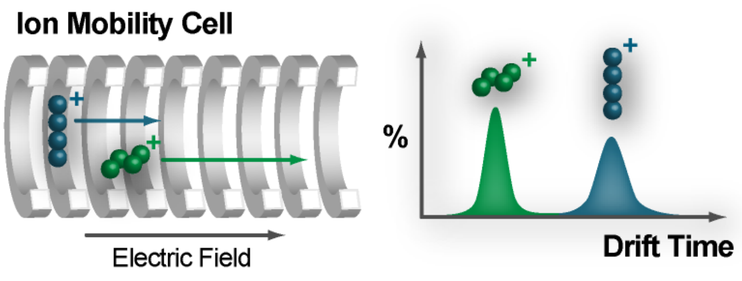


**Figure 4:** Scheme of the separation process in an ion mobility cell. Ions are guided through the drift cell by a weak electric field and collide with a buffer gas. The shape and size of the ions affect the ion´s probability of colliding with the buffer gas and therefore influence the time needed to traverse the cell. Thus isomers with the same m/z but different shape can be separated.

The general principle of IMS is shown in Figure 4. The ionized sample is injected into the drift cell, which contains an inert buffer gas like nitrogen or helium. The ions are guided through the cell by a weak, uniform electric field under constant pressure of 1-15 mbar and on their way undergo collisions with the buffer gas. The amount of collisions is dependent on their shape and influences the time needed to travel through the drift cell. As a consequence, compact ions and ions with high charge states have a higher mobility and travel faster through the cell than extended molecules and low charged molecules.[46, 47] The resulting drift times ($t_D$) of ions are dependent on several instrument parameters, but can be converted into CCSs, which represent a molecular property. In order to obtain more structural information, the IM separation can be easily coupled with MS instrumentation due to their similar mechanics in terms of sample ionization, timescale and detection. The combination of both methods yields a two dimensional measurement method that stands out due to a low detection limit and response time. The commercial availability of different IM-MS instruments increased the distribution of this technique immensely in the last ten years and provides the potential to be applied for routine analysis.[48]

### 2.2.1 Collision Cross-Section

The CCS represents the rotationally averaged area of an ion which is able to interact with a buffer gas and depends on the conformational shape of a molecule without being dependent on instrumental parameters. It is an intrinsic value, which also can be calculated theoretically and allows direct comparisons of theoretical models with experimental results. Furthermore, its universal character allows the implementation in databases such as GlycoMob[49] to simplify the identification of unknown samples. CCSs are determined with the Mason-Schamp-equation (1):[50, 51]

$$CCS = \frac{3ze}{16N}\frac{1}{K_0}\sqrt{\frac{2\pi}{\mu k_B T}} \tag{1}$$

The equation contains the reduced mobility *$K_0$*, the ion charge *$z\cdot e$*, the buffer gas number density *N*, the reduced mass μ of the buffer gas and the ion, the Boltzmann constant *$k_B$* and the effective temperature *T*. The reduced mobility *$K_0$*isderived from the ion´s drift velocity *v* through the drift cell, which can be described as a product of the electric field *E* and the mobility *K* of an ion (eq. 2).

$$v = KE = \frac{L}{t_D} \tag{2}$$

The drift velocity *v* can be determined in experiments by measuring the time $t_D$ required for an ion to travel through a drift cell with the dimension *L*. The reduced mobility $K_0$ of an ion is then normalized with respect to the pressure *P*in the drift cell and temperature *T* (eq. 3).

$$K_0 = \frac{L}{Et_D}\frac{273}{T}\frac{P}{760} \tag{3}$$

The experimental variables to solve for the reduced mobility $K_0$ therefore are the intrinsic drift cell voltage and drift time, while the extrinsic parameters are temperature and pressure.

### 2.2.2 *Application*

IM-MS combines conventional MS methods with an additional separation dimension and therefore has the potential to distinguish between configurational isomers, as well as stereo- and regioisomers. This approach has recently been shown for many oligosaccharides, including the separation and identification of different naturally relevant carbohydrates.[7] For this purpose, the intact precursor as well as fragment ions of several similar Lewis and blood group antigens were studied in order to obtain comparable *m/z* spectra and arrival time distributions (ATDs) as shown in Figure 5. The identification of the trisaccharides is based on the different drift times/CCSs of either precursor ions (see Lewis X vs. blood group H type 2) or fragment ions, which were generated prior to IMS separation and therefore allowed the assignment of defined structural features to the spectra (see Lewis Y).

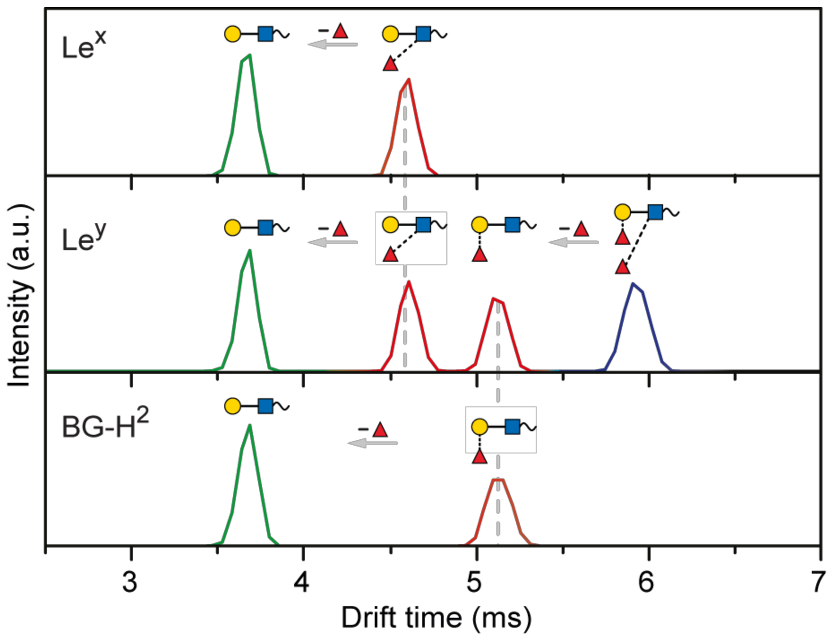


**Figure 5:** IM-MS analysis of the three antigens Lewis X (Le$^X$), Lewis Y (Le$^Y$) and the blood group H type 2 (BG-H$^2$). The drift peak highlighted in blue depicts the precursor ion of Le$^Y$, while the red peaks either represent the first dissociation of a fucose building block by Le$^Y$ or the precursor ions of Le$^x$ and BG-H$^2$. The ATD of Le$^Y$ shows two drift peaks for the first fragment (red), which can be identified by comparing to the precursor drift times of Le$^X$ and BG-H$^2$ (both red). Further fragmentation of all three antigens leads to the identical disaccharide with the exact same drift time (green). The figure is adapted from HOFMANN ET AL.[7]

The experiments were performed on a commercially available Synapt G2-S. The instrumental setup and the general workflow is presented in Figure 6.

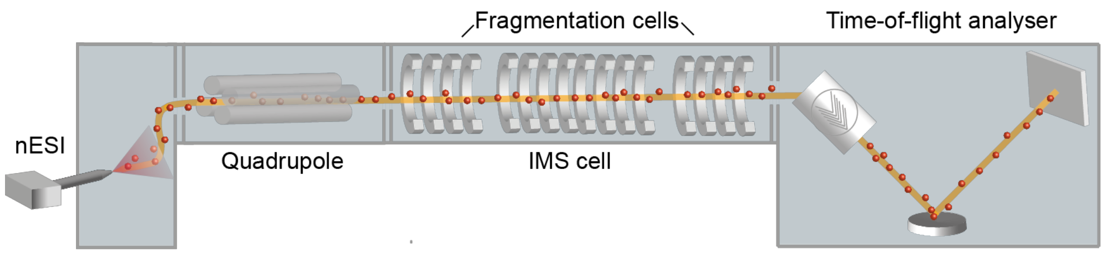


**Figure 6:** Schematic setup of a Waters Synapt G2-S IM-MS instrument.

Ions are generated in a nano-electrospray ionization source (nESI) using a metal-coated glass capillary and gently transferred into the gas phase. After entering the instrument, the ions are guided through an ion guide to maximize transmission and thus increase sensitivity. The following quadrupole enables the selection of specific *m*/*z* ion packages to traverse further to the ion-mobility region. Two collision cells allow fragmentation experiments before or after the drift cell as shown in Figure 7. The mobility separated ions are guided to a time-of-flight (ToF) analyser, where the ions are analysed via their *m*/*z* ratio and afterwards detected to generate a mass spectrum.

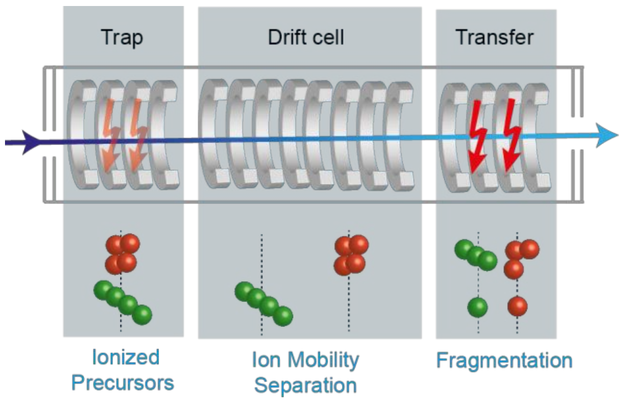


**Figure 7:** Schematic depiction of the IM separation area in the Synapt G2-S. A trap collision cell prior to the drift tube and a transfer collision cell afterwards allow the fragmentation of the ionized precursor before or after IM separation.

The Synapt G2-S is a travelling wave (TW) IM-MS instrument, whereby the ions traverse in a wave-like motion through the drift cell guided by a complex, non-uniform electric field.[52] The inhomogeneous electric field does not allow an accurate mathematical description, therefore the drift time measurements require calibration procedures to determine CCSs.[53] In this thesis, a modified Synapt was used, whereas the original travelling wave cell was replaced by a linear drift cell as described before.[54] This allows the direct conversion of drift times into CCSs, which enables a higher accuracy and faster processing of the measurements.

## 2.3 Automatic Data Interpretation

Despite the rapid development of powerful tools for carbohydrate analysis like LC-MS/MS and IM-MS, the field of glycomics still lags behind the advances that have been made in genomics and proteomics.[35] One major reason for that is the absence of supporting computational methods to store and handle the growing quantity of data.[55, 56] Especially orthogonal approaches of combining analysing techniques with searchable databases and analysis processing tools would increase the sample characterisation in terms of speed and accuracy. While many existing glycan-related databases rely on MS and LC-MS information[57-59], IM-MS derived CCSs have the potential to represent an additional dimension of structural information.[60] CCSs are highly complementary to already existing *m/z* information and furthermore are easily combined with the results of other separation techniques such as HPLC.

However, there is a need for bioinformatic tools to support such data processing and collection, especially software tools for analysing IM data. Data interpretation by hand is time consuming and the increasing quantity of data caused by the combination of multiple analysing techniques, complicates the data processing even more. Several software approaches concerning IM analysis have been released, but most of these tools focus on proteomics and fragment identification [61-63], while automatic processing of glycan data for high throughput analysis lacks behind.

# 3 Aim

Carbohydrates are ubiquitous biological macromolecules with a wide range of biological functions. Due to their complex structure and stereochemistry, they are difficult to characterize using established techniques, therefore carbohydrate analysis remains elusive to date. Recent studies indicate that IM-MS could be an effective tool to characterize complex carbohydrates.

The aim of this thesis is to establish a high-throughput method by using an experimental setup that combines HPLC and IM-MS, with software supported data processing. Several naturally occurring sugars, including Lewis antigens as well as different human milk oligosaccharides and synthetic sugars will be measured using IM-MS and analysed by a self-developed software. The automation of both parts, data acquisition and processing, is essential to build up a database in the future. Furthermore, the development of an automated software could provide a faster data analysis and might increase the reproducibility and quality of the results due to a higher experimental repetition rate and standardized data processing.

# 4 Automated Analysis of Biomolecules

This thesis is structured into three parts, which are illustrated as an interaction diagram of system components in Figure 8. The first part describes the data acquisition including the coupling of a HPLC with an ion mobility-mass spectrometer and the surrounding software as well as a discussion of the HPLC injection parameters. The second part focuses on the data analysis with a self-developed **al**gorithm for **pr**ocessing **i**on-mobility **d**ata (Aprid). Details about the code and modules can be found there. The evaluation of the entire automation setup takes place in the last part, which focuses on evaluating the functionality, speed and accuracy of the implemented algorithm.

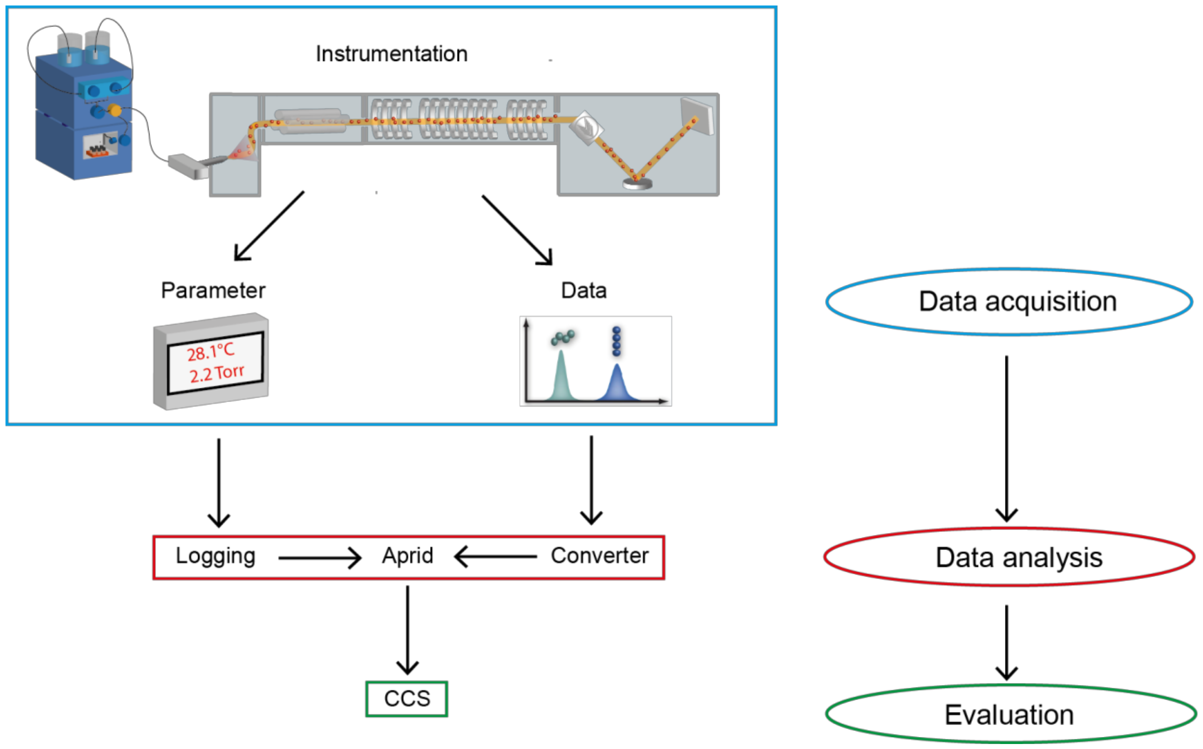


**Figure 8:** Schematic diagram of interactions and workflow between all modules used in this thesis. The coupling of the HPLC to the IM-MS as well as the generated output is summarized in the data acquisition part (highlighted in blue) of this thesis. All software related modules are described in the data analysis part (red), where the used algorithms are shown in more detail. Furthermore, the algorithm for processing ion-mobility data (Aprid) is tested and evaluated in the third part of this thesis (green).

## 4.1 Data Acquisition

### *4.1.1 Parameter Acquisition*

The pressure in the drift cell is regulated by an external flow controller and the temperature is measured with a standard temperature sensor PT100 (platinum measuring resistor) that is directly connected to the drift cell. Intrinsic parameters such as voltages in the drift cell as well as other tuning settings can be regulated with the IM-MS instrument software MassLynx. While intrinsic parameters for each acquisition are automatically recorded in the files *_extern.inf* and *Header.TXT* by MassLynx, external parameters such as pressure and temperature need to be stored manually. Therefore, both external instruments are coupled with the measuring computer to allow digital parameter acquisition. The acquisition is enabled using the self-written *Logscript.py* (see algorithm 4 in the appendix), which calls up both external instruments periodically to read the current values and appends them to a logfile. The script is integrated as system service on the operating system, therefore enabling the script to automatically start as soon as the computer is switched on. This allows a non-stop recording of all relevant external parameters. In addition to temperature and pressure, the respective date and time of the measurement is added to allow a time-dependent search function within the logfile. The parameters are stored in a format similar to the automatically generated MassLynx files to allow comparability and simpler search mechanisms. Thus, the output of the instrument consists of two parts: The automatically generated tuning files *_extern.inf* and *Header.TXT*, and the self-generated *logfile.txt*, which are summed up as parameters, and the raw data files.

In addition to *Logscript.py*, a similar script was designed to enable visual output for the user on the screen. It enables direct control of external instruments like pressure, therefore allowing remote control of all relevant parameters.

### *4.1.2 Data characterisation*

The HPLC consists of a sample manager to automatically uptake samples from a sample plate and a solvent manager to regulate the flow rate and the gradient of the solvents. All parameters of the HPLC as well as of the IM-MS can be controlled using MassLynx with a programmable sample list. It allows to regulate the recording time window as well as the tune settings for both instruments for each run. As the available amount of carbohydrates is often limited, the characterisation of the HPLC sample management, especially for the required concentration and injection volume, is very important. To obtain this information, two commercially available glycans (see Figure 9 and Figure 10) are measured under different conditions. Dextran is a complex, branched homoglycan, consisting of only glucose building blocks with an average molar mass distribution of 1000. The second sample is raffinose, which is a trisaccharide composed of galactose, glucose and fructose. Both glycans have well-characterized IM-MS properties and therefore are used as examples in this thesis for the characterisation of the HPLC as well as for the later evaluation of the automation process. To identify the influence of the HPLC sample management on the quality of the obtained data, dextran and raffinose were measured with several concentrations and different injection volumes. Figure 9 shows the total ion current as a function of the running time of a HPLC run with dextran.

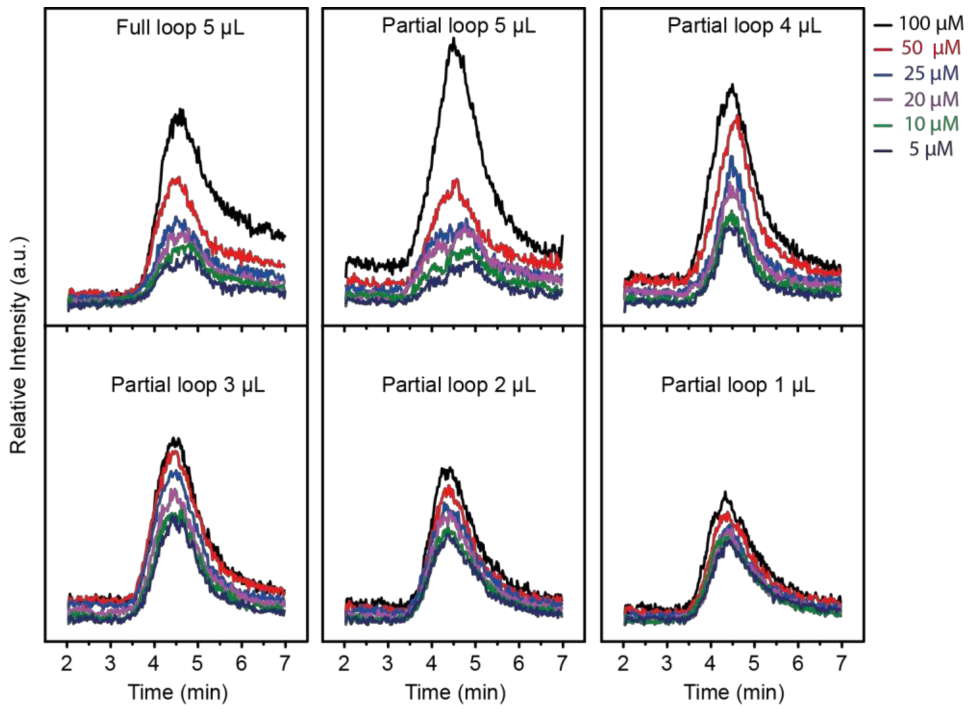


**Figure 9:** Comparison of total ion current against running time of HPLC for dextran. Each graph represents an injection volume, while the lines indicate the variation in concentration.

The experiment compares the effect of concentration on the quality of signal intensity. The injection volume depends on the sample loop, which in this case can store up to 5 µL of sample. While a full loop requires up to 7 µL due to additional sample uptake before and after the sample loop to guarantee an exact volume of 5 µL, the other five injection values represent a partial loop filling. Each graph represents one injection volume with several lines depicting the decreasing concentrations (5 µM - 100 µM). The full loop and the partial loop with 5 µL should result in data with similar in intensity due to the fact, that both have the maximum amount of sample in the sample loop. This is true for concentrations up to 50 µM, while the plots with 100 µM show a surprisingly large difference in intensity in favour of the partial loop. High injection volumes seem to lead to higher diffusion in the capillary resulting in tailing peaks and an overall low intensity in comparison to the smaller injection volumes. The experiments with lower injection volumes from 4 to 1 µL on the other hand show a very homogenous behaviour in terms of intensity and peak form. For decreasing injection volumes, the sample concentration has a minor influence and the overall signal intensity decreases accordingly. To fully characterize the influence of the HPLC running parameters, the impact of varying injection volumes was measured for different concentrations as shown in Figure 10. Comparable offline MS experiments use 3 to 5 µL of sample volume with a concentration of ≈ 5 to 20 µM. For that reason, the corresponding online experiment was done with a limited concentration range from 5 to 25 µM to increase the comparability to standard offline measurements. To validate the reproducibility of the experiment, the measurement was done with dextran as representative of the larger oligosaccharides as well as with the smaller trisaccharide raffinose.

The comparison of dextran and raffinose shows an overall similar trend. While the maximum injection volume of 5 µL results in a quite low intensity, the injection volume of 3 µL seems to be the optimum value to obtain high intensities for a wide variety of concentrations. The ideal concentration for this kind of measurements is difficult to determine, as even the lowest concentration of 5 µM results in sufficient intensity and spectra quality. It is heavily dependent on the ionization efficiency of the respective sample, therefore the concentration can vary between 5 and 25 µM, while the injection volume is optimal at 3 µL. Thus, the sample consumption for one HPLC run is similar as for a single offline injection.

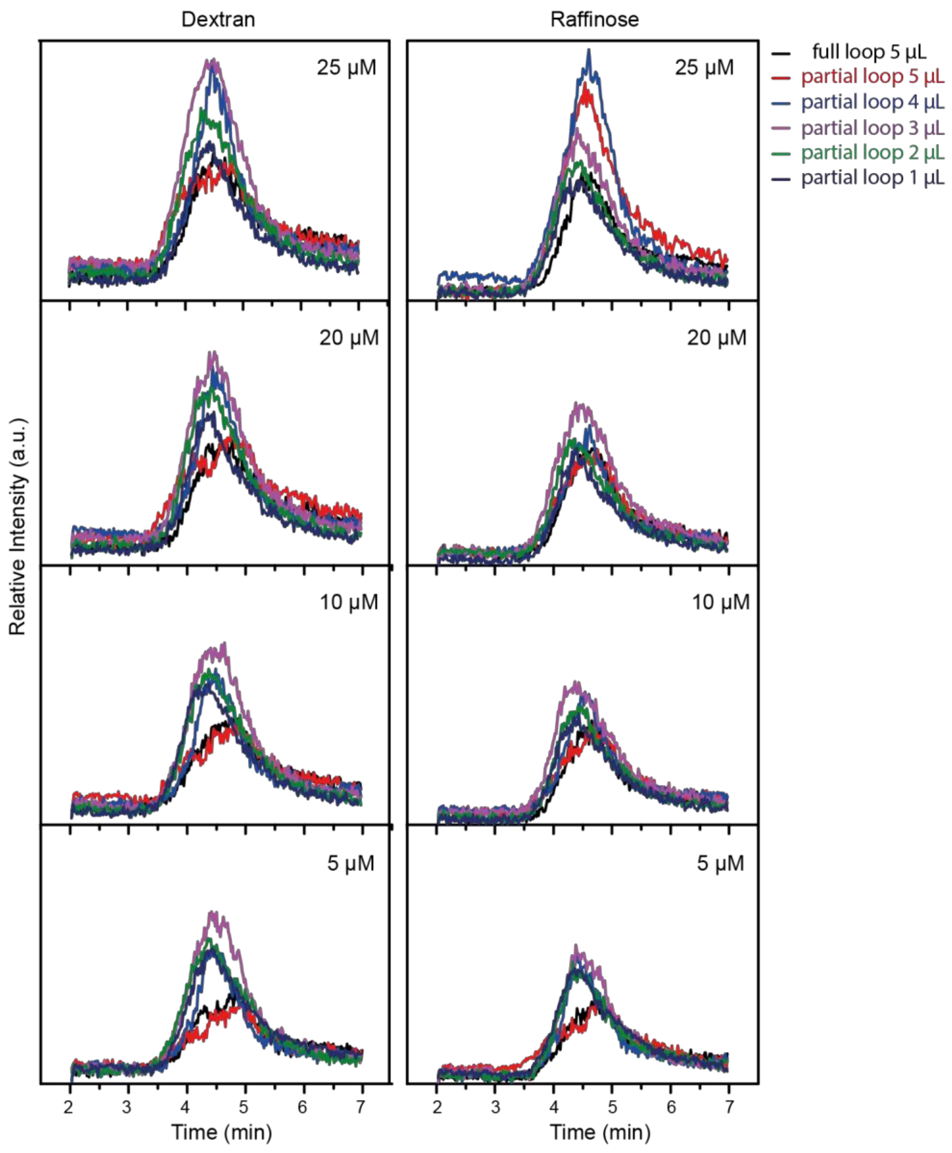
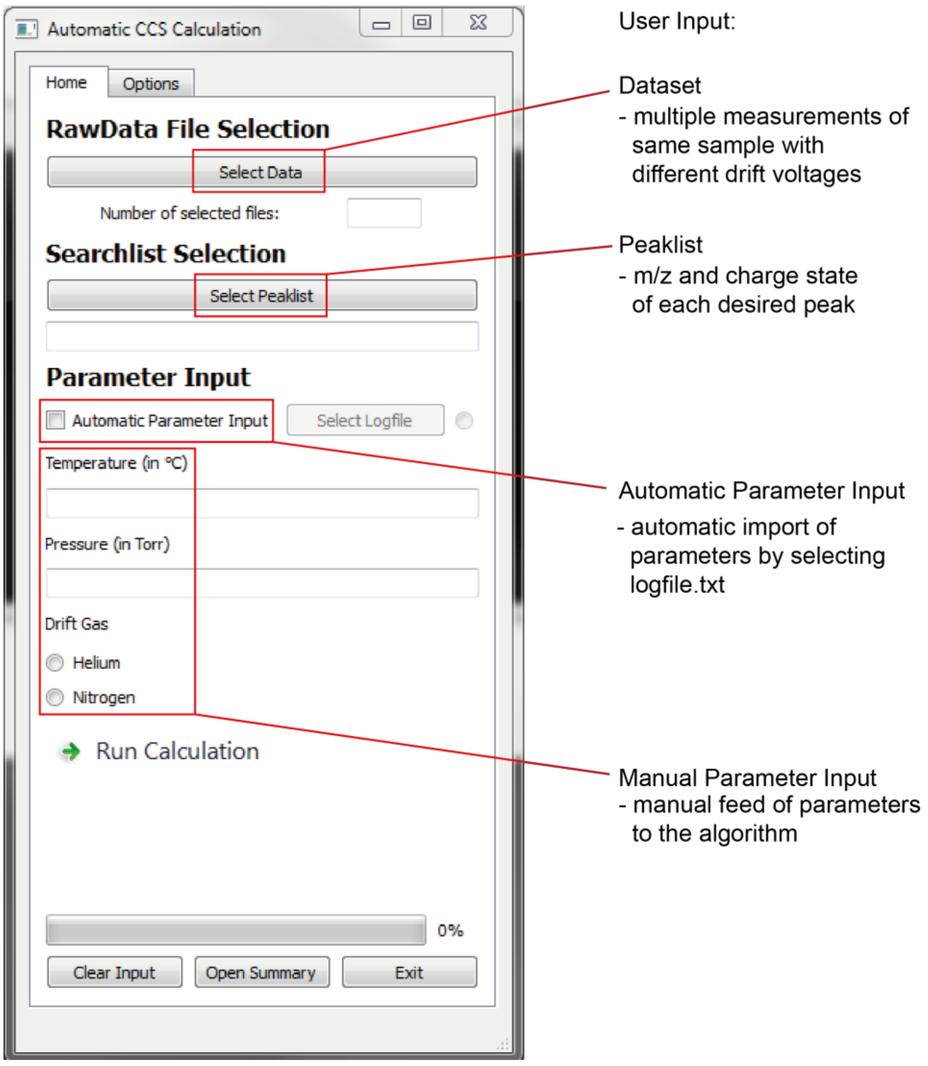


**Figure 10:** Comparison of total ion current against running time of HPLC for dextran and raffinose within a particular concentration range and varying injection volumes as annotated. Each graph contains the data for one concentration while the injection volume is varied.

## 4.2 Data Analysis

### 4.2.1 *Graphic Interface for User Input*

The main factor to accelerate an automation process is a software supported data analysis, therefore an analysing tool was created to replace manual data processing. Aprid is a script which was developed in this thesis for processing ion mobility data in an automated fashion. It is implemented using Python 2.7 and can be operated by the user through a graphic interface devised in PyQtDesigner. The main goal of the script is to determine CCSs of specific *m/z* values. For that purpose, the script follows the mathematical approach of solving the Mason-Schamp equation (see eq. 1), therefore requiring several input values as shown in Figure 11.



Automatic CCS Calculation
Home
Options
RawData File Selection
Select Data
Number of selected files:
Searchlist Selection
Select Peaklist
Parameter Input
Automatic Parameter Input
Select Logfile
Temperature (in °C)
Pressure (in Torr)
Drift Gas
Helium
Nitrogen
Run Calculation
0%
Clear Input
Open Summary
Exit
User Input:
Dataset
- multiple measurements of same sample with different drift voltages
Peaklist
- m/z and charge state of each desired peak
Automatic Parameter Input
- automatic import of parameters by selecting logfile.txt
Manual Parameter Input
- manual feed of parameters to the algorithm

**Figure 11:** Screenshot of Aprids graphical interface and a description of the main input parameters.

The user has to select a dataset as well as a peaklist, which consists of *m/z* values and charge states of the desired peaks. Besides that, the user can choose to either manually input the external parameters or define the location of the *logfile.txt*, which can be used to automatically feed the parameters to the algorithm. After starting the calculation, the script follows a similar workflow to manual data processing.

### 4.2.2 *Workflow of Data Processing*

To calculate CCSs using the Mason-Schamp equation (see eq. 1), the reduced mobility needs to be calculated from measurements as described in equation 3.For that, the electric field *E* is defined as the quotient of the applied voltage *V* in the drift cell and the length of the tube *L*. By considering the definition for *E*, we obtain equation 4:

$$t_D = \frac{L^2}{K}\frac{1}{V} + t_0 \tag{4}$$

Discrete drift times of an ion are acquired by fitting the drift time peak with a gaussian function and taking the centre of that fit as $t_D$. When measuring a dataset with several different drift voltages (at least eight different measurements), the resulting drift times $t_D$ can be treated as a function of the reversed voltages (1/V), therefore allowing equation 4 to be plotted in a linear fashion as shown in Figure 12.

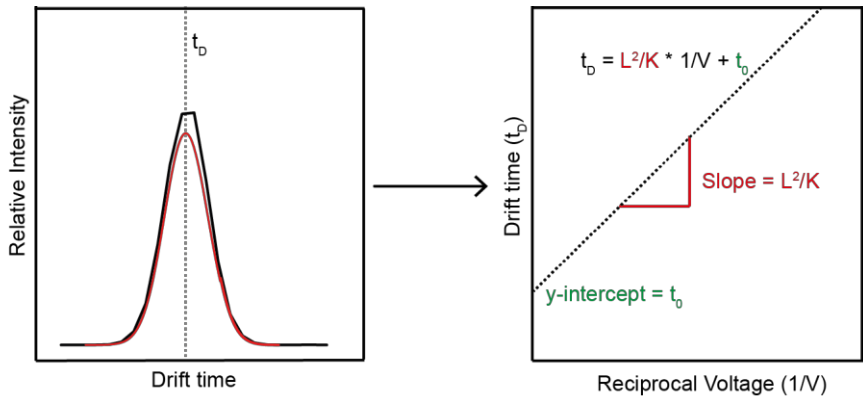


**Figure 12:** Illustration of the gaussian fitting for drift time peaks as well as the linear plot of drift times as described in equation 4 to obtain the mobility *K*.

The time $t_0$ describes the dead time required by ions to traverse the ToF analyser. While the dead time $t_0$ is represented as the y-intercept, the residence time can be derived from the slope, which depends on the length of the tube *L* and the mobility of the ion *K*. Therefore, the obtained mobility *K* can be used to calculate the reduced mobility $K_0$as described before. The calculation of the CCSs then follows the Mason-Schamp equation (see eq. 1).

### 4.2.3 Example of use: Calculation Algorithm for IM-MS data

To illustrate the embedding of the processing workflow into the functioning script, the main algorithm of Aprid is given as the following pseudocode (Algorithm 1).

```
Iterate through set_of_files:
    # parameters
    find parameter_file in raw_file:
        voltage = GetVoltage (parameter_file)
        pusher_frequency = GetPusherInterval (parameter_file)
        timestamp = GetDate (parameter_file)
    # convert raw data
    converted_file = Converter (raw_file + conversion_options)
    # find specific peaks
    find peaks in peaklist (user_input):
        set_limit = peaks+/-sensitivity_threshold
    iterate through m/z in converted_file:
        if lower_limit< (m/z) >upper_limit:
            drift_time = GetDriftTime (m/z)
            results.append (voltage, drift_time, charge)
    # additional parameters
    logfile = OpenTextFile (logfile.txt)
    matching_time = FindDate (timestamp==logfile[time])
    temperature, pressure, drift_gas = GetParameter (matching_time)
    temporary_list.append (temperature, pressure)
repeat for all converted_files
# write output
findvoltage, drift_time, charge in results:
    find temperature, pressure in temporary_list:
    CCS, error_CCS = GetCCS (voltage, drift_time, charge, temperature, pressure)
    output.append (CCS, error_CCS)
write output_file (output)
```

***Algorithm 1:*** *Calculation algorithm for processing IM-MS raw data files. The orange comments will be described in more detail in the following paragraph.*

The script starts by iterating through a set of data files chosen by the user. For each raw file selected, five main steps are accomplished, which are described in the following paragraph.

PARAMETERS. For later processing steps like drift time fitting and CCS calculation, different parameters need to be set. Therefore, the script will parse through the parameter file *_extern.inf* to search for the values of Helium Cell DC, Helium Cell Exit, IMS Bias as well as Transfer DC Entrance and pusher interval. Furthermore, the date and time of the measurement are searched for in the parameter file *_HEADER.TXT*. The pusher interval describes the frequency of MS scans executed for each IMS separation. When ions reach the IMS cell, they are separated based on their drift time as shown before. Each IMS run is divided into 200 individual MS scans, so-called drift bins. This is due to the detection mechanism following after the IMS cell, where the ions arrive at the ToF analyser. There, they are pushed to the detector as ion packages in a defined interval, so-called pusher_frequency. The value of the pusher interval differs for different mass ranges used during acquisition. The overall drift time of an ion can be calculated by the product of the pusher_frequency and the number of drift bins (200) as shown in equation 5.

$$(Pusher\_frequency + 0.00025) * drift\ bins = Drift\ time \qquad (5)$$

The pusher_frequency is saved as a rounded value in *_extern.inf* and needs thus to be shifted by an empirical factor (highlighted in blue in equation 5). The date and time of the measurement is necessary to find the setting details later in the logfile, therefore this values are stored in a way matching the format of the *Logfile.py* output. The drift voltage is calculated from the voltages applied in the drift and fragmentation cells (Figure 13).

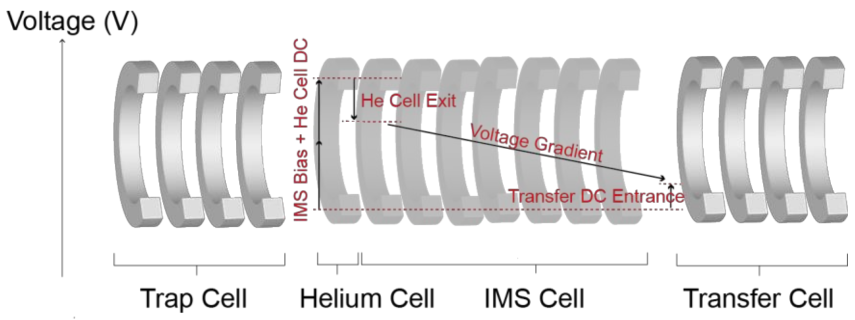


**Figure 13:** Scheme of the required voltages to apply a linear gradient in the drift cell. The ring electrodes illustrate the IMS and fragmentation region, while the black arrows indicate a potential energy diagram.

The highlighted voltages Helium Cell DC and Helium Cell Exit represent the applied voltage for entering and exiting the Helium cell, while IMS Bias is applied to the IMS cell. The

Transfer DC Entrance voltage is the threshold for the ions to reach the following transfer cell. The actual applied drift voltage can be calculated using equation 6.

$$\textit{He Cell DC} + \textit{IMS Bias} + \textit{He Cell Exit} - \textit{Transfer DC Entrance} = \textit{Drift Voltage} \qquad (6)$$

The stored values for *pusher_frequency*, *timestamp* and *voltage* are internal parameters that are automatically created by MassLynx during the measurement. Other parameters like temperature and pressure are measured externally and imported by the script in the later step *additional parameters*.

CONVERT RAW DATA. The raw data files need to be converted to an accessible format first. Therefore, each file is processed by *CDCReader.exe*, a transformation script provided by Waters. It transforms the encoded data points into text files using two libraries (*cdt.dll* and *MassLynxRaw.dll*). The converted text files consist of three rows including the information for *m*/z, drift time and the signal intensity. The conversion is controlled by specific commands, from which the resolution value has the largest impact on the further procedure due to its influence on the accuracy of the measurements as shown in Figure 14. The resolution value can adopt values from zero to one, whereas the value represents the distance between *m*/*z* values in the converted data file.

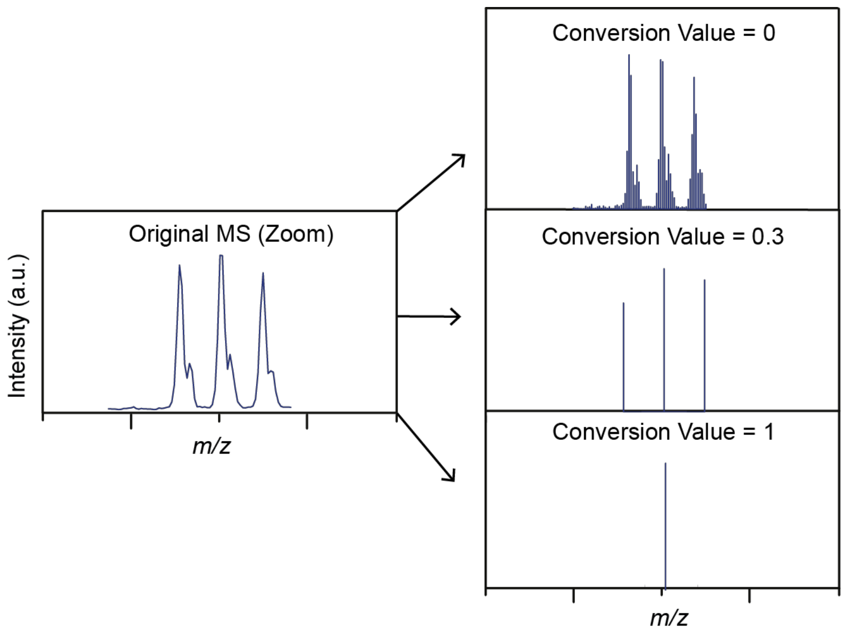


**Figure 14 :** Generic MS spectra to illustrate the impact of the conversion options on the quality of the spectra. Each line in the converted spectra possesses an own drift time peak, which shows the significantly increased processing time to gain discrete drift time information from conversion value 0, while conversion values of 0.3 and 1 offer quite simple spectra and therefore are more rapid to interpret.

The original mass spectrum is zoomed in and shows three peaks. When converting the original file using a resolution value of zero, each peak is split up into several subpeaks, leading to a multiplication of signals. Each of this subpeaks possesses its own drift time peak. To obtain the actual drift time of the entire *m*/*z* peak, each drift time subpeak has to be fitted with a gaussian function. This extra step decreases the performance of the script significantly, therefore the resolution value of zero is not used. Increasing the resolution value leads to more simple spectra, which can be analysed faster and more accurate. The resolution value represents the distance between the *m*/*z* values in the converted file. When converting a file with a resolution value of one, all *m*/*z* peaks are one unit apart. Peaks in between are summed, resulting in rather inaccurate converted spectra. Testing different values lead to an optimal resolution of 0.3, which results in satisfying processing speed and adequate spectra quality. Other conversion parameters were set on default.

FIND SELECTED PEAKS. After the conversion of the raw data, Aprid browses through the converted files looking for *m*/*z* values specified by the user in the peaklist. For this purpose, each *m*/*z* value of the peaklist is used to set delta parameters to define upper and lower bounds, which the algorithm will use to iterate through the converted data files. A peak that falls between the limits is stored with its corresponding drift bins. The overall drift time peak is calculated according to equation 5. This drift time peak is fitted with a gaussian function as depicted in the following pseudocode (algorithm 2).

```
# initial guess
find maximum_intensity in dataset
initial_guess = set point of maximum_intensity as starting point
# single fit
result_1 = FitSingleGaussian (initial_guess)
standard_deviation_1 = CalculateError (result_1)
shift initial_guess
    result_2 = FitSingleGaussian (shifted starting point)
    standard_deviation _2= CalculateError (result_2)
    repeat for multiple shifted starting points
    until residual_error is minimal
if standard_deviation (SingleGaussian) is <given_threshold:
    break algorithm
# multiple fits
else:
    result_1 = FitDoubleGaussian (initial_guess)
    standard_deviation _1 = CalculateError (result_1)
    shift initial_guess
        result_2 = FitSingleGaussian (shifted starting point)
        standard_deviation _2 = CalculateError (result_2)
        repeat for multiple shifted starting points
        until residual_error is minimal
    if minimal_standard_deviation (DoubleGaussian) is <given_threshold:
        break algorithm
    else:
        continue for multiple Gaussians
repeat until minimal_standard_deviation (MultipleGaussian) is <given_threshold
```

**Algorithm 2:** Generic algorithmfor the peak fitting process described as GetDriftTime function in algorithm 1.

After determining the maximum signal intensity of a dataset as initial parameter, the algorithm starts fitting a single gaussian and calculates the root-mean-square deviation (RMSD) for this fit. RMSD is a statistical tool to represent the sample standard deviation of the discrepancy between the data and an estimation model.[64, 65] To find the optimum starting parameter for the fit, the algorithm shifts the initial guess and repeats the fitting process until it reaches a minimal RMSD value. If this minimum satisfies a certain threshold, the algorithm will stop at that point, otherwise it repeats the process with multiple gaussian functions until the threshold is reached. The fitting with multiple gaussian functions repeats several times and is applied especially in case of double/ multiple peaks or peak shoulders. The centre of the fitted gaussian functions represents an accurate value for the drift time $t_D$ of an ion. The results will be appended to a temporary list, which contains the voltage, the drift time and charge for each found peak. The accuracy of the drift time fit is crucial for the CCS calculation later on, therefore the fitting algorithm is evaluated with two well-studied and biochemically relevant oligosaccharides. As shown before, the blood antigen Lewis Y can be identified by the drift time of its fragment ions. The study from HOFMANN ET AL.[7] utilized a travelling wave IM-MS, which gives a higher resolution than the modified drift tube instrument used in this thesis. Nevertheless, Lewis Y was measured to evaluate the capability of the algorithm to fit different kinds of peak shapes. The ATD of the fragment *m/z* 552 measured on both instruments is shown in Figure 15a.

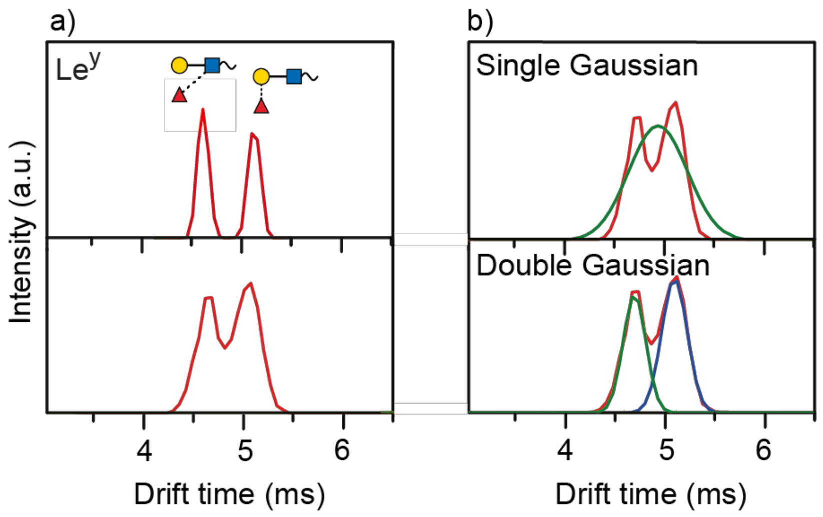


**Figure 15:** a) Direct comparison of the original travelling wave data (upper panel) with the new data measured with a linear drift tube (lower panel) in positive ion mode. The data shows the sodium adducted species at *m/z* 552, resulting from the neutral of fucose from the precursor ion. b) Scheme of the stepwise fitting process of the algorithm. It first tries to fit a single gaussian (green) into the data points (red). When the RMSD value does not satisfy a certain threshold, the algorithm continues to fit multiple gaussians as shown here for the double gaussian fit (green and blue), which satisfies the threshold and stops the fitting algorithm.

The difference in resolution between the travelling wave cell and the linear drift cell is quite significant. While the travelling wave cell can baseline separate both isomers (Figure 15a, upper panel), the linear drift cell shows a broad double peak (Figure 15a, lower panel). The peak fitting algorithm starts the fitting procedure as described before with fitting a single gaussian function to the data (Figure 15b, upper panel). Because the RMSD does not satisfy the threshold, the algorithm adds a second gaussian function (Figure 15b, lower panel). The two gaussians represent the actual measurement accurately, therefore satisfying the threshold and the algorithm stops at this point. Another biologically relevant oligosaccharide used for testing the algorithm is lacto-*N*-hexaose (LNH), which can be found in human milk.[28]

The measurement of the deprotonated precursor ion (*m/z* 1071) is shown in Figure 16.

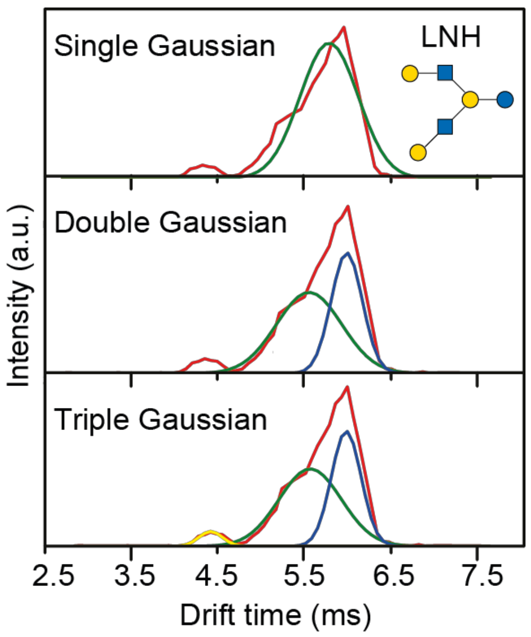


**Figure 16:** Measurement of LNH as deprotonated precursor ion (*m/z* 1071) and the peak fitting process performed by Aprid. The red line represents the measured data points, while the green illustrates the first gaussian fit, blue the second and yellow the third.

The deprotonated precursor shows multiple features due to synthetic impurities. This leads to a broad drift peak with a shoulder (5.5 to 6 ms) and a small second peak (4.5 ms). The algorithm starts by default with a single gaussian at the maximum point (6 ms), which does not represent the actual measurement. The two gaussians depict the overall peak shape pretty well, but the small peak at lower drift time is ignored due to the low intensity. Despite the missing peak fit, the RMSD is satisfying the threshold and the algorithm would stop at that point. When manually lowering the threshold, the algorithm continues and fits a third gaussian function to the missing peak. The actual difference between two and three gaussians is somewhat small in terms of standard deviation due to the small occupied area in comparison to the broadened peak. In ideal case, the threshold should be minimal to allow realistic fits, but the data at times has a high signal-to-noise ratio, which would lead to mismatches in the process. Several experiments lead to an empirical threshold, which has a good balance of accuracy and performance.

ADDITIONAL PARAMETERS. Certain parameters like temperature, pressure, and type of drift gas are extrinsic values measured with the help of external instruments, but they are needed to convert the drift time into a CCS. There are two ways to provide the program with these data. The user can either input the external parameters manually for each calculation or specify the location of the *logfile.txt*, which allows the algorithm to import the desired parameters automatically. In order to do that, each data file in the dataset is examined for its time of measurement. After that, Aprid searches for that points of time in the *logfile.txt* to assign the respective pressure and temperature values to each data file, which are afterwards stored in two temporary lists. The lists consist of at least eight values, which are averaged to result in a single value of pressure and temperature for each dataset that can be used for later calculations. If one or more values deviate more than a defined threshold from the others, the user will receive a warning message to check the *logfile.txt* manually.

WRITE OUTPUT. To calculate a CCS using the Mason-Schamp equation 1, the reduced mobility $K_0$ needs to be calculated from the measurements as shown in Figure 12 before. The slope of the linear fit results in the mobility $K$, which can be used to calculate the reduced mobility $K_0$ as described in equation 3. The calculation of the CCS then follows the Mason-Schamp equation 1. The final output file contains two sections, with the searched values for *m/z* and charge in one section, and the found values for *m/z* and CCS in the other row. Furthermore, the error of each CCS value is given as well as the acquired parameter values for temperature, pressure, and drift gas used for the calculation. It is written to a simple comma separated file (csv) to allow a fast summary of the results. The evaluation of the results can be carried out in various ways. The accuracy of the results can either be obtained by the $R^2$ value from the linear drift time plot or as calculated error from slope, mobility or CCS. Besides numerical values, graphical output can also be obtained with minor changes in the output algorithm. The final output format is yet to be determined due to a missing database to upload the results and will be adapted in the near future.

## 4.3 Evaluation

### 4.3.1 *MS Experiments*

To evaluate the automated setup, dextran was measured several times to track possible deviations in the results and external parameters. The CCSs in this part are obtained by utilizing the predetermined optimum concentration range of 5 to 25 µM and injection volume of 3 µL. Dextran is very suitable for online MS experiments due its broad molar mass distribution which allows observing multiple ions in the spectrum without the need to isolate or fragment specific ions. Figure 17 shows the measured IM-MS data for dextran and the calculated CCS information.

a)

| *m/z* \ $No_{EXP}$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | $CCS_{AVG}$ | STD |
|---|---|---|---|---|---|---|---|---|---|---|
| 527.34 | 210.0 | 210.2 | 211.6 | 210.3 | 212.2 | 212.9 | 211.2 | 210.8 | 211.16 | 1.11 |
| 689.33 | 237.2 | 236.6 | 237.1 | 236.6 | 237.0 | 237.2 | 236.9 | 237.0 | 236.95 | 0.27 |
| 851.32 | 263.3 | 262.6 | 263.4 | 262.7 | 263.1 | 263.2 | 263.2 | 263.1 | 263.08 | 0.31 |
| 1013.32 | 287.6 | 287.8 | 287.2 | 287.1 | 287.4 | 287.3 | 287.1 | 287.2 | 287.34 | 0.26 |

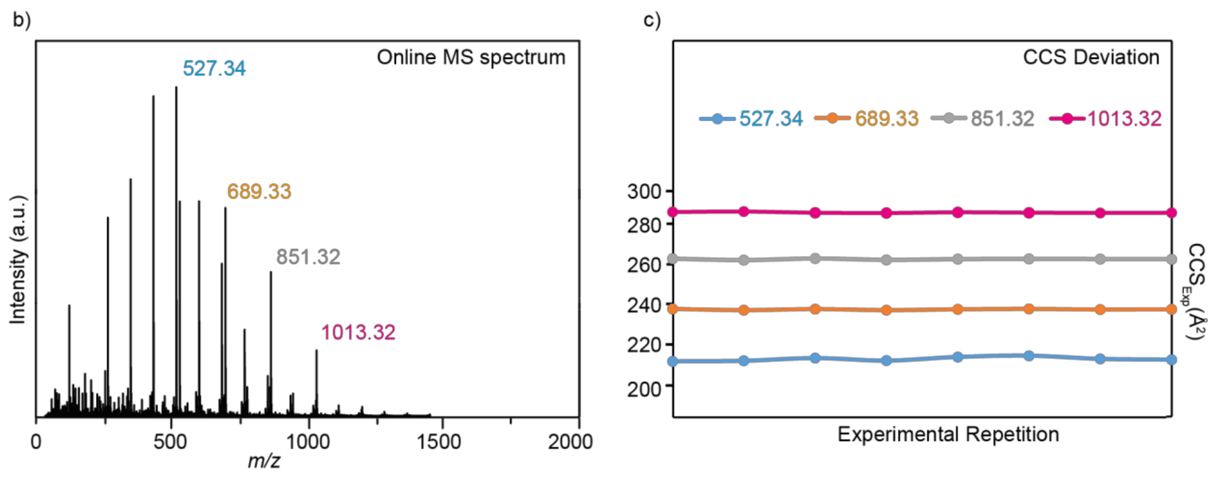


**Figure 17:** Evaluation of stability trends of CCS determination using dextran. a) Table of the measured $CCS_{N2}$ (in Å²) for four $[M+Na]^+$ ions of dextran. The table also illustrates the average ($CCS_{AVG}$) and standard deviation (STD) of each dataset. b) Online MS spectrum of dextran. Several peaks of the spectrum are identified, but for reasons of simplicity, only the highlighted peaks are focused on in this thesis. c) Illustration of the CCS deviation for a high number of experimental repetitions.

There are several observed peaks in the positive ion mode spectrum, representing protonated and adduct species, but only four specific ions were examined in order to simplify the evaluation. The four highlighted peaks in the depicted spectrum represent the sodium adduct-ions of the respective tri-, tetra-, penta- and hexasaccharide. The comparison of their CCSs for the repetitive measurements shows minimal fluctuation between the calculated CCS. This can be observed in the small standard deviation of the datasets as well as in the deviation chart (Figure 17a/c), which illustrates the overall steady trend of the repetitive CCS measurements. To further prove the concept for MS/MS experiments, raffinose was measured in a similar approach.

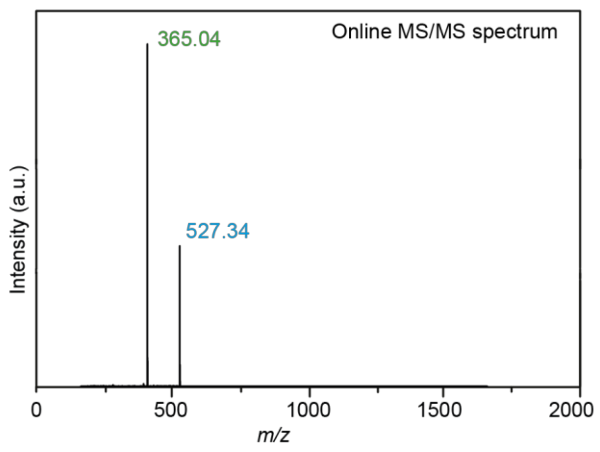
## 4.3.2 *MS/MS experiments*

Raffinose is a well-studied non-reducing trisaccharide, consisting of galactose, glucose and fructose. Unfortunately, the amount of CCS reference data is quite low, but as an oligosaccharide, it represents a typical target of the relevant substance class, therefore allowing to draw conclusions about the results as proof of principle. Furthermore, it has a simple fragmentation pattern in positive ion mode and thus the fragmentation experiments in MS/MS mode are carried out with raffinose as sample. The results of the repetitive CCS measurements in MS/MS mode are shown in Figure 18.

a)

| *m/z* \ No$_{EXP}$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | CCS$_{AVG}$ | STD |
|---|---|---|---|---|---|---|---|---|---|---|
| 527.34 | 212.7 | 210.6 | 206.6 | 210.4 | 210.2 | 207.7 | 209.3 | 210.1 | 209.71 | 2.01 |
| 365.04 | 180.2 | 179.2 | 172.7 | 178.8 | 178.8 | 174.7 | 177.2 | 179.5 | 177.63 | 2.74 |

b)



c)

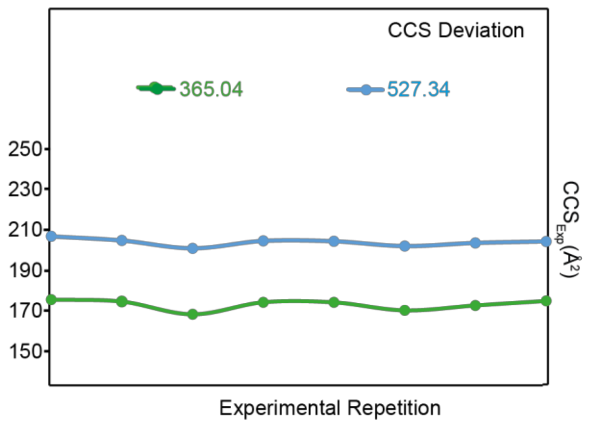


**Figure 18:** Evaluation of automated MS/MS experiments with raffinose. a) Table of the measured CCS (in Å$^2$) for the [M+Na]$^+$ precursor and fragment ion of raffinose. The table provides the average (CCS$_{AVG}$) and standard deviation (STD) of both datasets. b) Online MS/MS spectrum of raffinose, which was produced using a collision energy of 30 V in the trap cell. The peaks represent the precursor ion (blue) and the fragment ion (green). c) Illustration of the CCS deviation for a high number of experimental repetitions.

The fragmentation pattern of raffinose shows one fragment, which represents the cleavage of one building block. The comparison of the CCSs of the precursor ion as well as of the fragment ion over different experimental repetitions shows a higher fluctuation than for dextran before (Figure 17). While the maximum standard deviation for dextran lies at one Å$^2$ (0.5 %), raffinose reveals a relatively high standard deviation of two to three Å$^2$ (1 to 1.5 %), especially for the fragment ion. This could indicate a bigger impact on smaller *m/z* in terms of fluctuation between measurements. The reason could lie in software problems regarding the control of the gas flow in the trap cell. Despite the input of a specific value, MassLynx shifts the value to lower regions, therefore leading to an inconstant gas flow in the trap cell. This could influence the collisions of the ions with the buffer gas in the fragmentation cell, from which larger ions are more affected than smaller ions. Nevertheless, the repetition of the CCS measurements in MS/MS mode shows a similar trend to the MS mode, therefore proving that the approach can be applied to both acquisition modes. Both showed the anticipated results and can compete with offline measurements in terms of signal intensity and quality. Moreover, the timescale of data acquisition for online measurements is comparable to the time required to measure offline data, but there is still a major flaw in terms of sample consumption and time. Due to the modified IM-MS instrument, the voltage regulation for the sample list of MassLynx is constrained to one value only for each run. While one injection would be more than enough to record IM spectra at different voltages, the software does not allow voltage regulation inside one run. This means, that for efficient CCS measurements, each sample needs to be sprayed eight times, therefore needing eight times the usual sample amount and time. Future application focus on circumvent this limitation. Nevertheless, the overall timescale of the entire automated setup decreased dramatically due to the software supported processing. Aprid allows the analysis of datasets in few minutes, independent from the number of peaks and enabling the analysis of even complex oligosaccharides. Furthermore, the small deviations in the CCS calculation indicate an accurate processing of the data. This means, that the automated setup works and could enable the measurement of a high number of samples in a short amount of time.

# 5 Conclusion and Outlook

In the frame of this thesis, a high throughput setup for an automated IM-MS data acquisition and data analysis were established. The automated data acquisition is realised by the coupling of IM-MS to HPLC to allow the measurement of multiple samples in an automated fashion, while the data analysis is performed by a self-designed software tool.

Dextran and raffinose were used to characterize the new established setup for the automated online measurements in terms of sample consumption, time, robustness and accuracy. The systematic analysis showed a similar sample consumption and time amount as it is the case for offline methods. Using an injection volume of 3 μL, an ideal concentration range from 5 to 25 μM was observed. This shows, that the sensitivity of the new setup is very high with a sample requirement of pmol or ng scales per run.

The modification of the Synapt G2-S instrument with a new linear drift cell and the old MassLynx software does, however, not allow the simultaneous adjustment of the drift voltage during HPLC runs. Multiple runs are required to obtain reliable CCS values derived from at least eight different drift voltage, which further increase the actual required sample amount. In order to overcome this software problem for future applications, the instrument and software supplier is now involved.

Nevertheless, the total sample consumption is still below alternative techniques like NMR which usually requires mg scales.

The self-developed software Aprid is implemented using Python 2.7 and showed enormous potential to decrease the processing time of IM-MS data by replacing the manual data interpretation. Besides the decrease in timescale to determine CCSs, Aprid has the ability to analyse even complex ATDs of oligosaccharides. This was validated on the basis of complex drift time peaks of Lewis Y and LNH, which showed multiple peaks and peak shoulders. The stepwise approach of fitting an increasing amount of gaussian functions leads to a reliable data interpretation. Nevertheless, the limitation of this new approach relies in fitting low intensity peaks and needs to be further improved. This observation was obtained for the low intensity drift time peaks of LNH, caused by impurities of the synthesis. The threshold, which determines the accuracy of the fitting procedure, was too high to recognise this low intensity peak and therefore a mismatch in the estimation model was observed. High signal-to-noise ratios do not allow to set small threshold values and instead a compromise between accuracy and performance has to be made. The automated fitting procedure could in future benefit from adjustments of specific fitting parameters such as baseline correction, width and height of the gaussian fits, which would further allow to set a lower threshold value for CCS determination.

The entire instrumental setup, including the software-supported data processing, was evaluated in MS mode using Dextran as well as in MS/MS mode using raffinose. The signal intensity and quality of the recorded spectra is in excellent agreement with previous offline experiments. A high repetition number showed small fluctuations of 0 to 1.5 % in the calculated CCSs for precursor as well as fragment ions. These fluctuations are also often observed for individual offline measurements and therefore the new described automated method can be used to obtain reference data for future implementations in databases. The experiments in this thesis were exclusively performed in the positive ion mode. Further measurements will be performed in future to also validate this new method in the negative ion mode as well as for the investigation of adduct-ion species.

The current combination of HPLC and IM-MS in this work is managed without separation column. Future applications should include this additional separation step to acquire retention times from HPLC runs as an additional characterisation factor for the carbohydrate analysis. This could increase the accuracy of the identification as well as the possibilities of applications, e.g. in terms of mixture separation as well as derivatisation of samples as shown in Figure 19.

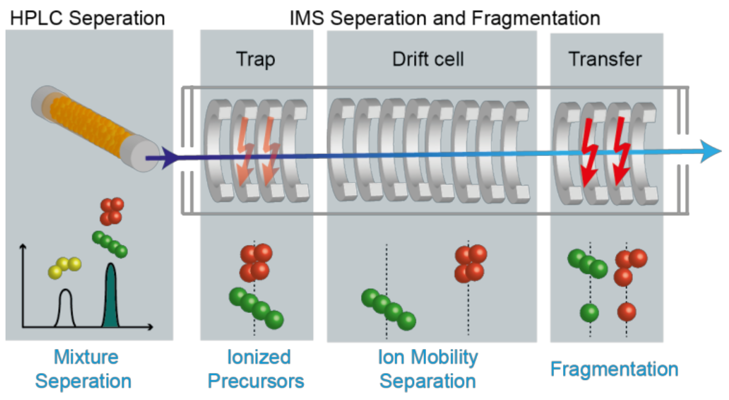


**Figure 19:** Modified setup of the instrument used in this thesis. Adding a column to the HPLC would allow chromatographic separation as complementary information, which increases the resolution of the method.

Furthermore, the timescale of IM-MS experiments is complementary to HPLC runs, allowing to combine both, the retention time, mass and CCSs in future databases. Such a database could set the benchmark for carbohydrate characterisation. The quality of the database, however, correlates with the amount of data stored within and therefore it is crucial to accumulate an enormous amount of reference data and to build up the basic framework for a software supported database browse. Methods for database searching will rely on specific properties of smaller oligosaccharides to draw conclusions on the structure of larger polysaccharides. Especially small oligosaccharides with a specific repeating core unit as lactose and mannose structure as well as the already studied antigens will take up an important role to allow automatic identification of their larger counterparts in database searches.

# 6 Experimental Section

## 6.1 Instrumental Setup

Experiments were performed on a nanoAcquity UPLC system (Waters, Milford, USA) and a modified Synapt G2-S HDMS (Waters, Milford, USA) equipped with a nano-electrospray ionization source (nESI) as shown inFigure 20. The originally built-in travelling wave cell of the Synapt G2-S was replaced by a linear drift cell to allow the direct conversion of measured drift times into CCSs. The instrument is coupled to a UPLC to enable automatic sample uptake with the help of the autosampler.

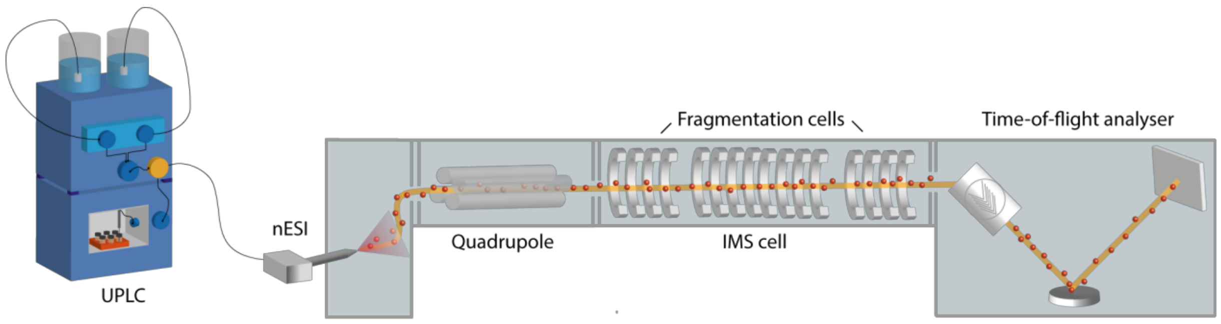


**Figure 20:** Schematic setup of an UPLC coupled to an IM-MS instrument. The UPLC system is used without column and acts as autosampler to directly inject samples through the nESI source into the ion mobility-mass spectrometer.

### 6.1.1 *Nano UPLC*

The nano UPLC system is made up of a sample manager containing an autosampler with two 24 position 1.5 mL vial plates and a binary solvent manager with two HPLC pumps with solvent degasser and a high pressure gradient mixer. The UPLC contained a sample loop that takes up to 5.0 µL of sample and was operated with a flow rate of 1.0 µL/min using a defined gradient of solvent, usually acetonitrile (ACN) and water (*v*/*v*, 1:1), both containing 0.1 % formic acid (FA). The column was replaced by a fused silica capillary with the attributes of 360 µm outer-diameter (OD) and 100 µm inner diameter (ID) (Idex Health & Science, Oak Harbor, USA), which was directly connected to the ion mobility-mass spectrometer.

### 6.1.2 *Ion Mobility-Mass Spectrometer*

The IM-MS instrument is composed of a nESI source with a pre-cut TaperTip online emitter (360 µm OD x 20 µm ID, Waters, Milford, USA), an ion guide, a quadrupole, two fragmentation cells (trap and transfer cell) as well as an ion mobility cell in between and a time-of-flight mass analyser. The originally installed travelling-wave drift cell was replaced by a linear drift tube using a design reported previously. The online emitter was used for ionizing samples provided by the UPLC, while offline measurements were performed using in-house made metal-coated glass capillaries. Typical instrument setting were: capillary voltage, 2.0-3.5 kV (0.8 kV for offline measurements); sample cone, 70.0 V; source offset, 40.0 V; source temperature, 30 °C; Trap DC Entrance, 2.0 V; Trap DC Bias, -15.0 V; Trap DC Exit, 2.0 V; IMS DC Entrance, -20.0 V; Helium Cell DC, 110.0-160.0 V; Helium Exit, -40.0 V; IMS Bias, 110.0-160.0 V; Transfer DC Entrance, 5.0 V; Transfer DC Exit, 10.0 V; backing pressure, 3.8 mbar; trap pressure, $2.45 \times 10^{-2}$ mbar; ion mobility gas, $N_2$ or He; ion mobility cell pressure, 3.5 mbar; time-of-flight analyser pressure, $1.0 \times 10^{-6}$ mbar. Each of the parameters were, however, optimized in order to have an abundant signal for the parent ion.

Both instruments, the Synapt G2-S as well as the UPLC, are controlled by MassLynx V4.1 (Waters, Milford, USA) with either direct manual control or with the help of a sample list, which allows to automatically run the entire instrumentation for longer periods of time without user interference.

## 6.2 Data Interpretation

For manual data analysis the resulting data points were extracted with MassLynx V4.1 and the drift times were fitted with a Gaussian distribution using OriginPro 2015G (OriginLab Corporation, Northampton, USA). Fitted drift times were converted to collision cross-sections as described before. All graphics were produced using Adobe Illustrator CS6 (Adobe Systems GmbH, München, Germany).

## 6.3 Samples

Prior to HPLC-IM-MS the samples were typically diluted in water/methanol (1:1, *v*/*v*) or water/ACN (1:1, *v*/*v*). Commercially available samples were used without any further purification.

Dextran and raffinose were purchased from Sigma Aldrich (Munich, Germany).

LNT/LNnT and LNH/LNnH standards were purchased from Carbosynth Limited (Berkshire, UK).

Lewis antigens were purchased from Dextra Laboratories (Reading, UK), with the exception of BG-H1, which was purchased from Elicityl SA (Crolles, France).

Non-commercial samples were prepared by solid phase synthesis by Heung Sik Hahm (Group of P. Seeberger, Max Planck Institute of Colloids and Interfaces) and were diluted in water/methanol (1:1, *v*/*v*).

# References

[1] J. Berg, J. Tymoczko, L. Stryer, *Biochemistry*, 5th ed., W. H. Freeman, New York, **2002**.

[2] M. D. Disney, P. H. Seeberger, *Chem. Biol.* **2004**, *11*, 1701-1707.

[3] R. Kleene, M. Schachner, *Nat. Rev. Neurosci.* **2004**, *5*, 195-208.

[4] G. A. Khoury, R. C. Baliban, C. A. Floudas, *Sci. Rep.* **2011**, *1*.

[5] R. S. Haltiwanger, J. B. Lowe, *Annu. Rev. Biochem.* **2004**, *73*, 491-537.

[6] N. G. Karlsson, K. A. Thomsson, *Glycobiology* **2009**, *19*, 288-300.

[7] J. Hofmann, A. Stuckmann, M. Crispin, D. J. Harvey, K. Pagel, W. B. Struwe, *Anal. Chem.* **2017**, *89*, 2318-2325.

[8] R. P. Estrella, J. M. Whitelock, N. H. Packer, N. G. Karlsson, *Biochem. J.* **2010**, *429*, 359-367.

[9] L. Schofield, M. C. Hewitt, K. Evans, M.-A. Siomos, P. H. Seeberger, *Nature* **2002**, *418*, 785-789.

[10] D. B. Werz, P. H. Seeberger, *Angew. Chem. Int. Ed.* **2005**, *44*, 6315-6318.

[11] B. Aussedat, Y. Vohra, P. K. Park, A. Fernández-Tejada, S. M. Alam, S. M. Dennison, F. H. Jaeger, K. Anasti, S. Stewart, J. H. Blinn, H.-X. Liao, J. G. Sodroski, B. F. Haynes, S. J. Danishefsky, *J. Am. Chem. Soc.* **2013**, *135*, 13113-13120.

[12] R. A. Dwek, *Chem. Rev.* **1996**, *96*, 683-720.

[13] C. R. Bertozzi, D. Rabuka, in *Essentials of Glycobiology*, 2nd ed. (Eds.: A. Varki, R. D. Cummings, J. D. Esko, H. H. Freeze, P. Stanley, C. R. Bertozzi, G. W. Hart, M. E. Etzler), Cold Spring Harbor (NY), **2009**.

[14] T. R. I. Cataldi, C. Campa, M. Angelotti, S. A. Bufo, *J. Chromatogr. A* **1999**, *855*, 539-550.

[15] U. M. Abd Hamid, L. Royle, R. Saldova, C. M. Radcliffe, D. J. Harvey, S. J. Storr, M. Pardo, R. Antrobus, C. J. Chapman, N. Zitzmann, J. F. Robertson, R. A. Dwek, P. M. Rudd, *Glycobiology* **2008**, *18*, 1105-1118.

[16] J. Ø. Duus, C. H. Gotfredsen, K. Bock, *Chem. Rev.* **2000**, *100*, 4589-4614.

[17] A. Dell, H. R. Morris, *Science* **2001**, *291*, 2351-2356.

[18] D. J. Harvey, *Proteomics* **2001**, *1*, 311-328.

[19] D. J. Harvey, *J. Am. Soc. Mass Spectrom.* **2005**, *16*, 631-646.

[20] D. J. Harvey, *J. Am. Soc. Mass Spectrom.* **2005**, *16*, 647-659.

[21] N. G. Karlsson, B. L. Schulz, N. H. Packer, *J. Am. Soc. Mass Spectrom.* **2004**, *15*, 659-672.

[22] J. M. Prien, B. D. Prater, S. L. Cockrill, *Glycobiology* **2010**, *20*, 629-647.

[23] M. Pabst, J. S. Bondili, J. Stadlmann, L. Mach, F. Altmann, *Anal. Chem.* **2007**, *79*, 5051-5057.

[24] G. R. Guile, P. M. Rudd, D. R. Wing, S. B. Prime, R. A. Dwek, *Anal. Biochem.* **1996**, *240*, 210-226.

[25] M. Zhu, B. Bendiak, B. Clowers, H. H. Hill, *Anal. Bioanal. Chem.* **2009**, *394*, 1853-1867.

[26] K. Pagel, D. J. Harvey, *Anal. Chem.* **2013**, *85*, 5138-5145.

[27] J. Hofmann, H. S. Hahm, P. H. Seeberger, K. Pagel, *Nature* **2015**, *526*, 241-244.

[28] W. B. Struwe, C. Baldauf, J. Hofmann, P. M. Rudd, K. Pagel, *Chem. Commun.* **2016**, *52*, 12353-12356.

[29] D. C. Collins, M. L. Lee, *Anal. Bioanal. Chem.* **2002**, *372*, 66-73.

[30] R. Cumeras, E. Figueras, C. E. Davis, J. I. Baumbach, I. Gracia, *Analyst* **2015**, *140*, 1376-1390.

[31] L. S. Fenn, J. A. McLean, *Phys. Chem. Chem. Phys.* **2011**, *13*, 2196-2205.

[32] H. Hinneburg, J. Hofmann, W. B. Struwe, A. Thader, F. Altmann, D. Varon Silva, P. H. Seeberger, K. Pagel, D. Kolarich, *Chem. Commun.* **2016**, *52*, 4381-4384.

[33] K. F. Aoki-Kinoshita, *PLoS Comput. Biol.* **2008**, *4*, e1000075.

[34] J. C. Tran, L. Zamdborg, D. R. Ahlf, J. E. Lee, A. D. Catherman, K. R. Durbin, J. D. Tipton, A. Vellaichamy, J. F. Kellie, M. Li, C. Wu, S. M. Sweet, B. P. Early, N. Siuti, R. D. LeDuc, P. D. Compton, P. M. Thomas, N. L. Kelleher, *Nature* **2011**, *480*, 254-258.

[35] J. F. Rakus, L. K. Mahal, *Annu. Rev. Anal. Chem.* **2011**, *4*, 367-392.

[36] G. Lauc, J. Kristic, V. Zoldos, *Front. Genet.* **2014**, *5*, 145.

[37] E. Juaristi, G. Cuevas, *The anomeric effect*, CRC Press, Boca Raton and London, **1995**.

[38] R. A. Laine, *Glycobiology* **1994**, *4*, 759-767.

[39] R. D. Cummings, *Mol. BioSyst.* **2009**, *5*, 1087-1104.

[40] A. Varki, R. D. Cummings, M. Aebi, N. H. Packer, P. H. Seeberger, J. D. Esko, P. Stanley, G. Hart, A. Darvill, T. Kinoshita, J. J. Prestegard, R. L. Schnaar, H. H. Freeze, J. D. Marth, C. R. Bertozzi, M. E. Etzler, M. Frank, J. F. Vliegenthart, T. Lutteke, S. Perez, E. Bolton, P. Rudd, J. Paulson, M. Kanehisa, P. Toukach, K. F. Aoki-Kinoshita, A. Dell, H. Narimatsu, W. York, N. Taniguchi, S. Kornfeld, *Glycobiology* **2015**, *25*, 1323-1324.

[41] K. M. Roscioli, E. Davis, W. F. Siems, A. Mariano, W. Su, S. K. Guharay, H. H. Hill, Jr., *Anal. Chem.* **2011**, *83*, 5965-5971.

[42] V. Ruzsanyi, *J. Breath. Res.* **2013**, *7*, 046008.

[43] L. S. Fenn, M. Kliman, A. Mahsut, S. R. Zhao, J. A. McLean, *Anal. Bioanal. Chem.* **2009**, *394*, 235-244.

[44] J. Seo, W. Hoffmann, S. Warnke, M. T. Bowers, K. Pagel, G. von Helden, *Angew. Chem. Int. Ed. Engl.* **2016**, *55*, 14173-14176.

[45] K. Thalassinos, M. Grabenauer, S. E. Slade, G. R. Hilton, M. T. Bowers, J. H. Scrivens, *Anal. Chem.* **2009**, *81*, 248-254.

[46] D. R. Hernandez, J. D. DeBord, M. E. Ridgeway, D. A. Kaplan, M. A. Park, F. Fernandez-Lima, *Analyst* **2014**, *139*, 1913-1921.

[47] W. B. Struwe, J. L. Benesch, D. J. Harvey, K. Pagel, *Analyst* **2015**.

[48] B. C. Bohrer, S. I. Merenbloom, S. L. Koeniger, A. E. Hilderbrand, D. E. Clemmer, *Annu. Rev. Anal. Chem.* **2008**, *1*, 293-327.

[49] W. B. Struwe, K. Pagel, J. L. Benesch, D. J. Harvey, M. P. Campbell, *Glycoconj. J.* **2016**, *33*, 399-404.

[50] E. A. Mason, H. W. Schamp, *Ann. Phys.* **1958**, *4*, 233-270.

[51] H. E. Revercomb, E. A. Mason, *Anal. Chem.* **1975**, *47*, 970-983.

[52] K. Giles, S. D. Pringle, K. R. Worthington, D. Little, J. L. Wildgoose, R. H. Bateman, *Rapid Commun. Mass Spectrom.* **2004**, *18*, 2401-2414.

[53] J. Hofmann, W. B. Struwe, C. A. Scarff, J. H. Scrivens, D. J. Harvey, K. Pagel, *Anal. Chem.* **2014**, *86*, 10789-10795.

[54] S. J. Allen, K. Giles, T. Gilbert, M. F. Bush, *Analyst* **2016**, *141*, 884-891.

[55] A. V. Everest-Dass, J. L. Abrahams, D. Kolarich, N. H. Packer, M. P. Campbell, *J. Am. Soc. Mass Spectrom.* **2013**, *24*, 895-906.

[56] M. Crispin, D. I. Stuart, E. Y. Jones, *Nat. Struct. Mol. Biol.* **2007**, *14*, 354-354.

[57] M. P. Campbell, R. Peterson, J. Mariethoz, E. Gasteiger, Y. Akune, K. F. Aoki-Kinoshita, F. Lisacek, N. H. Packer, *Nucleic Acids Res.* **2014**, *42*, D215-221.

[58] R. Raman, M. Venkataraman, S. Ramakrishnan, W. Lang, S. Raguram, R. Sasisekharan, *Glycobiology* **2006**, *16*, 82R-90R.

[59] C.-W. von der Lieth, A. A. Freire, D. Blank, M. P. Campbell, A. Ceroni, D. R. Damerell, A. Dell, R. A. Dwek, B. Ernst, R. Fogh, M. Frank, H. Geyer, R. Geyer, M. J. Harrison, K. Henrick, S. Herget, W. E. Hull, J. Ionides, H. J. Joshi, J. P. Kamerling, B. R. Leeflang, T. Lütteke, M. Lundborg, K. Maass, A. Merry, R. Ranzinger, J. Rosen, L. Royle, P. M. Rudd, S. Schloissnig, R. Stenutz, W. F. Vranken, G. Widmalm, S. M. Haslam, *Glycobiology* **2011**, *21*, 493-502.

[60] M. F. Bush, Z. Hall, K. Giles, J. Hoyes, C. V. Robinson, B. T. Ruotolo, *Anal. Chem.* **2010**, *82*, 9557-9565.

[61] D. Xia, F. Ghali, S. J. Gaskell, R. O'Cualain, P. F. Sims, A. R. Jones, *Proteomics* **2012**, *12*, 1912-1916.

[62] M. T. Marty, A. J. Baldwin, E. G. Marklund, G. K. Hochberg, J. L. Benesch, C. V. Robinson, *Anal. Chem.* **2015**, *87*, 4370-4376.

[63] T. M. Allison, E. Reading, I. Liko, A. J. Baldwin, A. Laganowsky, C. V. Robinson, *Nat. Commun.* **2015**, *6*, 8551.

[64] J. S. Armstrong, F. Collopy, *Int. J. Forecast.* **1992**, *8*, 69-80.

[65] R. J. Hyndman, A. B. Koehler, *Int. J. Forecast.* **2006**, *22*, 679-688.

# Appendix

The appendix contains all written code of this thesis. The first part describes the main algorithm code, which represents the user operation by controlling an interface, data accommodation of required input parameters and processing tools to create the desired output. The second algorithm describes the logging script, which is responsible for saving external measurement parameter such as temperature and pressure in a formatted way as text file. The third script is similar to the second in terms of reading external measurement data, but instead of saving it to a text file, it enables a visual interface to the user to control all parameter and change them manually if required.

The code is highlighted in different colours to simplify the reading. Green sentences starting with the hashtag-sign (#) highlight personal comments for better understanding, while blue words describe Python key words and imports from external libraries. Besides Python library imports to allow mathematical operations, there are PyQtDesigner library imports to simplify user operation and allow parallel threads. Pink colour indicates self-written functions, which describe standard working steps to use in the whole script. Standard written code is simply black.

```python
# import libraries
import sys
from scipy import optimize
from PyQt4.QtCore import SIGNAL
from PyQt4.QtGui import QFileDialog
from PyQt4 import QtGui, QtCore, uic
from scipy import asarray as ar,exp
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import numpy as np
import os, fnmatch
import subprocess
import operator
import scipy
from scipy import stats
import csv
from numpy import *
from pylab import *
import math

# seperate thread to open the calculated results after running the script
class OpenSummary(QtCore.QThread):
    def __init__(self):
        QtCore.QThread.__init__(self)

    def run(self):
        subprocess.Popen(UpdateThread.OpenOutput, shell=True).wait()

# seperate thread to calculate CCS from given input data. Calculation will
not interfere with mainwindow
class UpdateThread(QtCore.QThread):
    def __init__(self):
        QtCore.QThread.__init__(self)

    def run(self):
        # current path of the script
        rootdir = os.path.dirname(os.path.realpath(__file__))
        # create all needed lists and dictionaries
        results = {}
        output = []
        T_list = []
        P_list = []
        datax = []
        datay = []

        # call options stands for the string, which stands behind the
CDCReader.exe and
        # describes the resolving power of the translation of the measured
data.
        # 'add' is a variable, which corresponds to the field of 'peak
sensitivity' in the main window
        # Limit corresponds to the given limit in the main window
        add = str(myGUI.peak_sens)
        Limit = float(myGUI.peak_limit)
        call_options = '--im_bin '+add
        # iterate through "path" and find all files with the wanted ending,
e.g. '*.txt.'
        # return all files with path+filename
        def findFiles (path, filter):
            for root, dirs, files in os.walk(path):
                for file in fnmatch.filter(files, filter):
                    yield os.path.join(root, file)

        # definition of the gaussian fits
        # the multiple gaussian fit just describes the sum of many 
gaussians
        #it returns the calculated y-values
        def gaussian(x, height, centre, width):
            return height * np.exp( - (x - centre)**2.0 / (2.0 * 
width**2.0) )

        def two_gaussian(x, h1, c1, w1, h2, c2, w2):
            return gaussian(x, h1, c1, w1)+gaussian(x, h2, c2, w2)

        def three_gaussian(x, h1, c1, w1, h2, c2, w2, h3, c3, w3):
            return two_gaussian(x, h1, c1, w1, h2, c2, w2)+gaussian(x, h3, 
c3, w3)

        # open file to read and split the lines to make it possible to 
iterate through lines
        # returns readable lines
        def GetLines(InputFileName):
            InputFile = open(InputFileName,'r')
            Lines = InputFile.read().splitlines()
            InputFile.close()
            return Lines

        # searching for key words in textFile = _extern file and 
calculating the drift voltage
        def GetVoltage(textFile):
            for line in GetLines(textFile):
                if line.find('ADC Pusher Frequency') >= 0:
                    PusherInterval = float(line[25:])/1000
                    #print "time:" + str(PusherInterval)
                if line.find('Helium Cell DC') >= 0:
                    Helium_Cell_DC = float(line[15:])
                    #print "He Cell DC:" + str(Helium_Cell_DC)
                if line.find('Helium Exit') >= 0:
                    Helium_Exit = float(line[15:])
                    #print "Helium Exit:" + str(Helium_Exit)
                if line.find('IMSBias') >= 0:
                    IMS_Bias = float(line[10:])
                    #print "IMS Bias:" + str(IMS_Bias)
                if line.find('Transfer DC Entrance') >= 0:
                    Transfer_DC_Entrance = float(line[22:])
                    #print "Transfer DC Entrance:" + 
str(Transfer_DC_Entrance)
            Voltage = float((Helium_Cell_DC + Helium_Exit + IMS_Bias - 
Transfer_DC_Entrance)*(1.0-2.0/170.0))
            return Voltage

        # searching for measuring time and date in text file = _header.txt 
and returns a timestamp
        def GetDate(textFile):
            for line in GetLines(textFile):
                if line.find('$$ Acquired Date:') >= 0:
                    date = line[18:]
                    #print date
                if line.find('$$ Acquired Time:') >= 0:
                    time = line[18:]
                    #print time
            return str(date+' '+time)

        # searching for matching times between measuring and Logfile
        # returning the assigned Temperature and Pressure for each
timestamp
        def GetTemp(timesearch):
            data = np.genfromtxt(str(myGUI.logfile), names=True,
dtype=("|S11", "|S5", float, "|S2", float))
            search_date = timesearch[:-9]
            search_time = timesearch[12:-3]
            match =
np.where((data['Date']==search_date)&(data['Time']==search_time))
            Temp = data[match][0][4]
            Driftgas = data[match][0][3]
            Pressure = data[match][0][2]
            return Temp, Pressure, Driftgas

        # searching for key words in textFile = _extern file and returning
        the Pusher Interval
        # the factor of 0.00025 is empirical found and describes the
        mismatch between displayed pusher interval in the files and the
        real pusher interval
        def GetPusherInterval(textFile):
            for line in GetLines(textFile):
                if line.find('ADC Pusher Frequency') >= 0:
                    PusherInterval_old = float(line[25:])/1000
                    PusherInterval = PusherInterval_old + 0.00025
                    #print "time:" + str(PusherInterval)
            return PusherInterval

        # the input x,y-values are to put in as arrays and corresponds to
the drift times and intensities of the wanted peak
        # script will try to first fit single gaussian and see the residual
error in %
        # if error is <25%, it will output the value for a single gaussian,
otherwise it will continue will multiple gaussian fits
        # the optimated mean value will be returned
        def GetDriftTime(x, y):
            try:
                # this will find the biggest peak in data and use this as
starting guess for the gauss fits
                index_max_int = np.nonzero(y == max(y))[0][0]
                total = y.sum()
                # this describes the starting 100 percent to evaluate the
residual error after each run
                one_gauss = 100
                two_gauss = 100
                three_gauss = 100

                # this loop will try out several starting points in order
to find the optimal parameters
                for i in range(int(x[index_max_int]-
20),int(x[index_max_int]+20),5):
                    errfunc1 = lambda p, x, y: (gaussian(x, *p) - y)**2
                    initial_guess1 = [max(y),i,1]
                    opt_fit1, success1 = optimize.leastsq(errfunc1,
initial_guess1[:], args =(x, y),maxfev=3000)
                    err1 = np.sqrt(errfunc1(opt_fit1, x,
y)).sum()/total*100
```

```
          # simply compare the results of each rounds and see
whats the best fit
          # each new best fit will replace the old value
          if err1 < one_gauss:
            one_gauss = err1
            best_fit = opt_fit1
          else:
            continue
        # if the best fit of single gaussian fit is good enough, it
will stop here, else multiple gaussians are fitted
        if one_gauss < 20.0:
          fit = 'single gaussian'
        else:
          for i in range(int(x[index_max_int]-
20),int(x[index_max_int]+20),5):
            errfunc2 = lambda p, x, y: (two_gaussian(x, *p) -
y)**2
            initial_guess2 = [max(y),i,1,max(y),i+10,1]
            opt_fit2, success2 = optimize.leastsq(errfunc2,
initial_guess2[:], args =(x, y),maxfev=3000)
            err2 = np.sqrt(errfunc2(opt_fit2, x,
y)).sum()/total*100
            if err2 < two_gauss:
              two_gauss = err2
              best_fit = opt_fit2
            else:
              continue
          if two_gauss < 20.0:
            fit = 'double gaussian'
          else:
            for i in range(int(x[index_max_int]-
20),int(x[index_max_int]+20),5):
              errfunc3 = lambda p, x, y: (three_gaussian(x,
*p) - y)**2
              initial_guess3 = [max(y),i,1,max(y),i+10,1,
max(y),i+20,1]
              opt_fit3, success3 = optimize.leastsq(errfunc3,
initial_guess3[:], args =(x, y),maxfev=3000)
              err3 = np.sqrt(errfunc3(opt_fit3, x,
y)).sum()/total*100
              if err3 < three_gauss:
                three_gauss = err3
                best_fit = opt_fit3
              else:
                continue
            fit = 'triple gaussian'

        # this is for the output of three drift times, even if
there is only one available drift time
        if fit == 'single gaussian':
          time1 = best_fit[1]*PushInt
          time2 = False
          time3 = False
        elif fit == 'double gaussian':
          time1 = best_fit[1]*PushInt
          time2 = best_fit[4]*PushInt
          time3 = False
        else:
          time1 = best_fit[1]*PushInt
          time2 = best_fit[4]*PushInt
```

```
                    time3 = best_fit[7]*PushInt
                return time1, time2, time3
            except ValueError:
                return False, False, False

        # voltage list x gets reversed for plotting, y-values are the drift times of the peaks
        # linregress plots a linear fit
        # Charge and Drift_Mass will be put in by the search list, T and P will be searched inside of the Logfile (GetTemp)
        # sees if the results are not real numbers (nan) and then returns the calculated CCS and the error of this value
        def GetCCS(x, y):
            x2 = [1/value for value in x]
            xaxis = ar(x2)
            yaxis = ar(y)
            slope, intercept, r_value, p_value, std_err = scipy.stats.linregress(xaxis, yaxis)
            K = (25.05**2)/(slope/1000)
            K0 = K*(273.15/T)*(P/760)
            Mass = content * Charge
            reducedMass = (Mass*Drift_Mass)/(Mass+Drift_Mass)
            CCS = (Charge/K0)*((1/(reducedMass*T))**0.5)*18495.88486
            error = (std_err/slope)*CCS
            error_CCS = (error/CCS)*100
            if math.isnan(CCS) == True or math.isnan(error_CCS) == True:
                return str(False), str(False)
            else:
                return CCS, error_CCS

        # Truncates a float f to n decimal places without rounding
        def truncate(f, n):
            s = '{}'.format(f)
            if 'e' in s or 'E' in s:
                return '{0:.{1}f}'.format(f, n)
            i, p, d = s.partition('.')
            return float('.'.join([i, (d+'0'*n)[:n]]))

        # completed is the value for the progress bar in the mainwindow
        # the '1' is just for the user to see, that the calculation has started
        myGUI.completed = 1
        self.emit(QtCore.SIGNAL('calc_progress'), myGUI.completed)

        # list_of_raw is the list of data, which is selected by the user to get calculated
        for raw in myGUI.list_of_raw:
            head, tail = os.path.split(raw)
            # find CDCReader.exe and perform conversion of data into text files
            for Exe in findFiles(rootdir, '*.exe'):
                Exe_path = os.path.join(rootdir, Exe)
                subprocess.Popen(Exe_path+' '+raw+' '+call_options, shell=True).wait()
            # find _extern.inf file and get voltage and pusher interval out of it
            for extern in findFiles(raw, '*_extern.inf'):
                Voltage = GetVoltage(extern)
                PushInt = GetPusherInterval(extern)
            # find timestamp of the measured data in the _header.txt file
            for synaptLog in findFiles(raw, '*_HEADER.TXT'):
                timestamp = GetDate(synaptLog)

            # remove useless ms.txt files which automatically get created
while converting with CDCReader
            for too_much_files in findFiles(head, '*_1_ms.txt'):
                os.remove(too_much_files)

            # calc will update the 'completed' value for the progress bar
to inform the user, that one file is finished processing
            calc = 100/len(myGUI.list_of_raw)
            myGUI.completed += calc
            self.emit(QtCore.SIGNAL('calc_progress'), myGUI.completed)

            # this will search for the temperature and pressure inside the
logfile and append it to a temporary list
            if myGUI.parameter_checkbox.isChecked():
                temperature, pressure, Drift_Mass_new  = GetTemp(timestamp)
                T_list.append(temperature)
                P_list.append(pressure)

            # get data from current file
            data = np.genfromtxt(raw+"_1_imms.txt")

            # this gets the input searchlist as variable reader
            reader = np.genfromtxt(str(myGUI.checkPeaklist))
            # see if the searchlist only has one peak (m/z and charge) in
it or more
            if reader.size == 2:
                wanted_peak = reader[0]
                charge = reader[1]
                # then search for this peak in the measured data
                low_limit = wanted_peak-Limit
                high_limit = wanted_peak+Limit
                rows=
np.where((data[:,0]>low_limit)&(data[:,0]<high_limit))
                # found peak is for the file output later
                found_peak = data[rows][0][0]
                # this is the found data to calculate CCS
                x = data[rows][:,1]
                y = data[rows][:,2]
                Drift_Time1, Drift_Time2, Drift_Time3= GetDriftTime(x, y)

                # this multiple if statements look for single or multiple
CCS (depend on the best fit of gaussian)
                if Drift_Time1 != False:
                    if found_peak not in results:
                        results[found_peak] = []
                    results[found_peak].append([Voltage, Drift_Time1,
charge])
                    if Drift_Time2 != False:
                        double_peak = found_peak+0.001
                        if double_peak not in results:
                            results[double_peak] = []
                        results[double_peak].append([Voltage, Drift_Time2,
charge])
                        if Drift_Time3 != False:
                            triple_peak = found_peak+0.002
                            if triple_peak not in results:
                                results[triple_peak] = []
                            # create the results dictionary to save the
found data for later
                            results[triple_peak].append([Voltage,
Drift_Time3, charge])
            else:
                # this is doing the same like before, only this has more
than one peak in the searchlist
                # it is not pretty, but it works :-)
                for wanted_peak, charge in reader:
                    low_limit = wanted_peak-Limit
                    high_limit = wanted_peak+Limit
                    rows=
np.where((data[:,0]>low_limit)&(data[:,0]<high_limit))
                    found_peak = data[rows][0][0]
                    x = data[rows][:,1]
                    y = data[rows][:,2]
                    Drift_Time1, Drift_Time2, Drift_Time3= GetDriftTime(x,
y)
                    if Drift_Time1 != False:
                        if found_peak not in results:
                            results[found_peak] = []
                        results[found_peak].append([Voltage, Drift_Time1,
charge])
                        if Drift_Time2 != False:
                            double_peak = found_peak+0.0001
                            if double_peak not in results:
                                results[double_peak] = []
                            results[double_peak].append([Voltage,
Drift_Time2, charge])
                            if Drift_Time3 != False:
                                triple_peak = found_peak+0.0002
                                if triple_peak not in results:
                                    results[triple_peak] = []
                                results[triple_peak].append([Voltage,
Drift_Time3, charge])

        head, tail = os.path.split(raw)


        # if progress bar has not reached 100 percent (strange update
system), this will get it to 100 percent
        myGUI.completed = 100
        self.emit(QtCore.SIGNAL('calc_progress'), myGUI.completed)
        if myGUI.parameter_checkbox.isChecked():
            if Drift_Mass_new == 'N2':
                Drift_Mass = 28
            elif Drift_Mass_new == 'He':
                Drift_Mass = 4
            # get an average temperature and pressure out of the temporary
lists
            T_arr = np.array(T_list)
            P_arr = np.array(P_list)
            T_new = np.mean(T_arr)
            if np.std(T_arr) > 1:
                for item in T_list:
                    if abs(item-T_new)>3:
                        self.emit(QtCore.SIGNAL('warning'))
            P = np.mean(P_arr)
            T = 273.15 + T_new
        else:
            # check for the drift gas in the mainwindow
            if myGUI.nitrogen_button.isChecked():
                Drift_Mass_new = 'N2'
                Drift_Mass = 28
            elif myGUI.helium_button.isChecked():
                Drift_Mass_new = 'He'
                Drift_Mass = 4
            T_new = float(myGUI.lineEdit.text())
            T = 273.15 + T_new
            P = float(myGUI.lineEdit_2.text())

        # iterate through results and append the x,y-values for single
peaks in temporary list
        # calculate CCS out of temporary x,y-list and append to output list
        # empty datax and datay list for next peak
        for content in results:
            for vol, drift, Charge in results[content]:
                datax.append(vol)
                datay.append(drift)
            CCS, error_CCS = GetCCS(datax, datay)
            if CCS != 'False' and error_CCS != 'False':
                output.append([content, CCS, error_CCS])
            datax = []
            datay = []

        # Sort list of results for m/z
        output_sorted = sorted(output, key=lambda x: (x[0]))

        # sort searchlist for m/z
        if reader.size == 2:
            mass_search = reader[0]
            charge_search = reader[1]
        else:
            search_output_sorted = sorted(reader, key=lambda x: (x[0]))

        # Write results and parameters to file.
        OutPutFileName = raw[:-6]+".summary.csv"
        # create name for open summary thread to call
        UpdateThread.OpenOutput = OutPutFileName
        # open new file to write (will always overwrite old files with same
name)
        OutPutFile = open(OutPutFileName, 'w')
        # create csv format output (again two times, one for single peak
search and one for multiple peak search)
        OutPutFile.write('Searched:,')
        OutPutFile.write(',')
        OutPutFile.write('m/z, Charge')
        OutPutFile.write('\n')
        if reader.size == 2:
            OutPutFile.write(',')
            OutPutFile.write(',')
            OutPutFile.write(str(mass_search,)+',')
            OutPutFile.write(str(charge_search,)+',')
            OutPutFile.write('\n')
        else:
            for mass_search, charge_search in search_output_sorted:
                OutPutFile.write(',')
                OutPutFile.write(',')
                OutPutFile.write(str(mass_search,)+',')
                OutPutFile.write(str(charge_search,)+',')
                OutPutFile.write('\n')

        OutPutFile.write('\n')
        OutPutFile.write('Found:,')
        OutPutFile.write(',')
        OutPutFile.write('m/z, CCS, Error in %,')
        OutPutFile.write('\n')
        for mass, CCS, error_CCS in output_sorted:
            mass_new = truncate(mass,2)
            CCS_new = format(CCS, '.2f')
            error_CCS_new = format(error_CCS, '.2f')
            OutPutFile.write(',')
            OutPutFile.write(',')
            OutPutFile.write(str(mass_new,)+',')
            OutPutFile.write(str(CCS_new,)+',')
            OutPutFile.write(str(error_CCS_new,)+',')
            OutPutFile.write('\n')

        T_new2 = format(T_new, '.2f')
        P_new = format(P, '.2f')
        OutPutFile.write('\n')
        OutPutFile.write('\n')
        OutPutFile.write('Parameter:,')
        OutPutFile.write(',')
        OutPutFile.write('Drift Gas, P (Torr), T (C),')
        OutPutFile.write('\n')
        OutPutFile.write(',')
        OutPutFile.write(',')
        OutPutFile.write(str(Drift_Mass_new,)+',')
        OutPutFile.write(str(P_new,)+',')
        OutPutFile.write(str(T_new,)+',')

        # Close the output file.
        OutPutFile.flush()
        OutPutFile.close()

# this is the maindow and the graphical user interface
class myGUI(QtGui.QMainWindow):
    def __init__(self, parent=None):
        # start GUI
        super(myGUI, self).__init__()
        # load design
        uic.loadUi('myUI_new.ui', self)
        # connect buttons to a function call
        self.logfile_check.setDisabled(True)
        self.select_logfile.setDisabled(True)
        self.connect(self.data_button,SIGNAL("clicked()"),
self.select_data)
        self.connect(self.peaklist_button,SIGNAL("clicked()"),
self.select_peaklist)
        self.connect(self.commandLinkButton,SIGNAL("clicked()"),
self.Check_before_run)
        self.connect(self.lineEdit, SIGNAL("editingFinished()"),
self.Temperature)
        self.connect(self.lineEdit_2, SIGNAL("editingFinished()"),
self.Pressure)
        self.connect(self.clear_input,SIGNAL("clicked()"), self.Clear_All)
        self.connect(self.open_summary,SIGNAL("clicked()"), self.Summary)
        self.connect(self.exitbtn,SIGNAL("clicked()"),
self.Closing_Command)
```

```
        self.connect(self.default_btn,SIGNAL("clicked()"), self.setBack)
        self.connect(self.change_btn,SIGNAL("clicked()"), self.setNew)
        self.connect(self.select_logfile,SIGNAL("clicked()"), self.Logfile)
        self.connect(self.parameter_checkbox,SIGNAL("clicked()"),
self.Change_look)
        self.x = 0
        myGUI.peak_sens = self.peak_sens.text()
        myGUI.peak_limit = self.peak_limit.text()
        myGUI.parameter_checkbox = self.parameter_checkbox

    # function calls, which the buttons are connected to
    # select to input temperature and pressure by hand or automatically
    def Change_look(self):
            self.x += 1
            if self.x%2==0:
                self.logfile_check.setAutoExclusive(False)
                self.logfile_check.setChecked(False)
                self.select_logfile.setDisabled(True)
            else:
                self.logfile_check.setAutoExclusive(False)
                self.logfile_check.setChecked(False)
                self.select_logfile.setDisabled(False)

    # open file dialog to select logfile.txt
    def Logfile(self):
        myGUI.logfile = QtGui.QFileDialog.getOpenFileName(self,"Select
Logfile", filter="*.txt")
        if myGUI.logfile != '':
            self.logfile_check.setEnabled(True)
            self.logfile_check.setChecked(True)

    # this is for new entered peak sensitivity and peaks search limit
    def setNew(self):
        myGUI.peak_sens = self.peak_sens.text()
        myGUI.peak_limit = self.peak_limit.text()

    # put search sensitivity and limit back to default
    def setBack(self):
        self.peak_sens.setText('0.3')
        self.peak_limit.setText('0.15')
        myGUI.peak_sens = self.peak_sens.text()
        myGUI.peak_limit = self.peak_limit.text()
    # this updates the progress bar
    def progress(self, int):
        self.progressBar.setValue(int)

    # this shows a warning of too big temperature differences
    def update_warning(self):
        self.errorlabel.setText('Warning!')
        self.errorlabel_2.setText('One temperature variates more than')
        self.errorlabel_3.setText('three degrees celsius from average
value!')

    # close mainwindow
    def Closing_Command(self):
        self.close()

    # open the summary of the last calculated results (available after
first run)
    def Summary(self):
        self.OpenSummary = OpenSummary()
        self.OpenSummary.start()

    # clear all fields (like new run)
    def Clear_All(self):
        self.lineEdit.setText('')
        self.lineEdit_2.setText('')
        self.lineEdit_3.setText('')
        self.lineEdit_4.setText('')
        self.progressBar.setValue(0)
        self.logfile_check.setAutoExclusive(False)
        self.logfile_check.setChecked(False)
        self.logfile_check.setDisabled(True)
        self.errorlabel.setText('')
        self.errorlabel_2.setText('')
        self.errorlabel_3.setText('')

    # open file dialog to select multiple directories (the data set)
    def select_data(self):
        dialog = FileDialog(self)
        try:
            self.lineEdit_4.setText(str(len(dialog.raw_data)))
            myGUI.list_of_raw = dialog.raw_data
        except AttributeError:
            self.lineEdit_4.setText('False')

    # open file dialog to select the search list with the wanted peaks in
it (m/z and charge)
    def select_peaklist(self):
        adress = QtGui.QFileDialog.getOpenFileName(self,"Select
Peaklist",filter="*.txt")
        head, tail = os.path.split(str(adress))
        if adress == '':
            self.lineEdit_3.setText('Please enter valid peaklist')
        else:
            self.lineEdit_3.setText(tail)
        myGUI.checkPeaklist = adress

    # field for temperature
    def Temperature(self):
        text = self.lineEdit.text()

    # field for pressure
    def Pressure(self):
        text2 = self.lineEdit_2.text()

    # if 'run' is pressed, the script will first check the entered values
    def Check_before_run(self):
        check_data = self.lineEdit_4.text()
        if check_data == '':
            self.lineEdit_4.setText('False')
            d = False
        elif check_data == 'False':
            d = False
        else:
            d = True
        check_peak = self.lineEdit_3.text()
        if check_peak == '':
            self.lineEdit_3.setText('Please enter valid peaklist')
            e = False
        elif check_peak == 'Please enter valid peaklist':
            e = False
        else:
            e = True
        if myGUI.parameter_checkbox.isChecked() == False:
            try:
                temp = float(self.lineEdit.text())
                self.errorlabel.setText('')
                a = True
            except ValueError:
                self.errorlabel.setText('Check the temperature!')
                a = False
            try:
                press = float(self.lineEdit_2.text())
                self.errorlabel_2.setText('')
                b = True
            except ValueError:
                self.errorlabel_2.setText('Check the pressure!')
                b = False
            if self.nitrogen_button.isChecked() == True or
self.helium_button.isChecked() == True:
                c = True
                self.errorlabel_3.setText('')
            else:
                self.errorlabel_3.setText('Check the drift gas!')
                c = False

            # if all entered values seem to be ok, the script will continue
running the main calculation
            if a == True and b == True and c == True and d == True and e ==
True:
                self.completed = 0
                myGUI.completed = self.completed
                myGUI.lineEdit_3 =self.lineEdit_3
                myGUI.nitrogen_button = self.nitrogen_button
                myGUI.helium_button = self.helium_button
                myGUI.lineEdit = self.lineEdit
                myGUI.lineEdit_2 = self.lineEdit_2
                self.CCS_Calculation()
        else:
            if self.x%2!=0:
                if self.logfile_check.isChecked() == False:
                    f = False
                    self.errorlabel.setText('Please enter valid Logfile!')
                else:
                    f = True
                    self.errorlabel.setText('')
            if d == True and e == True and f == True:
                self.errorlabel.setText('')
                self.errorlabel_2.setText('')
                self.errorlabel_3.setText('')
                self.CCS_Calculation()
    # this is the main calculation and will be run by a seperate thread to
prevent freezing and stuff
    def CCS_Calculation(self):
        self.UpdateThread = UpdateThread()
        self.connect(self.UpdateThread,SIGNAL("calc_progress"),
self.progress)
        self.connect(self.UpdateThread,SIGNAL("warning"),
self.update_warning)
```

```
        self.UpdateThread.start()

# this is the file dialog to select multiple directories
# it looks complicated because QWidget FileDialog is used to choose
multiple files, not multiple directories like we need
class FileDialog(QtGui.QFileDialog):
    def __init__(dialog, parent=myGUI):
        QtGui.QFileDialog.__init__(dialog)
        dialog.setFileMode(dialog.Directory)
        dialog.setOption(dialog.ShowDirsOnly, True)
        dialog.setFilter("Raw Data (*.raw)")
        for view in dialog.findChildren((QtGui.QListView,
QtGui.QTreeView)):
            if isinstance(view.model(), QtGui.QFileSystemModel):

view.setSelectionMode(QtGui.QAbstractItemView.MultiSelection)
        #this will create the list of data files we will process later
        if dialog.exec_():
            dialog.filenames = dialog.selectedFiles()
            dialog.raw_data = map(str, dialog.filenames)
            # this solves a problem of other path entries in the list than
the data directories .raw we need
            for fnames in dialog.raw_data:
                if fnames[-4:] != '.raw':
                    dialog.raw_data.remove(fnames)
        dialog.show()

def main():
    app = QtGui.QApplication(sys.argv)
    myapp = myGUI()
    myapp.show()
    app.exec_()

if __name__ == "__main__":
    main()
```

***Algorithm 3***: *Main code to communicate with the user, accommodate the data and process it.*

```
import serial
import time
import datetime
import binascii

def Temp():
    device2 = serial.Serial(
            port = "COM4",
            baudrate = 9600,
            parity = serial.PARITY_NONE,
            stopbits = serial.STOPBITS_ONE,
            bytesize = serial.EIGHTBITS, timeout = 0)
    device2.write('\x55\x09\x11\x15\x58\x12\x34\x14\x09\x40')
    temperature = ''
    time.sleep(0.2)
    while device2.inWaiting() > 0:
        temperature += device2.readline()
    device2.close()
    output = binascii.b2a_hex(temperature)
        byte10 = output[-4:-2]
        byte11 = output[-6:-4]
        if byte10 == '01':
                temp_first = 255
        else:
                temp_first = int(byte10,16)
        temp_second = int(byte11,16)
        temp_ready = "%.1f" % float((temp_first+temp_second-2)/10.0)
        if temp_ready == "-0.2":
                temp_ready = "0"
    return temp_ready

with serial.Serial(
        port = "COM3",
        baudrate = 9600,
        parity = serial.PARITY_ODD,
        stopbits = serial.STOPBITS_ONE,
        bytesize = serial.EIGHTBITS, timeout = 0) as device:

    while True:
        device.write("PR <cr> \r")
                output = ''
        time.sleep(0.2)
        while device.inWaiting() > 0:
            output += device.readline().strip('\n')
        device.write("GC 1 R <cr> \r")
        gas = ''
        time.sleep(0.2)
        while device.inWaiting() > 0:
            gas += device.readline().strip('\n')
        device.close()
        try:
            temperature = Temp()
        except serial.serialutil.SerialException:
            time.sleep(1)
            temperature = Temp()

        pressure = "%.2f" % abs(float(output)/100)
                if int(gas) == 100:
                        driftgas = "N2"
                elif int(gas) == 145:
                        driftgas = "He"
                else:
                        driftgas = "Error"
        timestamp = datetime.datetime.now().strftime("%d-
")+datetime.datetime.now().strftime("%B")[:3]+datetime.datetime.now().strft
ime("-%Y %H:%M")
        #print timestamp,' ',pressure,' ',temperature, driftgas
        with open('C:\Users\Administrator\Desktop\Logfile.txt', 'a') as
log:
            log.write('\n')
            log.write(timestamp)
            log.write(' ')
            log.write(pressure)
            log.write(' ')
            log.write(driftgas)
            log.write(' ')
            log.write(temperature)
        with open('C:\Projects\Logfile.txt', 'a') as log:
            log.write('\n')
            log.write(timestamp)
            log.write(' ')
            log.write(pressure)
            log.write(' ')
            log.write(driftgas)
            log.write(' ')
            log.write(temperature)
        time.sleep(59.5)
        try:
            device.open()
        except serial.serialutil.SerialException:
            time.sleep(1)
            device.open()
```

***Algorithm 4****: Side script to automatically read and store external parameters.*

```
import sys
from PyQt4 import QtCore, QtGui, uic
import serial
import time
import datetime
import binascii

qtCreatorFile = "temp_press_display.ui"
Ui_MainWindow, QtBaseClass = uic.loadUiType(qtCreatorFile)

class SetPressure(QtGui.QMainWindow):
    def __init__(self, parent=None):
        super(SetPressure, self).__init__()
        uic.loadUi('change_press.ui', self)
        self.connect(self.start_change,QtCore.SIGNAL("clicked()"),
self.accept_pressure)

    def accept_pressure(self):
        try:
            value = float(self.write_pressure.text())
            self.error_label.setText('')
            self.new_value = value
            self.send_value()
        except ValueError:
            self.error_label.setText('Please insert a valid number')

    def send_value(self):
        self.emit(QtCore.SIGNAL('press'), self.new_value)
        self.close()

class UpdateThread(QtCore.QThread):
    def __init__(self):
        QtCore.QThread.__init__(self)

    def run(self):
        try:
            device = serial.Serial(
                port = "COM3",
                baudrate = 9600,
                parity = serial.PARITY_ODD,
                stopbits = serial.STOPBITS_ONE,
                bytesize = serial.EIGHTBITS,
                                timeout = 0)

            while True:
                device.write("PR <cr> \r")
                output = ''
                time.sleep(0.5)
                while device.inWaiting() > 0:
                    output += device.readline().strip('\n')
                device.close()
                out = "%.2f" % abs(float(output)/100)
                if len(str(out)) == 0:
                    self.emit(QtCore.SIGNAL('Error1'))
                else:
                    self.emit(QtCore.SIGNAL('Press_changed'), out)
                time.sleep(4.5)
                try:
                    device.open()
                except serial.serialutil.SerialException:
                    time.sleep(1)
                    device.open()
            self.emit(QtCore.SIGNAL('Error1'))
        except serial.serialutil.SerialException:
            self.emit(QtCore.SIGNAL('Error1'))

class UpdateThread2(QtCore.QThread):
    def __init__(self):
        QtCore.QThread.__init__(self)

    def run(self):
        try:
            device2 = serial.Serial(
                port = "COM4",
                baudrate = 9600,
                parity = serial.PARITY_NONE,
                stopbits = serial.STOPBITS_ONE,
                bytesize = serial.EIGHTBITS,
                timeout = 0)

            while True:
                device2.write('\x55\x09\x11\x15\x58\x12\x34\x14\x09\x40')
                temp = ''
                time.sleep(0.5)
                while device2.inWaiting() > 0:
                    temp += device2.readline()
                device2.close()
                if len(str(temp)) == 0:
                    self.emit(QtCore.SIGNAL('Error2'))
                else:
                                        output = binascii.b2a_hex(temp)
                                        byte10 = output[-4:-2]
                                        byte11 = output[-6:-4]
                                        if byte10 == '01':
                                                temp_first = 255
                                        else:
                                                temp_first = int(byte10,16)
                                        temp_second = int(byte11,16)

                                        temp_ready = "%.1f" %
float((temp_first+temp_second-2)/10.0)
                                        if temp_ready == "-0.2":
                                                temp_ready = "0"
                    self.emit(QtCore.SIGNAL('Temp_changed'), temp_ready)
                time.sleep(4.5)
                try:
                    device2.open()
                except serial.serialutil.SerialException:
                    time.sleep(1)
                    device2.open()
            self.emit(QtCore.SIGNAL('Error2'))
        except serial.serialutil.SerialException:
            self.emit(QtCore.SIGNAL('Error2'))

class MyApp(QtGui.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)
        # enable custom window hint
        self.setWindowFlags(self.windowFlags() |
QtCore.Qt.CustomizeWindowHint)
        # disable (but not hide) close button
        # self.setWindowFlags(self.windowFlags() &
~QtCore.Qt.WindowCloseButtonHint)
        self.setWindowFlags(self.windowFlags() &
~QtCore.Qt.WindowMaximizeButtonHint)
        self.UpdateThread = UpdateThread()
        self.UpdateThread2 = UpdateThread2()
        self.connect(self.UpdateThread2, QtCore.SIGNAL('Temp_changed'),
self.Update_Temp)
        self.connect(self.UpdateThread, QtCore.SIGNAL('Press_changed'),
self.Update_Press)
        self.connect(self.UpdateThread, QtCore.SIGNAL('Error1'),
self.Error_Press)
        self.connect(self.UpdateThread2, QtCore.SIGNAL('Error2'),
self.Error_Temp)
        self.connect(self.pressure_button,QtCore.SIGNAL("clicked()"),
self.set_pressure)
        self.UpdateThread.start()
        self.UpdateThread2.start()

    def Update_Temp(self, str):
        self.lcd_number.display(str)
    def Update_Press(self, str):
        self.lcdNumber.display(str)
    def Error_Press(self):
        self.lcdNumber.display('Error')
    def Error_Temp(self):
        self.lcd_number.display('Error')
    def set_pressure(self):
        self.Pressure = SetPressure(self)
        self.connect(self.Pressure, QtCore.SIGNAL('press'), self.change)
        self.Pressure.show()

    def change(self, float):
        try:
            device = serial.Serial(
                port = "COM3",
                baudrate = 9600,
                parity = serial.PARITY_ODD,
                stopbits = serial.STOPBITS_ONE,
                bytesize = serial.EIGHTBITS, timeout = 0)
            pre_number = "0"+str(int(float*100))
                        if len(pre_number) == 3:
                                number = "0"+pre_number
                        elif pre_number == "00":
                                number = "00"+pre_number
                        else:
                                number = pre_number
            print("PS"+" "+number+" <cr> \r")
            device.write("PS"+" "+number+" <cr> \r")
        except serial.serialutil.SerialException:
            time.sleep(1)
            device = serial.Serial(
                port = "COM3",
                baudrate = 9600,
                parity = serial.PARITY_ODD,
                stopbits = serial.STOPBITS_ONE,
                bytesize = serial.EIGHTBITS, timeout = 0)
            pre_number = "0"+str(int(float*100))
                        if len(pre_number) == 3:
                                number = "0"+pre_number
                        elif pre_number == "00":
                                number = "00"+pre_number
                        else:
                                number = pre_number
            print("PS"+" "+number+" <cr> \r")
            device.write("PS"+" "+number+" <cr> \r")

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = MyApp()
    window.show()
    sys.exit(app.exec_())
```

***Algorithm 5***: *Visual interface script to allow parameter control and changes in value.*