Business Studies Department

Bachelor in Applied Computer Science

Application Development

Analyzing Flow Cytometry Data Using KNIME Workflows

CAMPUS

Geel

Kevin Deyne

Academy year 2010-2011

PREFACE

This thesis was created during a 3-month internship at Janssen Pharmaceutica together with 3 other students. I thank the other students: Kris Melis, Toon Borgers and Veerle Verheyen. Thanks to them this project became a proper team effort.

We achieved a lot during the internship, and this would have not been possible without assistance. For leading the project, guiding the development and particularly for properly explaining the scientific context of the project to us in an understandable way, I would like to thank my mentor Frans Cornelissen.

Furthermore I would like to thank the developers, Frederick Michielssen and Ari De Groot who worked side by side with us on the application for their guidance and their always available helping hand. Their expertise allowed us to deal with problems we would have had a hard time solving.

This thesis was an effort that required some steering. Much thanks to my supervising lecturer Christine Smeets, whose tips and remarks improved the structure and content of this thesis.

SUMMARY

This thesis is an extensive report of an internship at Janssen Pharmaceutica.

Scientists at Janssen Pharmaceutica perform routine experiments on large amounts of cells using a technique that irradiates cells with laser light. This technique is called Flow Cytometry, and is explained in detail in the introduction.

The goal of the internship application was to automate the data analysis of many of these routine experiments. The existing Flow Cytometry machine is able to process the cells and save all its data into files. These files are used as input for the application.

Essentially, we created *nodes.* Nodes are modular units with a very specific task in mind. For instance, it is possible to have a node which handles the conversion of data into a table, or a node which visualizes its input into a graph. These different nodes can be connected to create complex structures. These complex structures, also called *workflows*, aid in the analysis performed on the data.

The final goal was to fully integrate our application with the existing research platform.

The internship application uses the KNIME tool. KNIME is an easy to use program which contains all the nodes and allows them to be connected. The tool is open-source, and thus allowed us to expand its existing array of nodes with our own creations.

The nodes themselves are programmed in Java 1.6, making use of the KNIME Framework. The existing analysis program for Janssen, with which we integrated our internship application, is also completely written in Java. This overarching system makes use of the RCP (Rich Client Platform) Framework. All persistent data is stored in an Oracle database.

LIST OF FIGURES

LIST OF ABBREVIATIONS

FCM	Flow Cytometry
IDE	Integrated development environment
PHAEDRA	Plate-based High-Content Analysis, Evaluation and Dynamic Reliability Assurance
KNIME	Konstanz Information Miner
SVN	Subversion
CVS	Concurrent Versions System
KDE	Kernel Density Estimation
BLOB	Binary Large Object
CLOB	Character Large Object
UI	User Interface
FSC	Front/Forward scatter
SSC	Side scatter
FCS	Flow Cytometry Standard format. Files generated by the Flow Cytometry device (raw data).
ACS	Analytical Cytometry Standard format. A collection file comparable to a .ZIP format, which contains all data for a plate. This collection often contains FCS files (raw data), Gating-ML files (applied gates) and a Table of Content (XML format)

LIST OF CODE SNIPPETS

TABLE OF CONTENTS

1 INTRODUCTION

This thesis describes an internship project which lasted 3 months at the Belgian pharmaceutical company Janssen Pharmaceutica, where we created an application to automate certain routine research.

The first chapter provides more information about the company, its overarching group and the department in which we worked. This should give you a good insight in to why this type of application was created.

During the second chapter we will explain in detail what the concept behind the entire application is. It contains information about what particular branch of research we aim to automate and the general goal of the application. We will explain the present situation, our assignment, the primary stakeholders and other interested parties.

The document will continue with more information about the software which supported our work during the internship.

The following chapter will detail my personal additions to this project. As we worked in a team, I will try to provide an overview of where within the application these additions fit.

Finally, more details will be provided on how this application was integrated into the existing overarching research application.

The document ends with a conclusion, containing a self-reflection on my 3 month period and the final result of my work.

2 JANSSEN PHARMACEUTICA AND JNJ

In this chapter we will provide you with a background of the company the internship project was created for - Janssen Pharmaceutica, its overarching group Johnson & Johnson and the department which we worked for. This will provide you with a contextual background on why the subject of the internship was chosen.

2.1 About the company

Figure 1 - Janssen Pharmaceutica Logo

In order to put the thesis in the correct context, it is interesting to put the beneficiaries into the spotlight. The project is directly written for Janssen Pharmaceutica, a well-known Belgian pharmaceutical company.

Janssen Pharmaceutica was started in 1953 by Doctor Paul Janssen, lending the company its name.

Right from the start, the main focus of the company was not the production of drugs. Instead, he chose to put the main focus on researching new cures and developments in the medical sector.

Having produced more than 80 new medical drugs, Janssen Pharmaceutica is considered one of the most innovative medical companies in the world.

The company has its headquarters in Beerse, with other sites located in Olen, Geel and Merksem. Furthermore there are sites in the People's Republic of China (Xi'an) and America.

The medical sector is broad, and so are the different amount of therapeutic areas Janssen researches.

Their research is not limited to a few sectors, instead including sectors such as:

- Neurosciences
- Infectious diseases
- Immunology
- Cancer research
- Cardiovascular conditions

In the future, the company aims to retain its world leader position in three main areas:

- Research into substances for psychiatry
- Neurology

- Infectious diseases

In the spirit of the company's goal of constantly innovating its research, the project aims to enable scientists to easily perform analysis on large quantities of cells.

2.2 About Johnson & Johnson

In 1961, the Janssen Pharmaceutica company was acquired by Johnson & Johnson, one of the world's largest pharmaceutical companies.

Figure 2 - Johnson & Johnson logo

Johnson & Johnson is an American pharmaceutical, medical devices and consumer packaged goods manufacturer founded in 1886. It was founded by Robert Wood Johnson, and its headquarters are in New Brunswick, New Jersey.

Their activities are organized in three business segments:

Consumer health care
A broad range of consumer health and personal care products in the beauty, baby, oral care and women's health categories, as well as nutritional products and over-the-counter medicines and wellness and prevention platforms.

Medical devices and diagnostics
The medical devices and diagnostics segment focuses on technologies, solutions and services in the fields of cardiovascular disease, diabetes care, orthopaedics, vision care, wound care, aesthetics, sports medicine, infection prevention, minimally invasive surgery, and diagnostics.

Pharmaceuticals
The Pharmaceutical segment's broad portfolio focuses on unmet medical needs across several therapeutic areas: oncology; infectious disease; immunology; neuroscience; cardiovascular and metabolism. It includes products in the anti-infective, antipsychotic, cardiovascular, contraceptive, dermatology, gastrointestinal, haematology, immunology, neurology, oncology, pain management, urology and virology fields.

http://www.jnj.com/connect/about-jnj/company-structure/

2.3 About the department

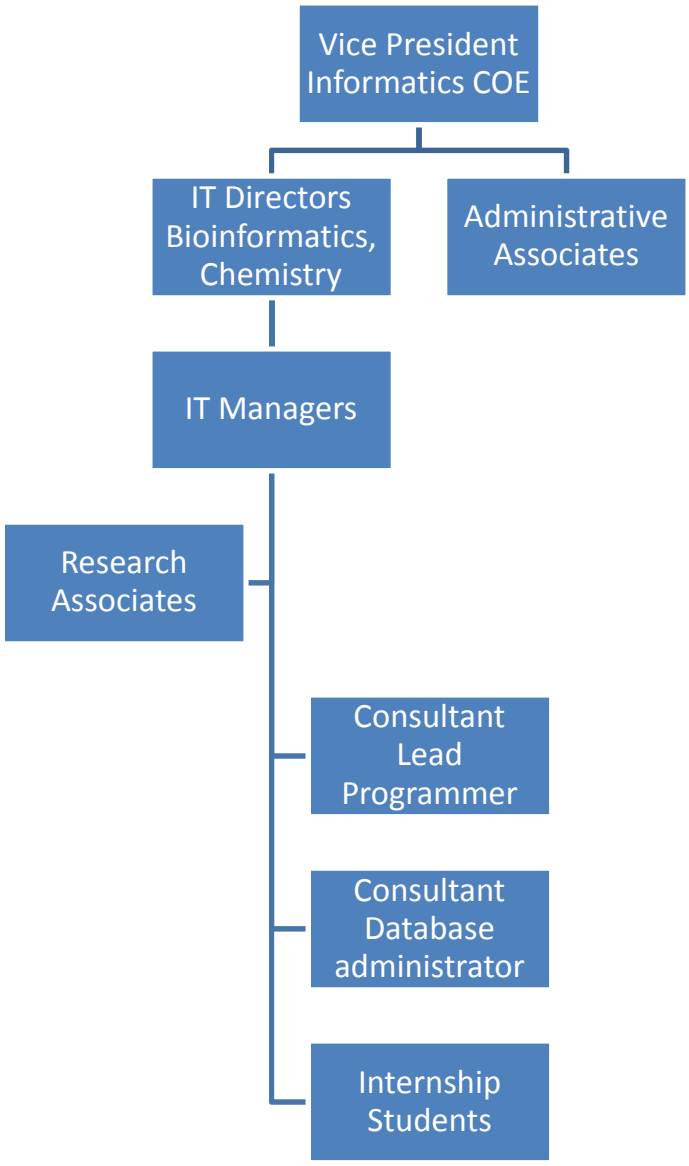
During the internship, we were a part of the department Informatics Centre of Excellence, often abbreviated as COE.

Obviously focused on IT development, this department is a collection of Janssen employees and external consultants. IT Managers and project leaders are often from the company itself, while the programmers themselves are a mixture of internals and externals.

For instance, we often worked with Frederick Michielssen, a consultant of Ordina. We also worked together with Ari De Groot, a Dutch freelance consultant.

Our superior, carrying the title of IT Manager, is an employee of Janssen and is the main coordinator of our project.

Below you will find a hierarchical breakdown of the department.



Hierarchical breakdown of Informatics COE

3 INTERNSHIP PROJECT

In this chapter, we will give you an overview on the context of our work and the goal of our internship.

Furthermore, this chapter will concentrate on elements of the plan of approach we created. You will find more information on the project assignment, the background and who will benefit from and actually utilise our project result.

3.1 Background of our assignment

In this section, some background information is provided on the research technique *Flow Cytometry*, the used file types and the existing PHAEDRA application.

3.1.1 Flow Cytometry research

Our internship project was situated within a medical research context.

During medical research, one of the techniques used to receive a large amount of data from microscopic particles, often blood cells, is by irradiating them with light. With this method, data can be gathered from thousands of cells at a time.

Essentially, all the different cells are inserted into a fluid. This fluid makes it easy to transport the cells through the device that will shoot the laser light and receive the data.

When the fluid, and thus the cells, enters the machine, it is guided through the device by a thin tube. The tube guides the cells past the laser light source. This is demonstrated in the picture below.

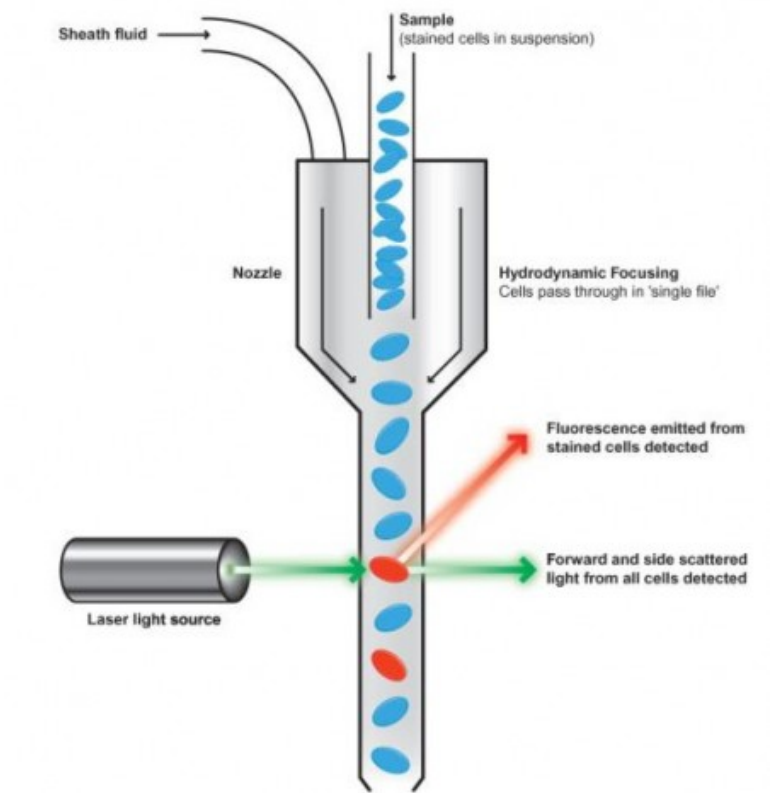


Figure 3 - Flow Cytometry Example

When the laser light reaches a cell, the device will capture data about how the light reacts with the cell. Generally, there are two types of data received.

The first type of data received is how the light scatters from the cells. This produces **F**orward- and **S**ide **sc**atter values (abbreviated as FSC and SSC). Forward or Front scatter gives the scientists information about the size of the cell. Side scatter tells them more about the internal structure.

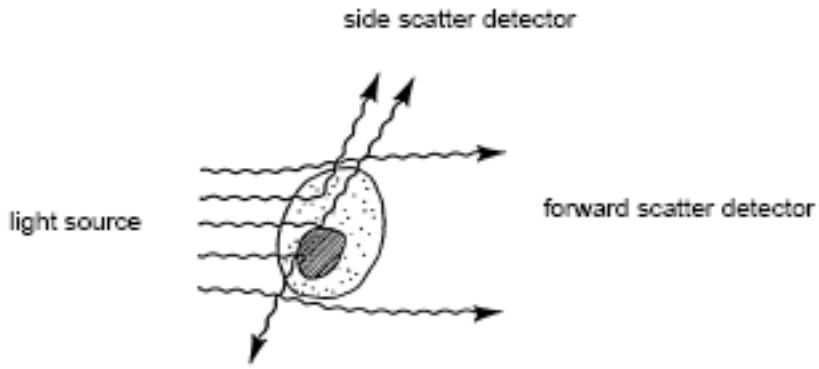


Figure 4 - Front scatter (FSC) and side scatter (SSC)

A second type of data received is the *fluorescence* of the cells. Some cells can start emitting light on their own, as long as they are excited with laser light of a certain wavelength.

This whole process of gathering data on how laser light reacts to the cells passing through a thin tube is called *Flow Cytometry.* It is often abbreviated as FCM.

3.1.2 Plates/Wells

The aforementioned cells that the FCM device reads are not processed all at once. The cells are grouped together in small pools of similar cells. These small pools are called *wells.*

These wells are also grouped together on a flat plastic called a *plate.* This is illustrated in the image below.

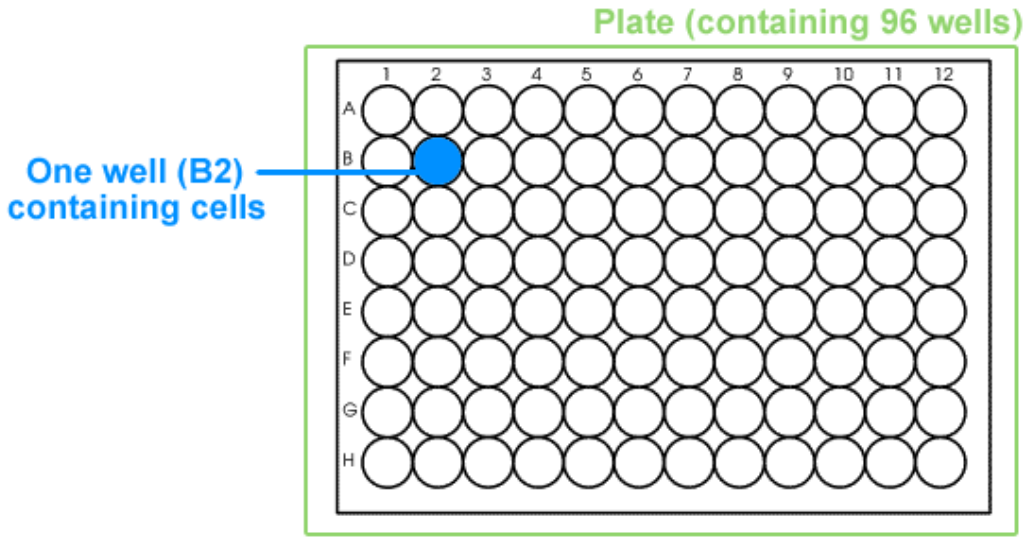


Figure 5 - Visual explanation of the cell / well / plate hierarchy

An overview of all terms can be found in the PHAEDRA drilldown on figure 13, though most fall outside the scope of this thesis. By knowing the distinction between a plate and a well, you should be able to continue with no problem.

As said, a plate (full name: microtiter plate) is a flat plastic plate with multiple wells (small test tubes). In these wells, biologists insert cells and reagents that are allowed to interact. After they have reacted sufficiently, they are taken out, and passed through a Flow Cytometry machine.

Common plate sizes are 96 and 384 well plates. The wells are commonly arranged in a 2:3 rectangular matrix. Below you can find an image of how these plates actually look.

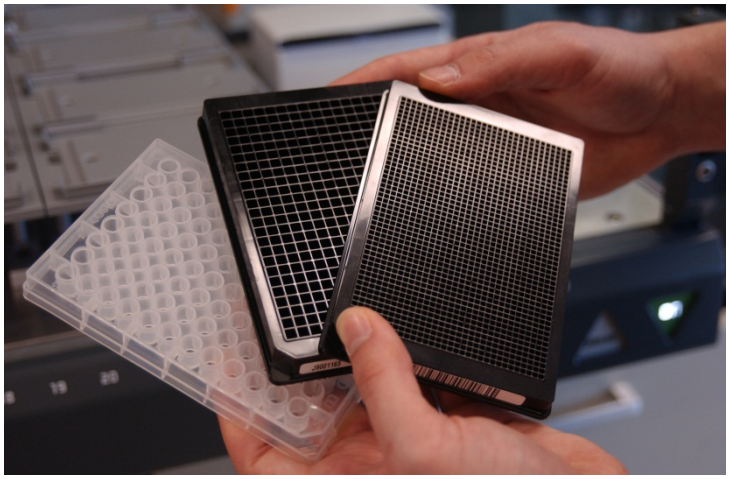


Figure 6 - Example of plates with respectively 96, 384 and 1536 wells

3.1.3 FCS

Once the Flow Cytometry device has processed the cells, it will write its results into a file. Each well will get a separate file. These files are built up according to a standard maintained by the International Society for Analytical Cytology (ISAC).

These basic raw data files are called *FCS* files.

FCS stands for Flow Cytometry Standard. It describes how Flow Cytometry results are to be stored in files. The first version was adopted in 1984, and the current version (FCS3.0) was proposed in 2004.

An FCS file consists of various segments:

- Header
- Text
- Data
- Analysis
- CRC
- Other

The header segment describes the locations of the other segments in the file.

The text segment consists of key-value pairs, such as the amount of values in the file, or the operator of the Flow Cytometry instrument.

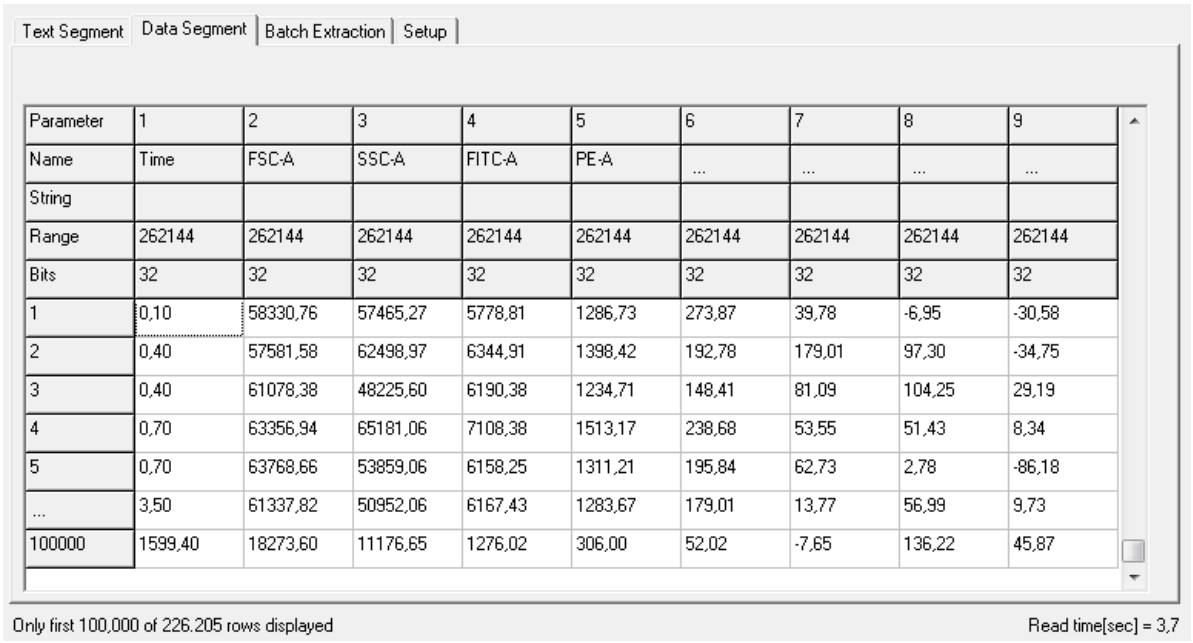
The data segment contains the actual data values.

The optional analysis segment describes the results of any analysis done on the data. It is typically appended sometime after the collecting of the results, when the files are analyzed by researchers.

The CRC segment contains a checksum to see if the FCS file is valid.

The optional other segment is freely filled with other information.

An example of such an FCS file can be seen below. Take note that FCS files are unreadable by normal text processors and require to be parsed first.


Text Segment | Data Segment | Batch Extraction | Setup

Parameter	1	2	3	4	5	6	7	8	9
Name	Time	FSC-A	SSC-A	FITC-A	PE-A	...	...	...	...
String									
Range	262144	262144	262144	262144	262144	262144	262144	262144	262144
Bits	32	32	32	32	32	32	32	32	32
1	0,10	58330,76	57465,27	5778,81	1286,73	273,87	39,78	-6,95	-30,58
2	0,40	57581,58	62498,97	6344,91	1398,42	192,78	179,01	97,30	-34,75
3	0,40	61078,38	48225,60	6190,38	1234,71	148,41	81,09	104,25	29,19
4	0,70	63356,94	65181,06	7108,38	1513,17	238,68	53,55	51,43	8,34
5	0,70	63768,66	53859,06	6158,25	1311,21	195,84	62,73	2,78	-86,18
...	3,50	61337,82	50952,06	6167,43	1283,67	179,01	13,77	56,99	9,73
100000	1599,40	18273,60	11176,65	1276,02	306,00	52,02	-7,65	136,22	45,87

Only first 100,000 of 226.205 rows displayed

Read time[sec] = 3,7

Figure 7 - An example of the contents of an FCS file (using mock data). Notice that previously mentioned terms such as Front Side Scatter (FSC) return here as well.

3.1.4 ACS

While the FCS files provide the raw data needed, each file only contains well data. When we want to work on a plate level, we need to collect several of these files.

This is why we use another file type, which acts as a container for FCS files. We call this file type *ACS*.

ACS stands for Archival Cytometry Standard. It is a container for Flow Cytometry-related files. This means it can not only contain well data in FCS format, but also other data. It is based on the ZIP file specification, and contains one or more TOC (table of contents) files, describing the contents of the container.

An example of the contents such an ACS file can be seen in the image below.

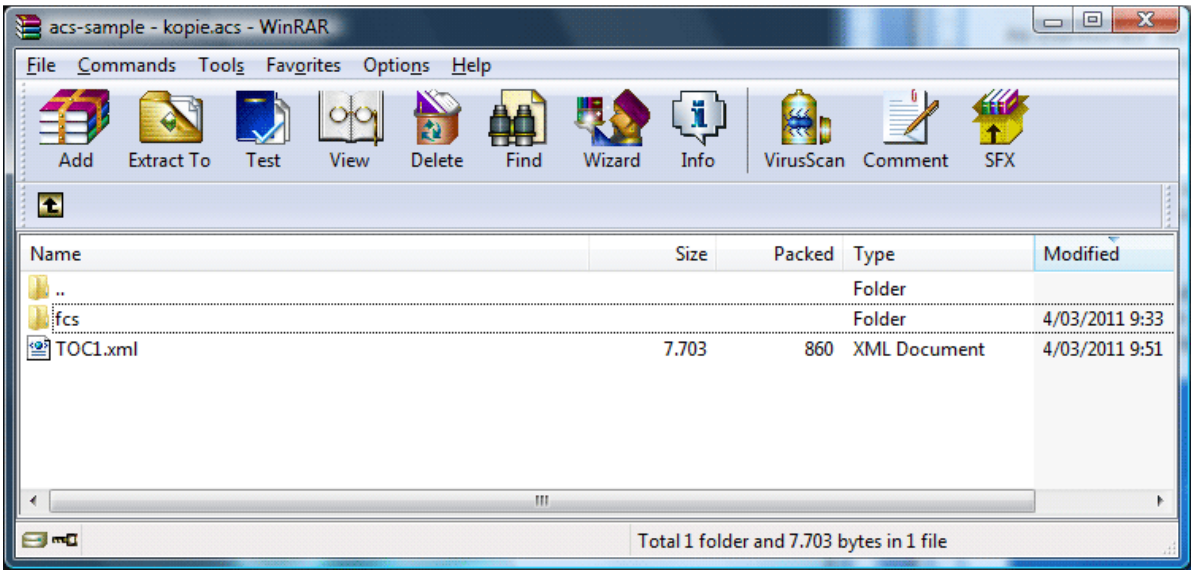


Figure 8 - Example of the contents within an ACS file. Here you can see the table of contents and a map containing FCS raw data.

As seen in the above image, we opted to use a clean structure with maps within the ACS files. Raw data contained in the FCS map is clearly separated from other files that might be added to the ACS at a later point.

Within the FCS map we will find all the raw data, as demonstrated in the image below.

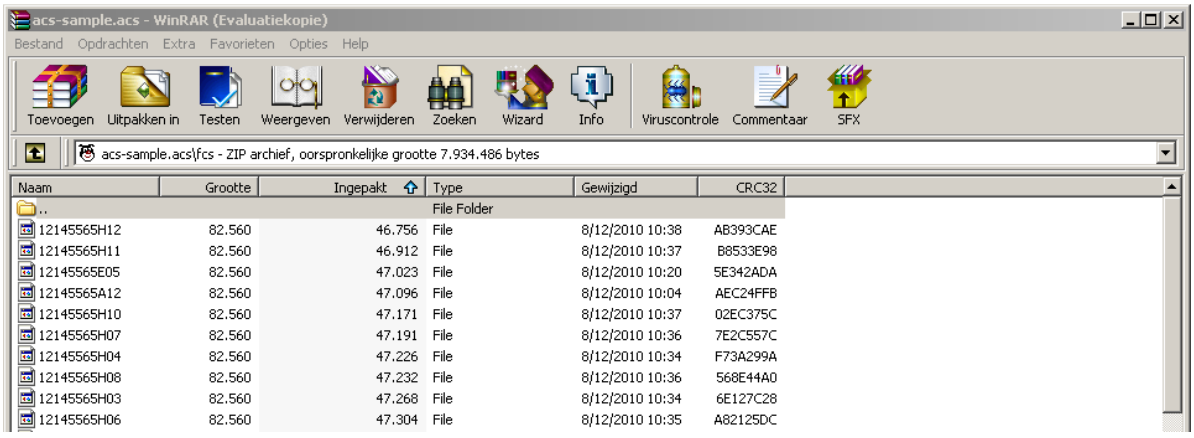


Figure 9 - Example of the contents within an ACS file. All files seen here are FCS raw data files.

The amount of raw data files clearly show the purpose of a container file used to keep all corresponding files together. Furthermore, this allows developers and biologists alike to maintain a clear structure within the data.

The table of content files are used by our application to navigate quickly through these container files. The TOC files are saved as XML. They follow naming convention TOC*n*.xml, with *n* being a number from 1 to the number of TOCs in the ACS file. At Janssen Pharmaceutica, we decided to use only one TOC file per ACS file.

```
<?xml version="1.0" encoding="UTF-8"?>
<toc:TOC ... >
      <toc:file toc:URI="file:///Well_A01.fcs" toc:description="well#1" />
      <toc:file toc:URI="file:///Well_A02.fcs" toc:description="well#2" />
</toc:TOC>
```

Code snippet 1 - Example ACS TOC file

In the above example, you can see the TOC from an ACS file that has 2 files in it. These two files are FCS files containing the results for well 1 and well 2.

3.1.5 Gates and Gating-ML

3.1.5.1 Gates

Another important concept often referred to within a Flow Cytometry context is *gating*.

Data is always represented in certain graphs. While the type of graph used will differ depending on the type of data being analysed, graphs are always used to find interesting areas within the plotted data. These interesting areas will often manifest themselves as clusters or peaks.

During analysis, these interesting areas are then marked by the scientist. Such a selection is called a *gate*. The process of applying gates to data is called *gating.*

A clear example of a simple cluster within a scatter plot can be seen in the figure below. A thick cluster is clearly visible. The dense area has been selected and a gate is applied. In this case, all elements within this gate are marked blue.

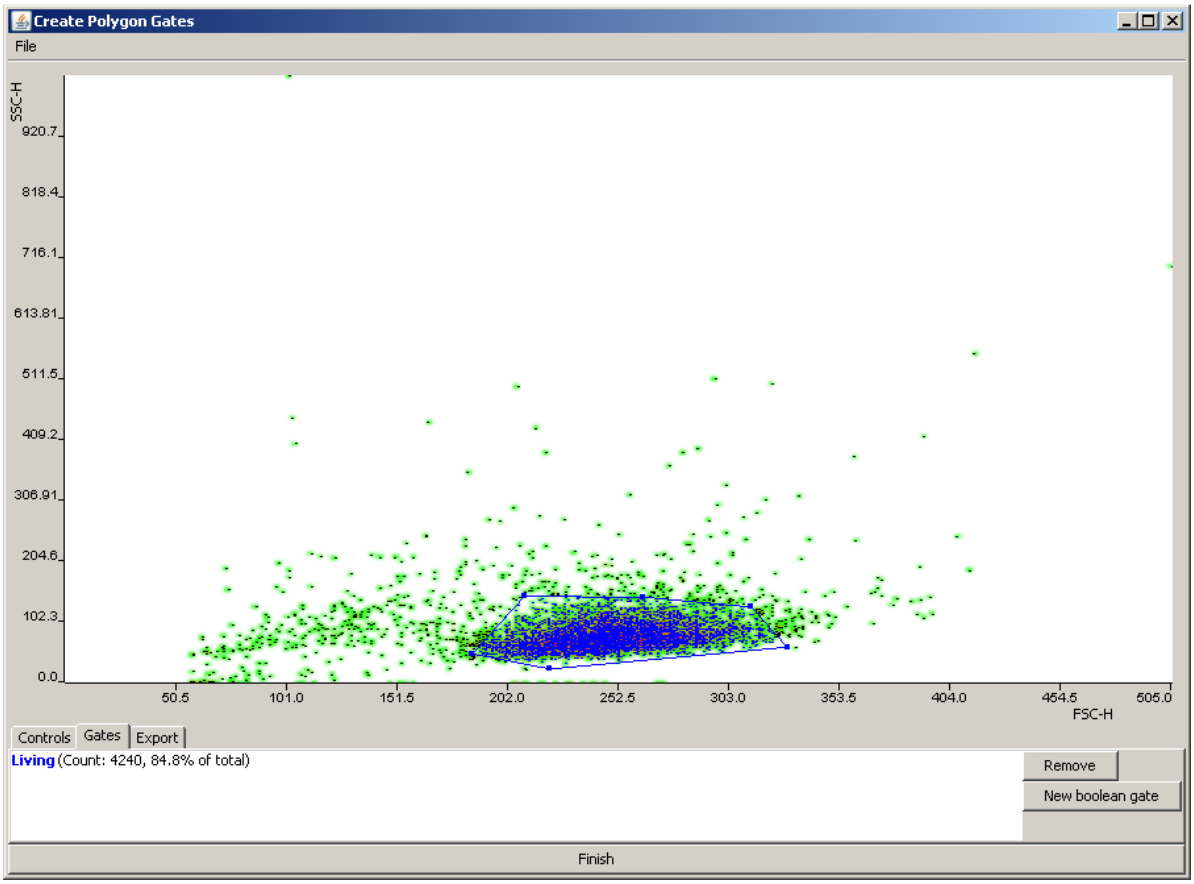


Figure 10 - Scatter plot example with a defined gate

During the workflows chapter, there will be one node in particular (1D Range Gating) which uses gating extensively. These concepts will be briefly refreshed there and applied to the graph specifically used there.

3.1.5.2 Gating-ML

The above mentioned *gates* are then saved within a file for later use. Essentially, they are saved as an XML document, though it has a very clearly defined structure.

An XML document following the strict convention is called a *Gating-ML* file.

This is a standard, developed by the International Society for Analytical Cytology (ISAC). Since Gating-ML is based on XML, the files are saved in an easy to use ASCII format.

The official standard describes many different kinds of gates. All of these perform a selection, as explained before. They differ in their implementation of such a selection. Different methods of implementation also mean that the file structure will be different.

Several of these selection methods will be discussed below, including:

- Polygon gates
- Range gates
- Boolean gates

Polygon gates

The gate shown before is the basic gate known as the *Polygon Gate*. Its name comes from the method of selection. A series of points are defined which, when connected with each other, will form a polygon. All data that falls within the polygon shape is counted among the gate its selected data.

An example is shown in the figure below:

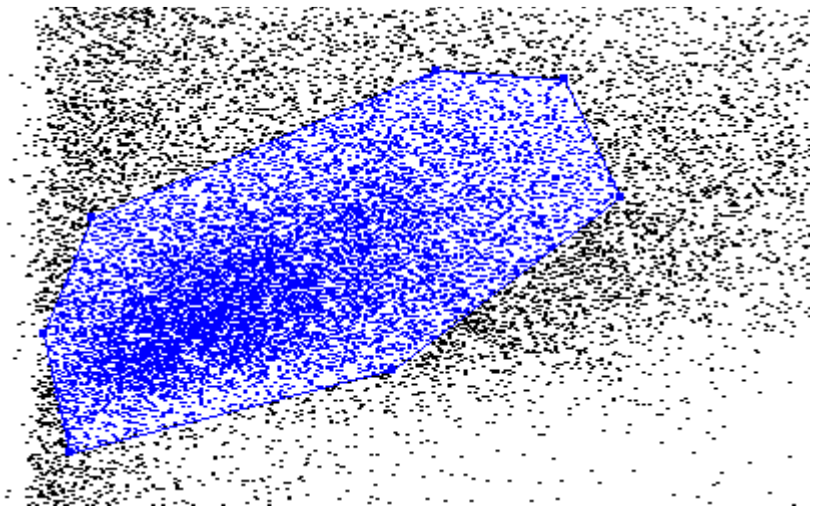


Figure 11 - Polygon gate example

The XML file this gate will generate will first define on which X and Y values the plot was created. Because this type of gate is mostly used within a 2D representation, you will most likely find two such parameters set.

The file will then continue to list all points used by defining the X and Y coordinate.

The code snippet below shows an example of the file such a gate:

```
<gating:PolygonGate gating:id="Gate example">
      <gating:dimension>
            <data-type:parameter data-type:name="FSC" />
      </gating:dimension>
      <gating:dimension>
            <data-type:parameter data-type:name="SSC" />
      </gating:dimension>
      <gating:vertex>
            <gating:coordinate data-type:value="0" />
            <gating:coordinate data-type:value="0" />
      </gating:vertex>
      <gating:vertex>
            <gating:coordinate data-type:value="400" />
            <gating:coordinate data-type:value="0" />
      </gating:vertex>
      <gating:vertex>
            <gating:coordinate data-type:value="400" />
            <gating:coordinate data-type:value="300" />
      </gating:vertex>
</gating:PolygonGate>
```

Code snippet 2 - Gating-ML Polygon Gate example of a polygon using three points

This polygon gate describes a polygon with three points on the FSC-SSC plane with coordinates (0;0), (400;0) and (400;300) for its points, with an id "Gate example".

Rectangle gates

Within a 1D environment, it is often less interesting to create polygon like shapes. Instead, ranges will be used to define what data is counted among the gate its population.

A rectangle gate is described by ranges on 1 or more dimensions. These ranges must either have a minimum value, a maximum value, or both. On 1D graphs, they are referred to as range gates.

If a gate has only a minimum value, its maximum value can be interpreted as positive infinity. Likewise, if a gate only has a maximum value, its minimum value can be interpreted as negative infinity.

In the below image you can see three rectangle gates defined. One ranging from -∞ to a specific coordinate, from one specific coordinate to another coordinate and from one coordinate to +∞.

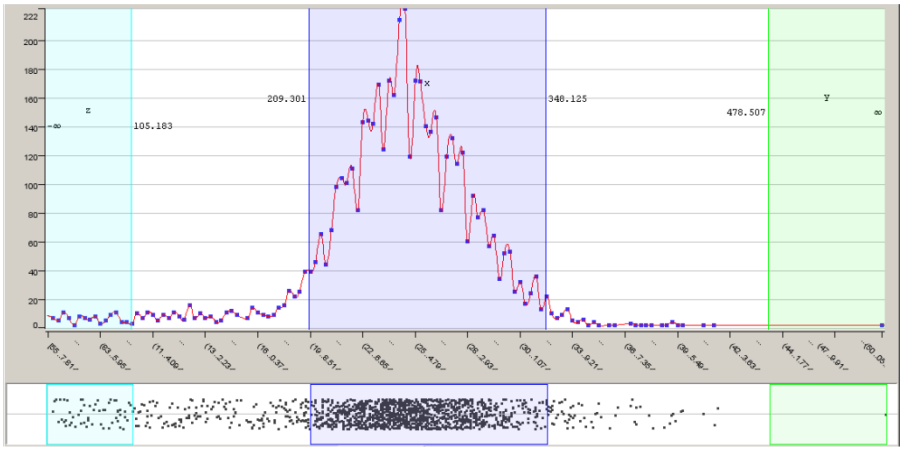


Figure 12 - An example of three defined rectangle gates

The code snippet below shows an example Gating-ML rectangle gate:

```
<gating:RectangleGate gating:id="Rectangle">
      <gating:dimension gating:min="100" gating:max="250">
            <data-type:parameter data-type:name="FSC" />
      </gating:dimension>
      <gating:dimension gating:min="50">
            <data-type:parameter data-type:name="SSC" />
      </gating:dimension>
</gating:RectangleGate>
```

Code snippet 3 - Gating-ML Rectangle Gate

This file contains two gates. On the FSC dimension, values from 100 to 250 are selected. On the SSC dimension, only the minimum value of 50 is defined. This means that the range goes from 50 to +∞.

Boolean gates

These gates are defined using logical operations on other gates. There are 3 possible operations:

- AND: Takes the intersection of the referred gates
- OR: Takes the union of the referred gates
- NOT: Takes everything but the referred gate.

The code snippet below shows an example Gating-ML Boolean gate:

```
<gating:BooleanGate gating:id="AndGate">
  <gating:and>
    <gating:gateReference gating:ref="Rectangle" />
    <gating:gateReference gating:ref="Triangle" />
  </gating:and>
</gating:BooleanGate>
```

Code snippet 4 - Gating-ML Boolean Gate

This gate describes the intersection of the gates with ids “Rectangle” and “Triangle”.

3.1.6 The overarching PHAEDRA System

As mentioned before, the primary directive of Janssen is research. The current, overarching system in place for supporting data analysis is called ***PHAEDRA***.

The name is actually an acronym of its full description, which is **P**late-based **H**igh-Content **A**nalysis, **E**valuation and **D**ynamic **R**eliability **A**ssurance.

Basically, PHAEDRA is a supportive system which handles data analysis. It starts off with data from plates which can be numerical or high-resolution JPEG2000 images, or both.

Using this raw data, scientists can drill down to analyze data from more specific areas of the plates. Below you can see the structure the system uses to provide data and analysis capabilities on all possible levels.

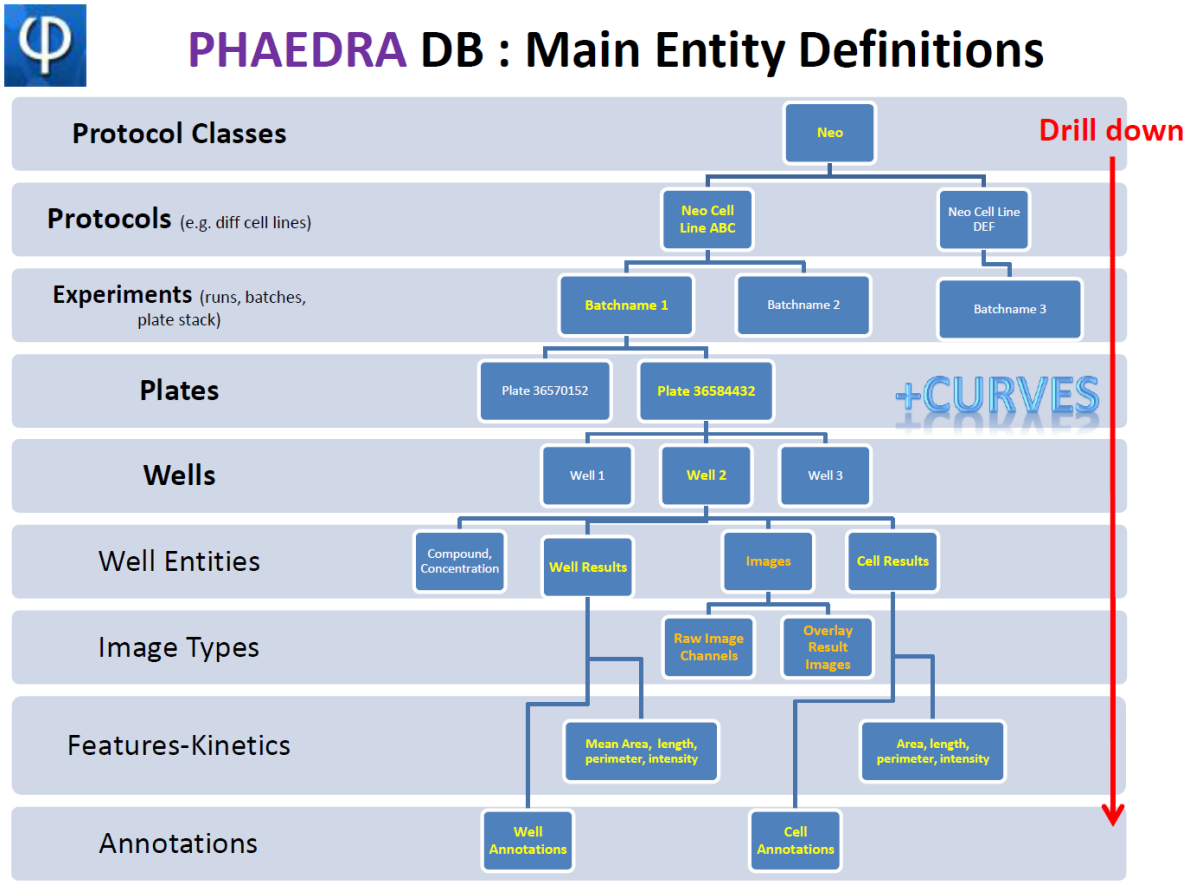


Figure 13 - PHAEDRA Drilldown

The goals of PHAEDRA in general are to provide a seamlessly integrated system with a full-drilldown capacity. It is not limited to one single type of research, but attempts to provide a platform usable in many research contexts.

The integration of our application with PHAEDRA allows it to provide Flow Cytometry analysis on top of its other features. More information on the integration itself can be found in the final chapter.

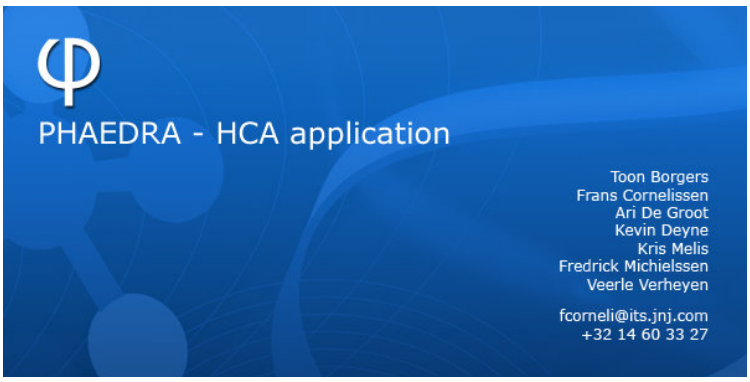


Figure 14 – PHAEDRA Splash Screen, with our names among the developers.

3.2 What was the goal of our internship?

The goal was to develop a brand new application, to automate the *Flow Cytometry* routine research explained before. This was accomplished by building an extension to an existing data-mining platform called *KNIME*. The extension had to allow scientists to

use the platform to automate the analysis of the data gathered by a Flow Cytometry device.

KNIME (short for Konstanz Information Miner) is an open-source platform created mainly for Data Mining. It is often used by chemo- and bioinformatics companies to process and analyze their data.

It works in a user-friendly way by giving the user access to a large collection of *nodes*.

Nodes are modular units with a very specific task. For instance, it is possible to have a node which handles the conversion of data into a table, or a node which visualizes its input into a graph. A node can receive input or process output, or both. It can also create a 'view', which is a visual form of output.

An example of a node can be seen in the picture below. The node itself has various sorts of *ports* available. Ports are entrances and exits to the node itself. Ports on the left side are ports which can receive input data, ports on the right side are this node's generated output. Normal data ports are visualised as seen below.

Figure 15 - An example of a node with one input and one output port. The yellow colour work as a visual aid to understand what role the node has. In this case, we are dealing with a node that manipulates data.

These different nodes can be coupled together to create complex structures. A connection is made by linking one node's port to another node's port. These complex structures, also called *workflows*, follow the same steps as a regular analysis performed on the data, though entirely automated. An example of a workflow can be seen in the picture below. More information on how these are built up will follow in a later chapter.

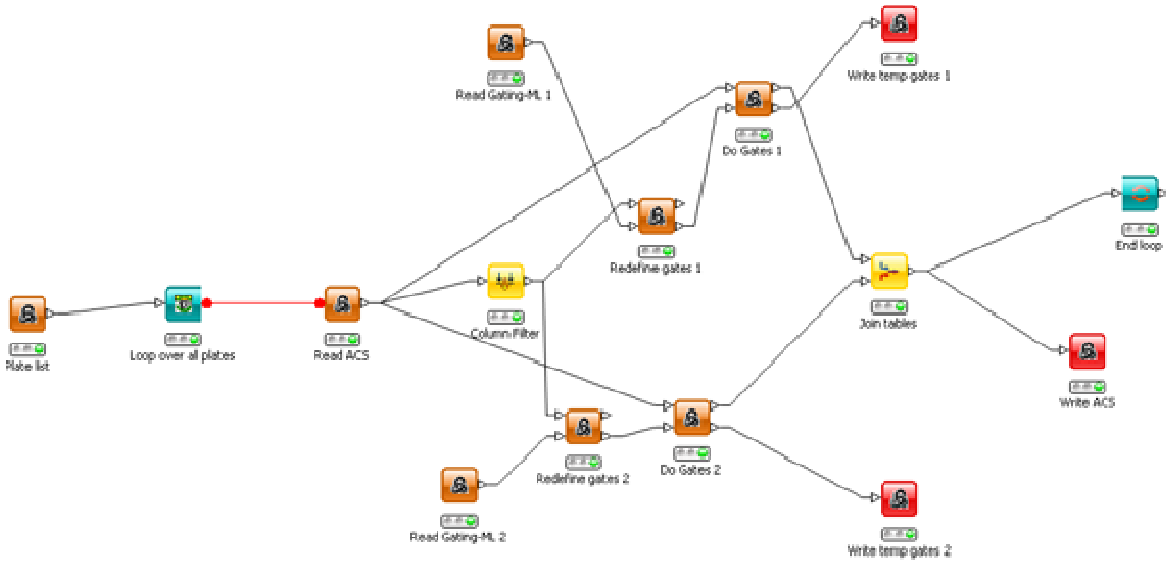


Figure 16 - A workflow example

Because the KNIME tool is open-source, we were able expand the existing array of nodes within the framework with our own creations.

For instance, we were required to expand the KNIME Node repository with new nodes which provide the functionality to import data, perform gating, subsetting and analysis and to save this data into the PHAEDRA database and relevant files.

Specifics on which nodes were created and how they fit into the big picture will be discussed in a later chapter.

Other students were in charge of expanding this application even further with reporting and implementing the statistical language 'R' into the KNIME project.

The final goal of our internship was to create this application and integrate it with the existing *PHAEDRA* application. PHAEDRA is built and tuned completely for biomedical data research, however its scope ranges far beyond just Flow Cytometry.

We never worked directly on PHAEDRA itself. Instead, we created applications for the KNIME platform also discussed earlier. A PHAEDRA programmer working closely with our team then published our creations at certain intervals and provided the possibility to start up our KNIME workflows with a simple click from within PHAEDRA.

More information on the result of the integration can be found in the final chapter of this thesis.

During the internship, our work focused on creating the different nodes which provided the functionality required to perform certain analysis.

3.3 Primary target audience

The people who will be using our application in the first place basically exist out of two main groups.

The biologists using the Flow Cytometry research method will be using the PHAEDRA system to support them, and will thus use our KNIME nodes during their analysis.

However, they will not work with them on a node-level. Instead, they will be presented a workflow, a complex structure of nodes, that requires specific input. This input will be provided by the PHAEDRA system. The workflow will then automatically perform its task, occasionally prompting the scientist with a user interface.

The second group of primary users are the PHAEDRA developers. The node system allows for the re-use of specific functionality, but in a completely different workflow. In other words, the individual nodes can be connected to each other in different ways depending on the required analysis procedure.

These workflow variations will be created by PHAEDRA developers, coupling different nodes together to get the result that is required.

As is the case with our internship application, these workflows get 'published' to PHAEDRA, integrating it with the overarching system and thus allowing the biologists to use it in their analysis.

3.4 Other stakeholders

The strength of using a node system like the one in KNIME lies within the re-use of the individual elements in different variations of workflows. While some nodes are particularly created to work with PHAEDRA, a majority of our nodes have completely independent functionality.

As such, Janssen is interested in supporting the KNIME community by providing them with our Flow Cytometry nodes.

As the KNIME community is actually already extensively used by medical research companies, this would provide them with useful additional tools.

3.5 Business Case

Janssen is not the first group to work with Flow Cytometry. As such, there is already FCM software available.

By creating their own FCM software, Janssen saves money on yearly licenses. The estimates for these are at € 1.500 per year, per user. The group working with FCM software consists of 10 biologists, resulting in a cut cost of € 15.000 a year.

More importantly, by developing their own system, Janssen has the ability to tailor it specifically to their experiments. Furthermore, they are able to integrate it with their existing PHAEDRA system containing data validation and dose-response filtering. This results in a large amount of time saved.

4 SUPPORTING SOFTWARE

During our internship we used various software to support our development. We will explain these tools some further, including:

- Eclipse
- Subversion
- KNIME Desktop Client
- FACEJava

4.1 Eclipse

Figure 17 - Eclipse logo

We designed the nodes using Eclipse. Eclipse is a software development environment which consists of an IDE (integrated development environment) and a plug-in system to add new features.

Eclipse is updated every year with a new version, the latest being Indigo (released June 2011). It is an ideal development environment, as it supports many servers and additions.

It can be used to develop software in many different languages, though it is most commonly used as a Java IDE. To include support for other languages, users can simply add the plug-ins for this language, or download an Eclipse version custom made for this language.

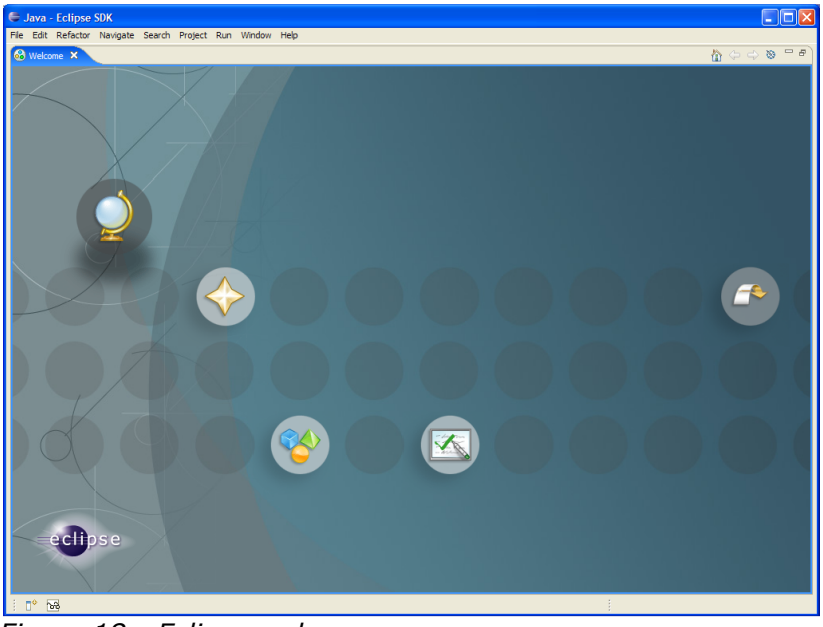


Figure 18 - Eclipse welcome screen

4.1.1 Visual concepts

4.1.1.1 Views and editors

Visually, Eclipse is made up out of components called *views* and *editors*. Use the image depicted below to easily understand each concept.

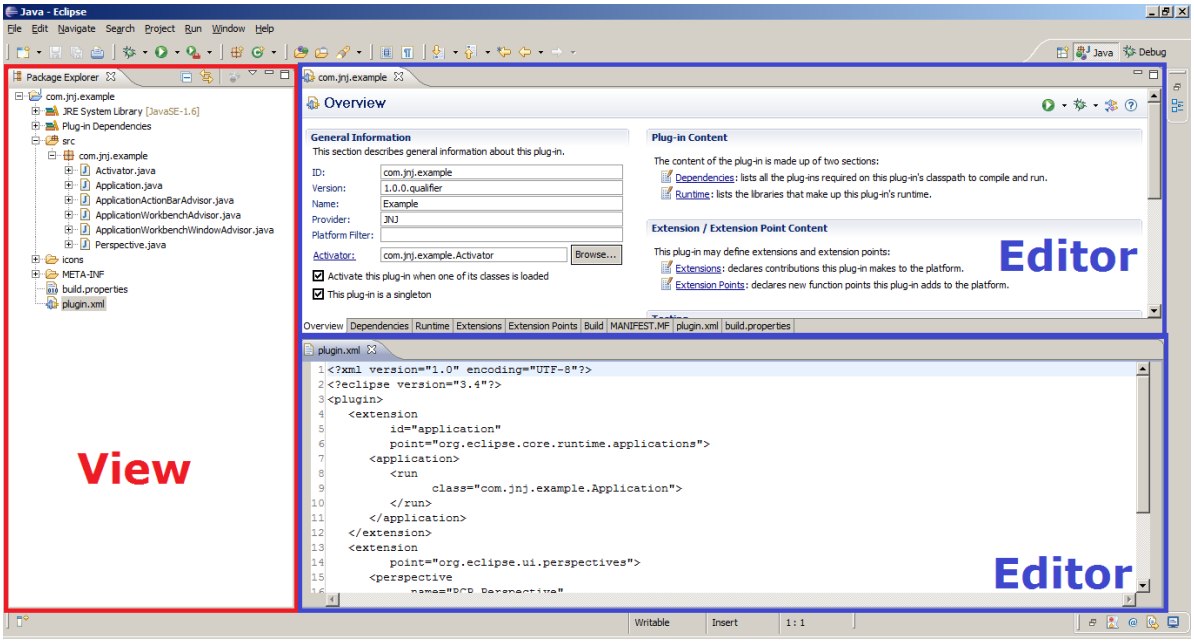


Figure 19 - Views and editors

A *view* is a window part which can serve different purposes, such as displaying a tree of the files of a project, or a list of errors in the application.

An *editor* is a component used to edit files. In the figure, the same file is being edited using two different editors.

The editor on top is called a *graphical editor*, and uses a set of controls like textboxes, checkboxes and comboboxes to edit files.

The bottom editor is called a *textual editor* and it displays the raw text file. Changes to a file in one editor are shown immediately in the other editors editing that same file.

4.1.1.2 Perspectives

A third important concept is the *perspective*. A perspective consists of different views and editors, and is used to define a GUI for a specific purpose.

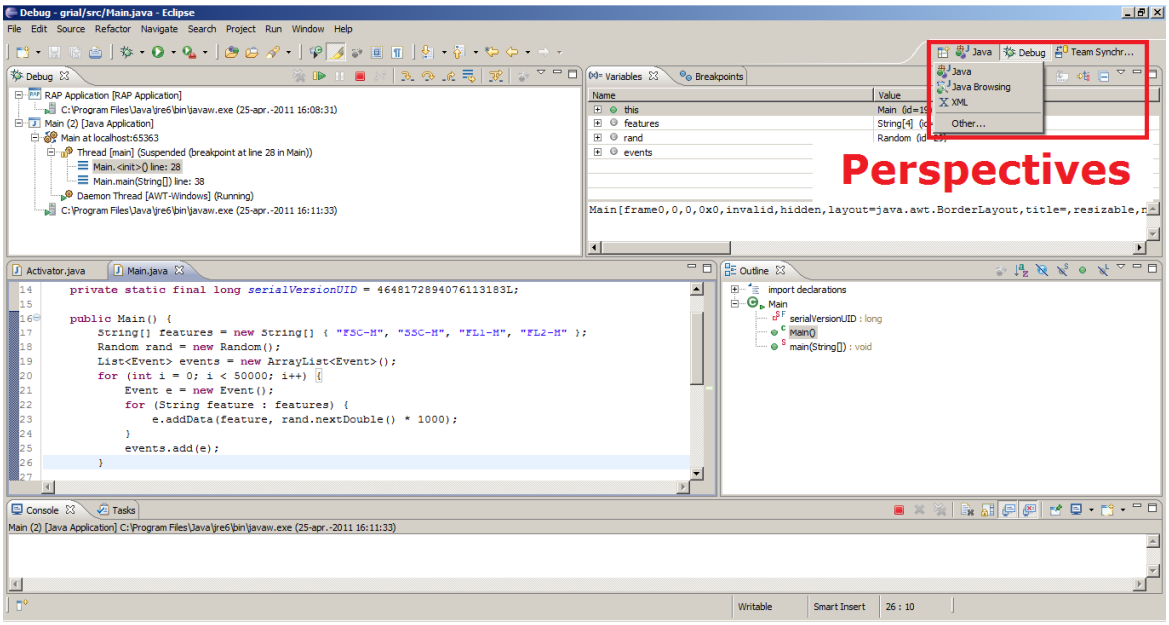


Figure 20 - Perspectives

In the above figure, the active perspective is the debug perspective, which is used to find errors (bugs) in your code by letting the execution pause at certain lines of code using breakpoints. Different views are shown, such as a view showing the current variables and their values, and a view showing the current line of code.

In the right upper corner of the Eclipse IDE, you can choose a perspective to open using a menu, and you can put your favourite perspectives on a toolbar for quick access.

4.1.2 Plug-ins

Plug-ins can contain many different things, such as third party libraries, or support for a new programming language.

The most important plug-in we used is Subclipse. It is a very intuitive plug-in that adds subversion capabilities to the Eclipse development environment. Subclipse itself will be further explained in part 4.2.2.

The process of installing a new plug-in is very simple; click 'Help > Install new software…' and the window shown below pops up. In the top textbox, you can enter a source for the plug-in. Eclipse will then search the entered site for available plug-ins and list the available options.

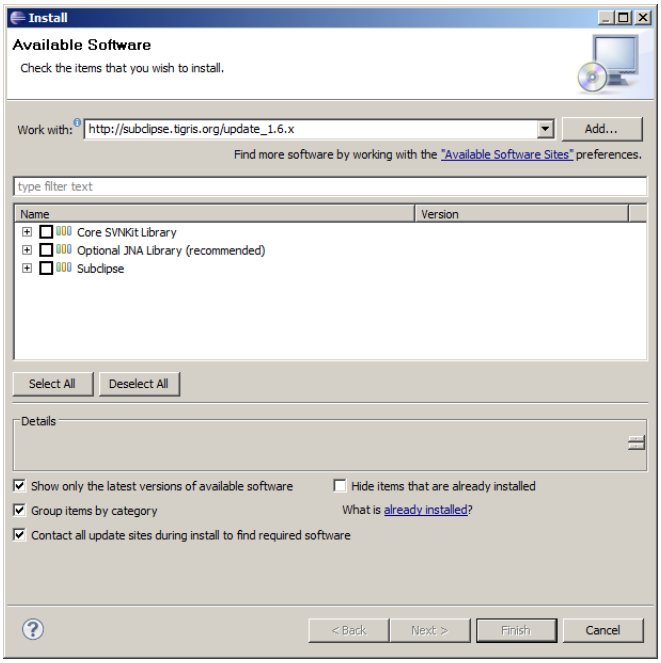


Figure 21 - Install new software

You can then select which software should be added to Eclipse. Eclipse will then check for additional plug-ins required by the plug-in and list them (see figure below).

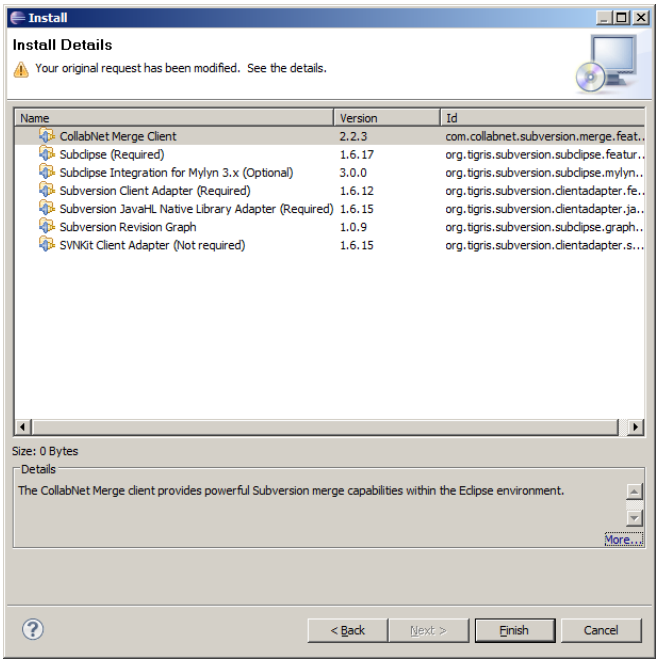


Figure 22 - Additional software

When you click the *Finish* button, the new software is downloaded and installed. Once that process is completed, Eclipse prompts you to either restart Eclipse immediately, or do not restart. For some plug-ins, it is recommended to restart immediately, while others do not have such a big impact and do not require an immediate restart.

4.2 Subversion

Figure 23 - Subversion logo

As mentioned in the previous section, the most important plug-in we used during development was Subclipse. This is a plug-in providing direct *Subversion* functionality within the development environment.

Subversion (SVN) is an open source versioning control system used to maintain the current version of code and its previous versions. You can think of it as a "time machine" which holds all changes to all code files within your project.

It was created to be an improvement on the de facto standard of the time, CVS (Concurrent Versions System). CollabNet, an American software company, started the development in 2000, and by the end of August 2001, it was "self-hosting", which means its source code was kept on a SVN repository.

4.2.1 Terminology

The location where the files are kept is called a "repository", or "repo". The latest version on the repository is called the "Head", and the initial version is the "Base" version. A "conflict" happens when you want to commit a change to a file, and someone else has made a change to that file since you last updated it or checked it out, and your versions have changed the same lines of code.

There is also a list of commands to add and retrieve files to and from a repository:

- Commit: Add your new version to the repository
- Checkout: Get file(s) from the repository
- Update: Update your local files to the latest version from the repository
- Diff: Show the differences between your file and a version from the repository
- Resolved: Let the repository know that you have solved a conflict

4.2.2 Subclipse

Subclipse is the SVN plug-in for Eclipse that we used during the development of our nodes. It is very lightweight and easy to install, and integrates nicely with Eclipse.

After installation, you can add a repository and either share a project, or checkout one from the repository. Subclipse places icons next to the files in the package explorer to indicate the status of the files:

 Modified file

 Missing file

 Deleted file

 New file

 A file that is up to date with the repository

4.3 KNIME Desktop

Figure 24 - KNIME Logo

As mentioned before, all of our work is created within the KNIME framework and can be used as nodes within KNIME. The user will interact with these nodes we create in a user-friendly application called the KNIME Desktop Client. In our case, the KNIME client does not work as a stand-alone application, but can be opened as a perspective from within PHAEDRA.

4.3.1 User interface

The Desktop Client is built up as a typical *RCP* Application. RCP stands for Rich Client Platform applications, an Eclipse framework which allows for the creation of applications which use user interface elements of Eclipse itself. Even updating and installing new items works in the exact same way as Eclipse.

As discussed in the visual concepts sections of the Eclipse chapter, this basically means the desktop client has familiar elements such as views and perspectives.

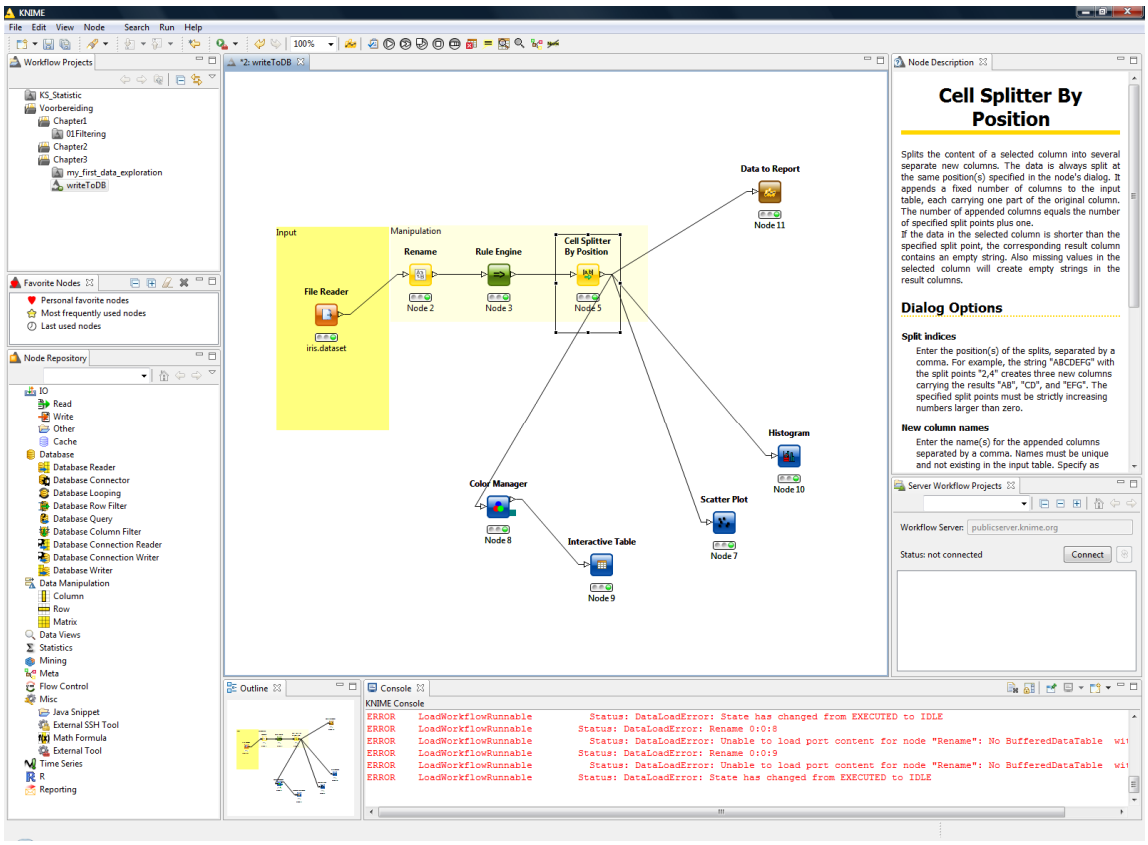
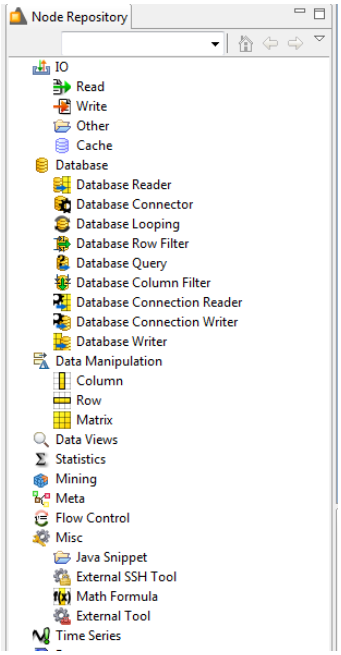


Figure 25 - KNIME Desktop Client User Interface

4.3.2 Node Repository



When the user starts the Desktop Client for the first time, his first step will be to build a workflow with nodes.

He can find these nodes in the *Node Repository*. As the name suggests, this is the entire collection of available nodes within the client.

The nodes have their own icons and are divided into categories. These categories generally categorize the nodes with similar functionality together.

If custom nodes are installed into KNIME, these will have their own categories so that they are easy to find.

Our nodes are similarly categorized under a 'JNJ' category, and further categorised under their functionality, such as Gating or Transformations.

This view also has a search bar above the listing of nodes. This search bar is able to filter out irrelevant nodes quickly.

Figure 26 - Node Repository

4.3.3 Workflow Editor

When a node is selected, it can be dragged to the centre of the client. By selecting the node ports, it is possible to connect nodes with each other, forming a complex whole.

By right-clicking a node, its menu can be opened. From there, the node can be configured or executed.

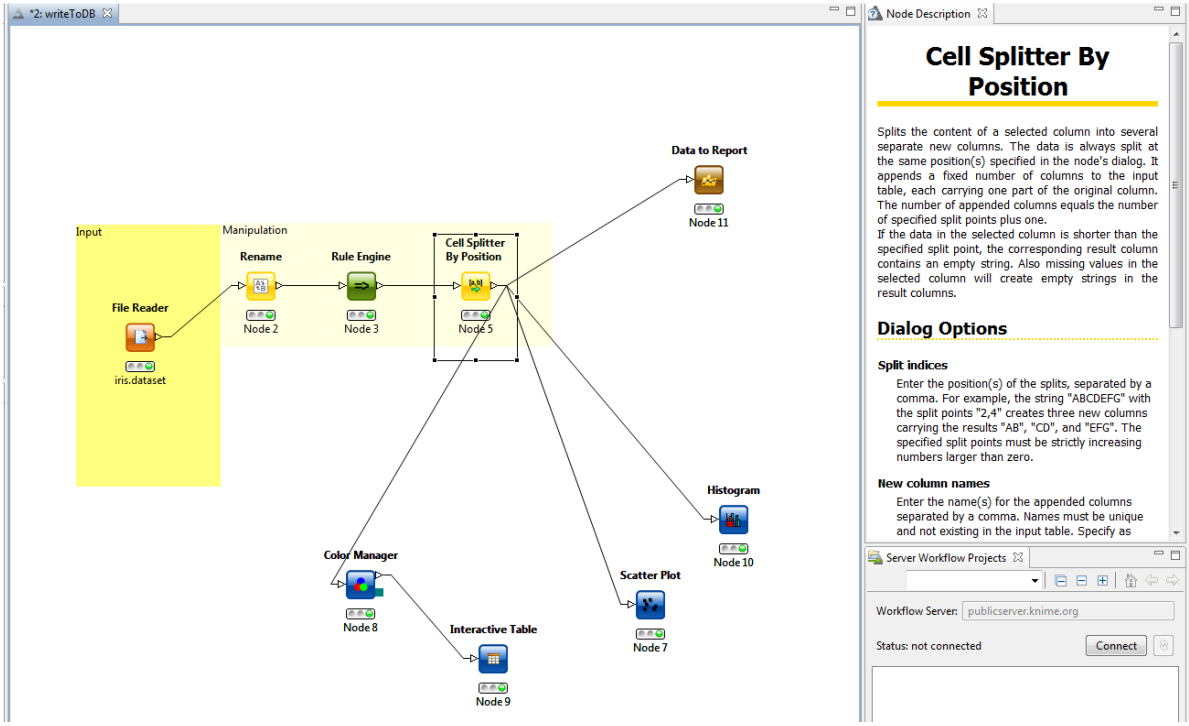


Figure 27 - Workflow Editor and Node Description

Selecting a node also triggers its node description to pop up in the sidebar, as shown in the image above.

The node description allows the developer to add a description to the node, detailing its purpose and preferred use.

Furthermore, there are certain returning segments in all descriptions. The descriptions can include Dialog Options, which give more detail to all options available in the configure menu of the node.

The description also included information about the ports of the node, describing what input and output is expected from the node.

Finally, the description might optionally include information about what views can be called from the node. Views are often used in the standard plots of KNIME itself, however we have chosen to call actual `JFrames`, allowing for more interactivity and persistency.

4.3.4 Data tables

Normally, information is passed between nodes using data tables. These are tables containing numbers or text.

The possible data types are:

- `Integer`: Whole numbers in the range $[-2^{31}; 2^{31}-1]$
- `Double`: Double precision numbers in the range $[2^{-1074}; (2-2^{-52})*2^{1023}]$
- `String`: Text

This information is passed via the ports. To create a data table, you must first create its specifications. This means the number of columns, and their types. After that, you fill the table, and pass it to the next node. Every row in the table also has a row key. This is a `String` value, and it is used to identify the rows (like the primary key in a database). This is done in the execute() method of the node.

The following example creates a simple data table with 50 rows. Each row has an `Integer`, `Double` and `String` column.

```
protected BufferedDataTable[] execute(final BufferedDataTable[] inData,
            final ExecutionContext exec) throws Exception {
      // Create the specifications
      DataColumnSpec[] allColSpecs = new DataColumnSpec[3];
      allColSpecs[0] = new DataColumnSpecCreator("String column",
                  StringCell.TYPE).createSpec();
      allColSpecs[1] = new DataColumnSpecCreator("Double column",
                  DoubleCell.TYPE).createSpec();
      allColSpecs[2] = new DataColumnSpecCreator("Integer column",
                  IntCell.TYPE).createSpec();
      DataTableSpec outputSpec = new DataTableSpec(allColSpecs);

      // Creates a container in which to buffer the rows during
creation
      BufferedDataContainer container =
exec.createDataContainer(outputSpec);
```

```
        // Add the 50 rows
        for (int i = 0; i < 50; i++) {
              // The row key
              RowKey key = new RowKey("Row " + i);
              // Create the cells
              DataCell[] cells = new DataCell[3];
              cells[0] = new StringCell("This is row " + i);
              cells[1] = new DoubleCell(0.5 * i);
              cells[2] = new IntCell(i);
              DataRow row = new DefaultRow(key, cells);
              container.addRowToTable(row);
        }
        // Close the container
        container.close();
        // Get the data table
        BufferedDataTable table = container.getTable();
        // Return the data table
        return new BufferedDataTable[] { table };
}
```

Code snippet 5 - Create a new data table

4.3.5 (Work)Flow variables

Another way to pass information between nodes is by using flow variables and workflow variables. These are special variables which can pass information from one node to another without using a data table. In these variables, data is stored as an `Integer`, `Double` or `String` value.

The difference between flow variables and workflow variables is their scope. Flow variables are only available "downstream" in the flow, and only in nodes connected with ports. Workflow variables on the other hand are available in all of the workflow, regardless of the nodes location in the flow or whether it is connected to the node which created the variable. It is also more difficult to create workflow variables, because normally they should not be created from within the code. They are normally used by KNIME internally.

The following code snippet illustrates the difference between creating a flow variable and creating a workflow variable:

```
/**
 * Create a normal flow variable
 *
 * @param name
 *            The name for the new flow variable
 * @param value
 *            The value for the new flow variable
 */
public void createFlowVariable(String name, String value) {
      pushFlowVariableString(name, value);
}
```

Code snippet 6 - Creating a normal flow variable

```
/**
 * Create a workflow variable
 *
 * @param name
 *            The name for the new workflow variable
 * @param value
 *            The value for the new workflow variable
 */
public void createWorkflowVariable(String name, String value) {
      WorkflowEditor editor = null;

      for (IWorkbenchWindow iww : PlatformUI.getWorkbench()
                  .getWorkbenchWindows())
            editor = (WorkflowEditor)
iww.getActivePage().getActiveEditor();
      WorkflowManager manager = editor.getWorkflowManager();

      manager.addWorkflowVariables(true, new FlowVariable(name, value));
}
```

Code snippet 7 - Creating a workflow variable

4.4 FACEJava

While on an abstract level it is possible to talk about Flow Cytometry-related entities such as structured *FCS data* and *gates*, these entities do not exist within a programming context.

While it would be possible to create these structures ourselves, using a common library eases the implementation of these entities.

FACEJava is such a Java library. It contains a set of classes which handle all Flow Cytometry-related operations. It includes a library which is able to read, parse and write FCS files, a utility for reading and parsing Gating-ML files and classes to calculate statistics based on a set of gates and an FCS dataset.

The basic structure is as follows:

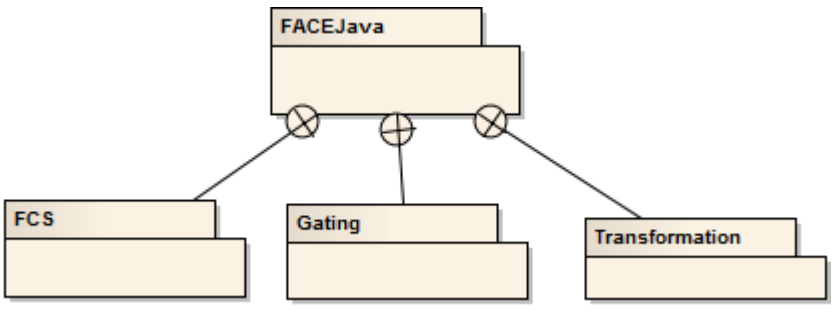


Figure 28 - FACEJava basic structure

The FCS part includes the so-called *CFCS* library. This has been the most commonly used Java library for FCS file in- and output. The library is able to receive a `FileStream` as input and is able to convert this into an easily useable `FCSDataSet` object.

It also includes classes to calculate statistics based on gates and an `FCSDataSet` object.

These calculations can include, for example, the average value of data on the FSC parameter inside a certain gate.

The gating part is used to read and parse Gating-ML files, and check if certain FCS data values are in certain gates.

To read the Gating-ML, *JAXB* is used. This is the **J**ava **A**rchitecture for **X**ML **B**inding. It includes the class `GatingMLFileReader`, to which a Gating-ML file is passed, and a FACEJava `GateSet` object is returned.

All of the gates also have a method `isInside`, to which one or more `Event` objects can be passed. Using this method, we can easily check if certain data (converted into an `Event` object) falls within the gate.

The transformation part is to manipulate the data to make it easier to distinguish interesting clusters of data. Transformations with FACEJava fall outside the scope of this thesis.

If transformations are used, such as the GLOG transformation within the Compensation Workflow mentioned later on, these are new transformations not yet implemented in the FACEJava package.

5 WORKFLOWS

The development process is focused mainly on the creation of individual nodes, those nodes only become truly useful when they are connected with each other, forming a genuine workflow.

The workflows used during development were often self-made. Once all required nodes were created, a PHAEDRA programmer integrated this with the overarching PHAEDRA system and reorganized the workflow as required by the scientists.

As such, this chapter will be focused on what was actually developed during my internship. The chapter will start each time with an overview and goal of a specific workflow and will then zoom in on the individual nodes created by myself.

Essentially two levels of functionality will be referred to: the functionality within the workflow and the internal workings of a node. I have tried to maintain a well-balanced combination of both.

In this chapter I describe the following items:

- Cell Filtering Workflow
  - I/O Nodes
  - Control Wells
  - 1 Dimensional Range Gating
  - Population Hierarchy
  - Writing features to the PHAEDRA database

- Compensation Workflow
  - Calculating Medians
  - Creating the spill over matrix
  - Applying the GLOG
  - Visualisation

- Purified Cells Workflow
  - Entire workflow is explained
  - Changes to I/O Nodes

5.1 Cell Filtering Workflow

The figure below depicts the first workflow which I worked on during my internship. At a later stage, it was given the name GFP Workflow, in reference to its main use: analysis of the **G**reen **F**luorescence **P**opulation. This workflow is pictured in the image below. An enlarged version can be found in the appendices.

It contains many fundamental aspects of Flow Cytometry, making it an excellent starting point.

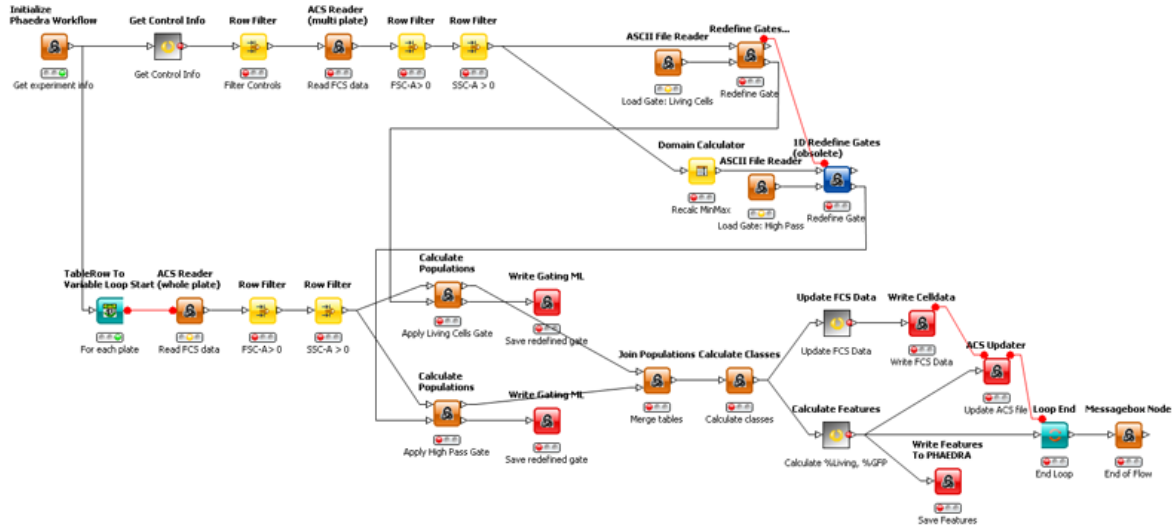


Figure 29 - Cell Filtering Workflow

The overall purpose of this workflow is the filtering of cell data through *gating*. Gating is the process of creating a *gate* on a large group of data. A gate is a selection of data that might be interesting. A gate can have many types, such as a range, ellipsis or polygon gate. The different types of gates define how the selection process occurs.

This selection process can happen in various dimensions, depending on what method is most optimal to visually distinguish the correct values. For instance, separating dead cells from living cells will often happen on a 2-dimensional scatter plot, using both the front and side scatter values. Both values contain relevant information, and plotting them together creates visually discernable clusters as seen in the example image below.

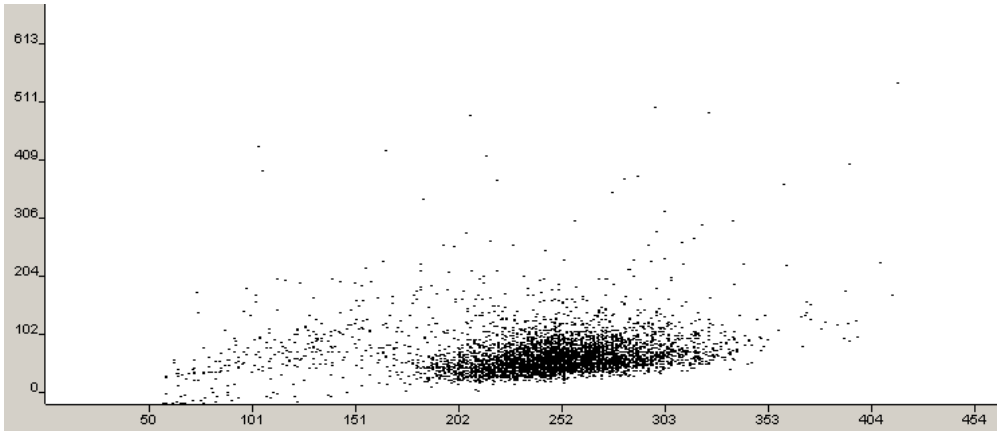


Figure 30 - Example of a cluster in a scatter plot, using the Front (FSC-H) and Side (SSC-H) Scatter as dimensions

However, when performing a selection based on the density of a specific type of fluorescence, it is more interesting to only have one dimension: the fluorescence itself. In particular, it is indispensible for the graph to give you as much information concerning density as possible. A visual example of this can be seen in the image below.

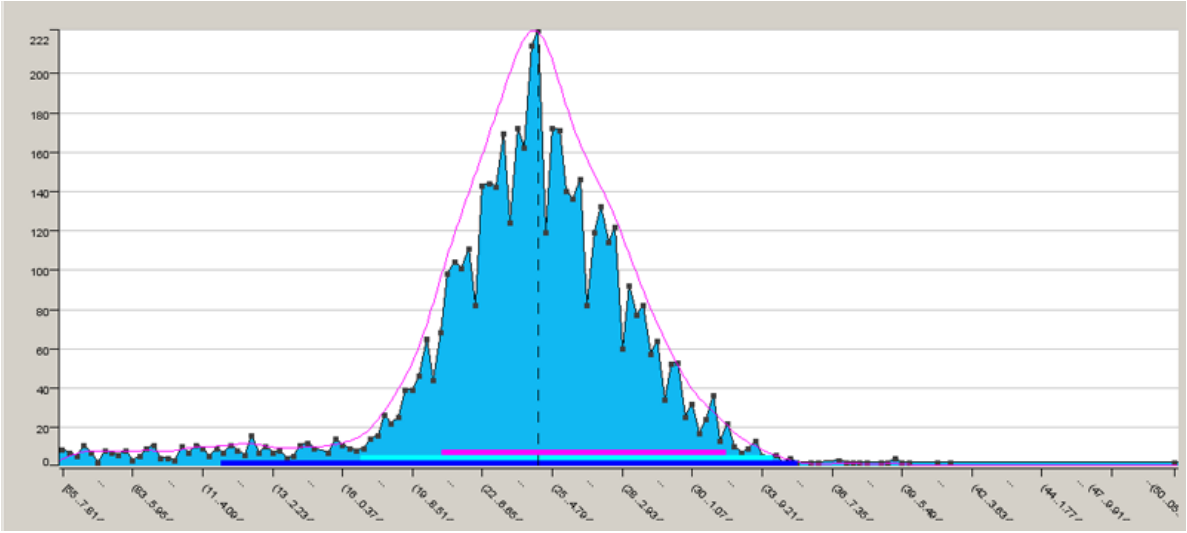


Figure 31 - Example of a 1d density plot, showing its various overlays of information including percentiles, the median and kernel density estimation

As you can see, gating is a fairly broad term. In short, gating allows for the selection of interesting data. Specific details can be found in the sub-chapter 5.1.3 '1 Dimensional Range Gating'.

Once a gate is selected, specific information about these data values can be calculated or gathered. This calculated data, known as features, are then written to the database. The calculation of classes and features is outside the scope of this thesis. Writing the features to the database is discussed in the sub-chapter '5.1.5 Writing features to the PHAEDRA database'.

The original collection file used as input is eventually also updated with information of the defined gates. Because input and output are discussed in the same chapter, you will find more information on the output in the sub-chapter '5.1.1 I/O Nodes'.

5.1.1 I/O Nodes

Before gating can be done on data, the workflow actually requires the data to be available in a useable format. Input data can exist in two formats: *ACS* or *FCS*.

These formats were already introduced in the background chapter.

FCS files are the files that are generated by the Flow Cytometry device. These files are unreadable by normal text editors, and require to be parsed first. Luckily, the PHAEDRA system already contained a parser for this specific file type. The file data is parsed into various data tables, discussed earlier. These data tables are the backbone of the KNIME nodes and are perfectly suited for FCS data.

ACS files are essentially .ZIP files, compressed collections of data. The data usually consists out of various FCS files, together with an XML file which contains the table of contents for the package.

In order to properly parse the data within these collections, the package has to be decompressed first. I found a lot of help regarding this in a technical article called 'Compressing and Decompressing Data Using Java APIs' on the Oracle website.

The article points out the usefulness of the Java.util.zip package for dealing with these kinds of data. The workings are quite similar to that of a `FileStream`, with the main difference being that you use `ZipEntry` objects.

The following piece of code, an example from the above mentioned article, demonstrates the use of these objects.

```
import Java.io.*;
import Java.util.zip.*;

public class UnZip {
   final int BUFFER = 2048;
   public static void main (String argv[]) {
      try {
         BufferedOutputStream dest = null;
         FileInputStream fis = new
         FileInputStream(argv[0]);
         ZipInputStream zis = new
         ZipInputStream(new BufferedInputStream(fis));
         ZipEntry entry;
         while((entry = zis.getNextEntry()) != null) {
            System.out.println("Extracting: " +entry);
            int count;
            byte data[] = new byte[BUFFER];
            // write the files to the disk
            FileOutputStream fos = new
            FileOutputStream(entry.getName());
            dest = new
              BufferedOutputStream(fos, BUFFER);
            while ((count = zis.read(data, 0, BUFFER)) != -1) {
               dest.write(data, 0, count);
            }
            dest.flush();
            dest.close();
         }
         zis.close();
      } catch(Exception e) {
         e.printStackTrace();
      }
   }
}
```

Code snippet 8 - Decompressing a ZIP file. Exactly the same process can be used to decompress an ACS file.

One important point to make is that when updating ACS files (or ZIP files, for that matter), it is not possible to actually append content directly into the file. Instead, the file has to be deconstructed in the above mentioned way and then reconstructed with the extra file.

While this is not an optimal situation, the increased complexity in the code did not pose many problems. However, this work-around could be circumvented by using a custom Java library. I would recommend TrueZip, as this provides the same functionality as a normal file system, but within a compressed file. TrueZip provides various methods which can handle this complex functionality directly.

XML files were easier to parse. I chose the simpler Document Object Model (DOM) method of extracting data from the XML file. While this method is notorious for being slower, I decided to use this method for two reasons:

- The XML files always have a relatively small file size
- The XML files have to read in their entirety

It should be pointed out that there are alternative methods available which could provide a performance boost and use less memory, such as SAX. The SAX method is

however more difficult to use while, in this specific case, providing only limited advantages. Therefore it was not worth the effort to use this instead.

The parser is used to retrieve information from the ACS Table of Content file.

Regular text files are read in by the ASCII Reader, which simply translates the file contents to a single cell data table. A file only has to be read once, and the workflow can from that moment on use the single cell data table. This will keep performance high by reducing unnecessary I/O actions. An example of such a single cell data table can be found below.

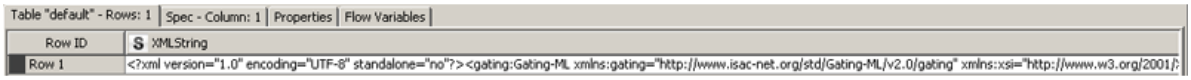


Figure 32 - Example of a single cell data table

5.1.2 Control Wells

Workflows will essentially perform gating on a large amount of cell data, iterating through all the files of the selected plate.

Applying a gate will require user input, but obviously it is not very user friendly to ask for such input each iteration.

Instead, the gating will only be performed on a selected amount of well data, called the 'control wells'. These wells are identified by their well type being either 'High Control' or 'Low Control'. Once applied to these wells, the gates are saved and sent through to the actual loop.

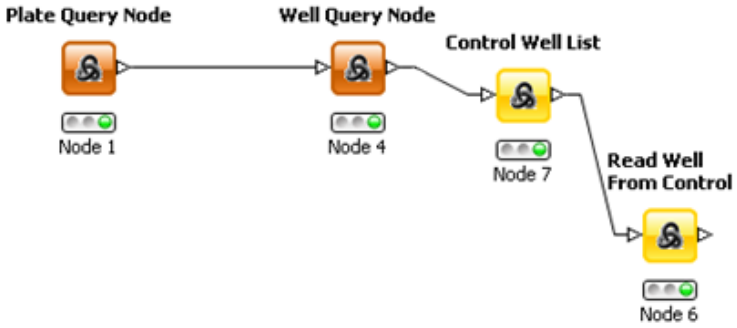


Figure 33 - Control Well nodes (yellow) connected to database nodes (orange)

With this in mind, I created the control well node. This node can filter out a specific set of data which represents the entire data pool. We call these specific files the *control wells*. As seen in the image above, we start out with several nodes (‘Plate Query’ and ‘Well Query’) which were created by other students. The combination of the first two nodes will produce a list of all the well data within a plate.

Such a list can then be filtered by the Control Well List node. It will iterate over the data table and check the type of all wells. When the type fits with what was specified during the configuration of the node, this well will be collected. When the node finishes its execution, it generates a table with all collected wells.

One of our general tasks was to keep the nodes as generic as possible, allowing re-use.

As such, this node was expanded to a node that was able to filter not only control wells, but every well type possible. This required a database connection during the configuration of the node, which would retrieve all available well types and set them as options.

You will notice such improvements in other nodes as well, always with re-use of nodes as a goal.

5.1.3 1 Dimensional Range Gating

As mentioned before, the crucial part in the cell filtering workflow is the gating. My specific task was to apply gating on a one-dimensional representation of the data.

The data that is interesting to the user in a one-dimensional environment is the intensity histogram of a certain fluorescence channel. In other words, how many times do similar values appear within the input data.

Visually, I represent this with a graph. The graph is capable of visualising the data through either a series of histogram bars or a line plot. The focus of the node is primarily on the line plot.

Because of the strong focus on density, I was advised to add an additional element - a *jitter plot*, as demonstrated in the image below.

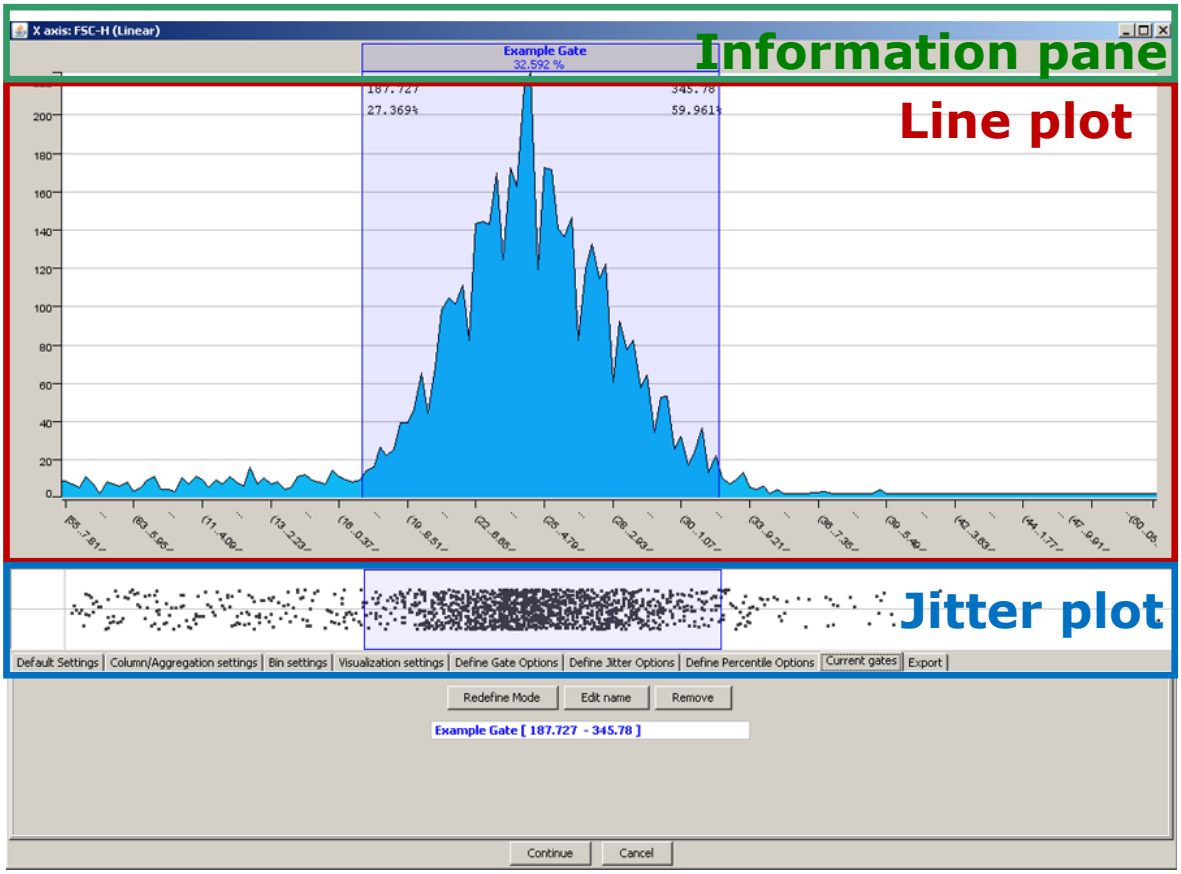


Figure 34 - Distinction between line plot, jitter plot and information pane

Jitter Plot

A jitter plot is a plot which represents all values as dots on a single line. When a value appears more than once, and would thus be placed on the same position, a random y-value is added to the position of the dot. This random value is called the jitter, to which the plot earns its name.

This has the effect seen above. In low density areas there will be little to no dots, while in high density areas there will be clusters of dots visible.

Visualisation

Now, when dealing with density visualisation of data, the following aspects have to be considered.

With the high precision of the input data, it is unlikely there will be many identical values. In order to view densities within this type of data, we apply a principle called *binning*. When binning, the data is categorized in groups of values with similar content. Each group is called a bin. Every bin has the same range, but the range itself depends on how many bins are requested.

The base amount of bins chosen is automatically calculated. This base amount can be edited by a certain margin set by the programmer. For instance, if the base amount of bins chosen is 200, the user could change this to anything from 150 to 250.

The calculation is done by applying the 'Scott's Choice' algorithm, described by David W. Scott in his paper 'On optimal and data-based histograms'. The formula, where σ represents the standard deviation of the data, is:

$$h = \frac{3.5\,\sigma}{n^{1/3}}$$

In order to create the line plot, each mid-point of such a bin is connected with each other. Since the scientists preferred the area below the line to be coloured, I used a `Polygon` for this visualization.

As visualized in the image below, the polygon shape can be used to create a rectangle with one complex side that follows the pattern of the line plot. Using the polygon allowed for an easy way to fill up the complex area of space beneath the line plot.

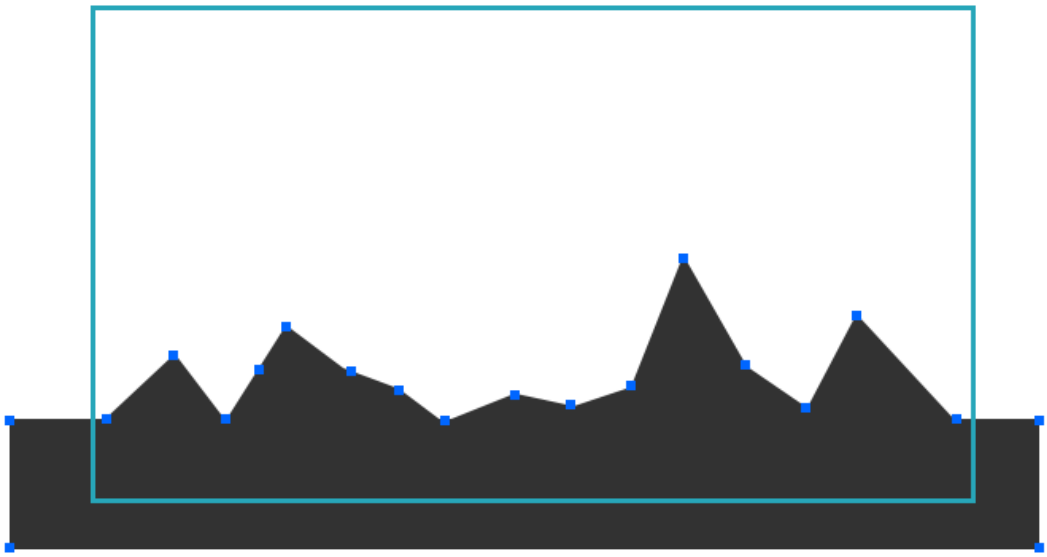


Figure 35 - Example of how a polygon form is used to fill a complex area

Applying gates

When adding or redefining gates to the plot, the raw values and corresponding percentages are displayed at the start and end of a gate. Above the plot itself an information pane was added. This serves as a clean method to show additional information such as the name of a gate and the percentage of the area within the gate.

Furthermore, axis information is added. It shows which channel is being shown and whether or not the data is represented linear or logarithmically. The information pane is illustrated in the example image below.

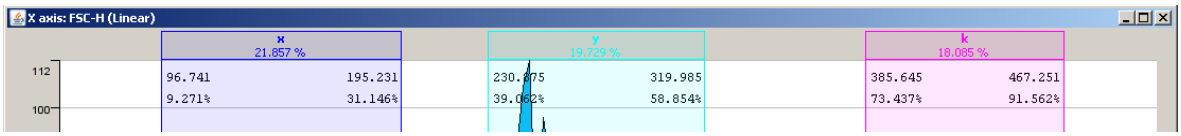


Figure 36 - An example of the information pane, containing gate and axis info

Informative features

Being an important node in the workflow, extra attention was spent on adding a large amount of features. All features are implemented with the purpose of providing as much information concerning the density, or the overall data, as possible.

Logarithmic data representation

The plot has the ability to process both linear and logarithmic data. By configuring the node, a checkbox defines which option should be used. When using logarithmic data, the node will perform conversions where needed between the raw data and it is logarithmic equivalent.

Logarithmic representations of data are used to provide a clearer image of the density distribution as the function creates data that is more uniformly spread among the plot.

Below you can see a visual example of the difference. In the left image, most data is stacked into the first few bins, while a small amount of extremely large values (visible in the jitter plot) cause the plot's range to be off.

In the right example you can see the same data, but this time using the logarithmic conversion built into the node. The data is spread more uniformly, resulting in a much nicer visual plot. Gates set will still use the original data as reference, so that the functionality remains as if the data was never converted.

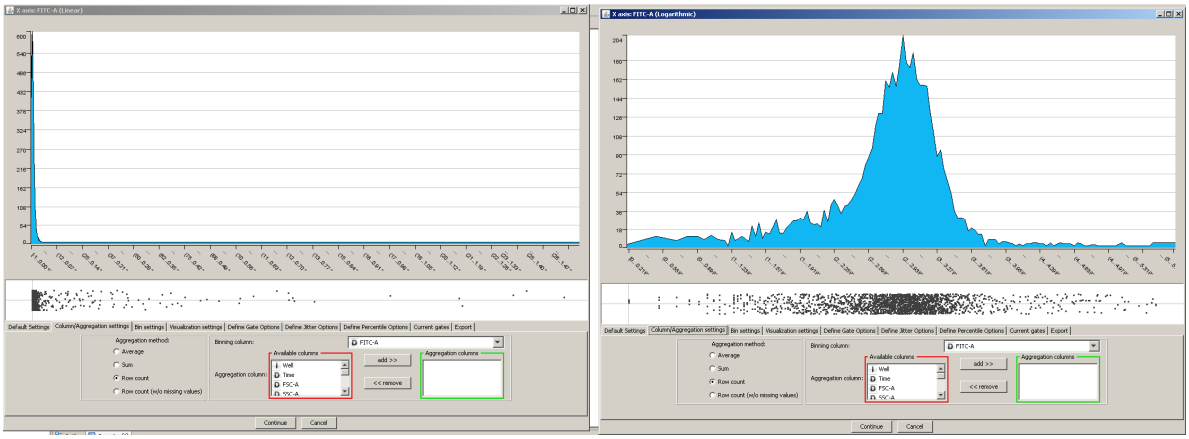


Figure 37 – A comparison of linear and logarithmic representations of the same data

Spline smoothing

To make the line plot more visually pleasing, spline smoothing was added. To implement this, I used code provided by the Australian University of New South Wales. All that was required was that I create a ControlCurve object and fill it with point data.

I chose to apply a Cubic Spline as visual smoothing. An improvement on this type of smoothing is Savitzky–Golay smoothing, which was added in a late stage of development. This is a smoothing technique suggested in 1964 by Abraham Savitzky and Marcel J. E. Golay. This technique is however outside the scope of this thesis.

Kernel Density Estimation

In order to provide more insight into the distribution of densities, *Kernel Density Estimation* (KDE) was added as well. As the name suggest, this is an algorithm that can estimate what the density will be on a certain point.

It uses the raw values as input data to fuel its calculations. I then loop over the entire range of the density plot, requesting an estimation for every point. The result is a smoother version of the line plot, clearly showing where densities are present, as is visible in the example below.

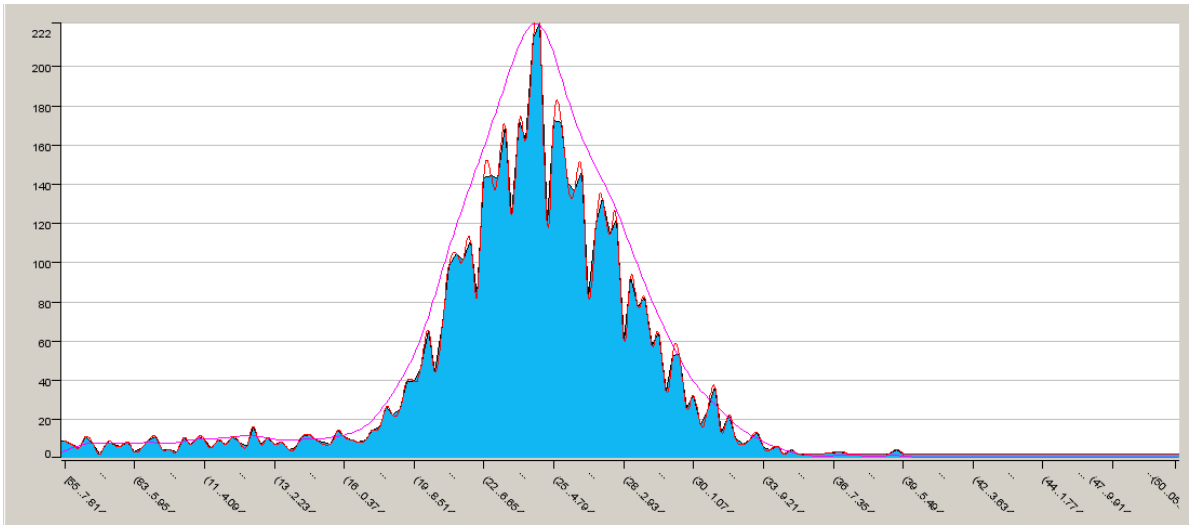


Figure 38 - Kernel Density Estimation (purple) on top of a line plot with splining (red) enabled

In practice, I used classes provided by the University of Waikato (WEKA). This university has provided the KNIME community with various KNIME nodes already concerning data mining and machine learning.

Hiliting

Another useful addition, originating from KNIME itself, is *hiliting* (deliberately spelled this way by KNIME). Hiliting allows for consistent highlighting of data over the entire workflow. If a specific range of data is highlighted in such a way in one plot in the workflow, it will become highlighted in all visualizations of the workflow.

Because this would be hard to implement into a line plot, this is only visualized in the jitter plot.

In the example below, you can see a standard, unedited node from KNIME itself. In it, certain histogram bars have been selected and 'hilited'.

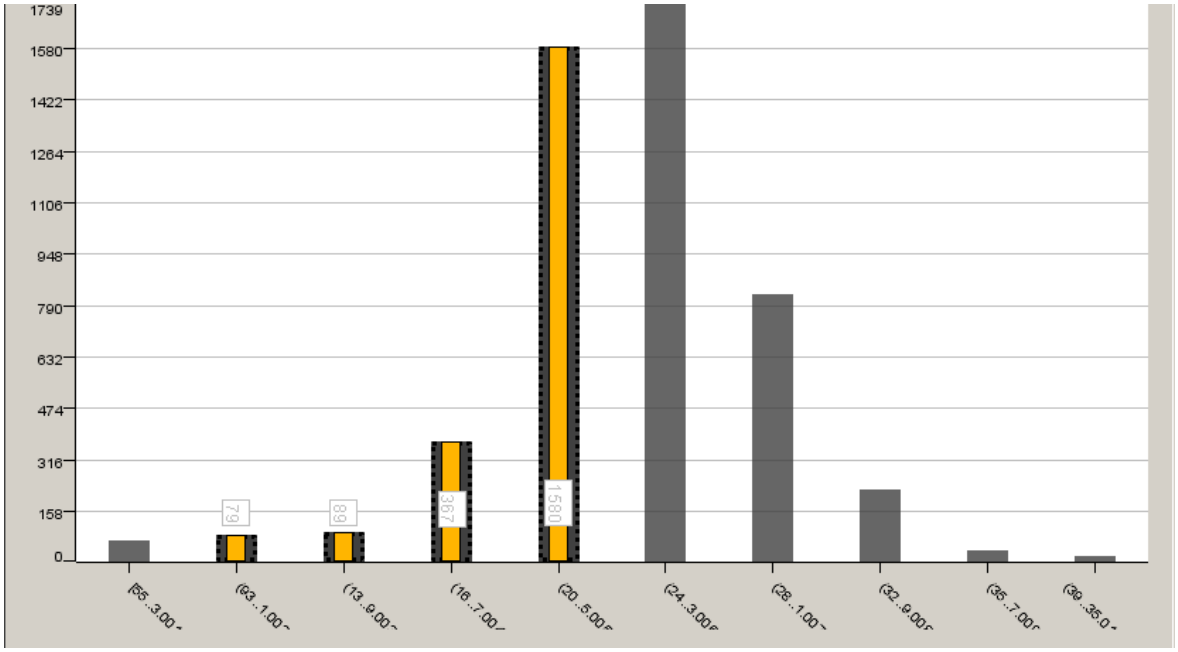


Figure 39 - Hiliting several bars in a common histogram

In the following example, within the same workflow, you can see my node. The jitter plot recognizes certain data as 'hilited' and marks that data in the global hilite colour.

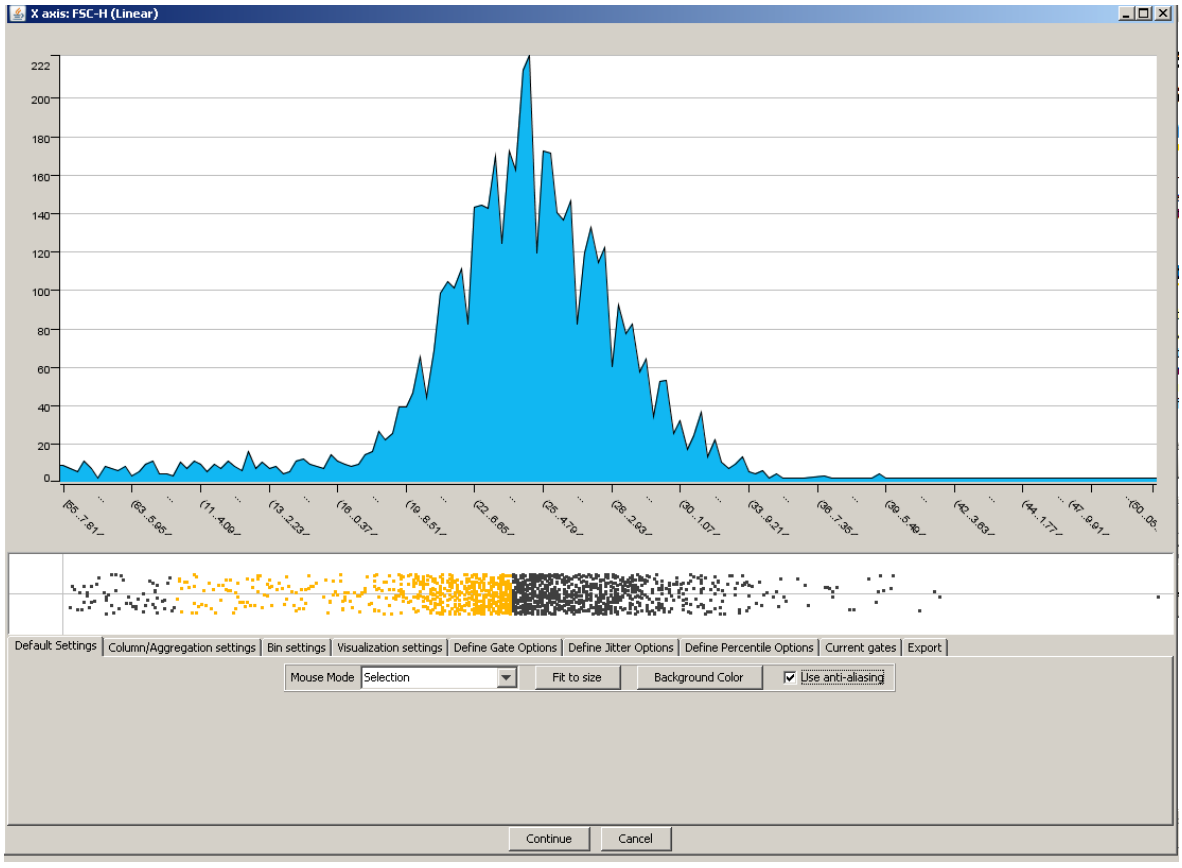


Figure 40 - The same data immediately becomes highlighted on the jitter plot

Percentiles

Finally, I added a series of percentile oriented options. A percentile is a range of data focused around the modus. For example, the 70th percentile defines a range of data in which 70% of all data is located.

The original version added provides this date by automatically calculating it and centring it on the mean, the largest value within the range.

Data, however, will not always simply contain a single peak. More complex data might consist out of multiple peaks, which would render the percentile information too restrictive. As such, I added an option which visually showed the mean value as a dotted line. An example can be seen below.

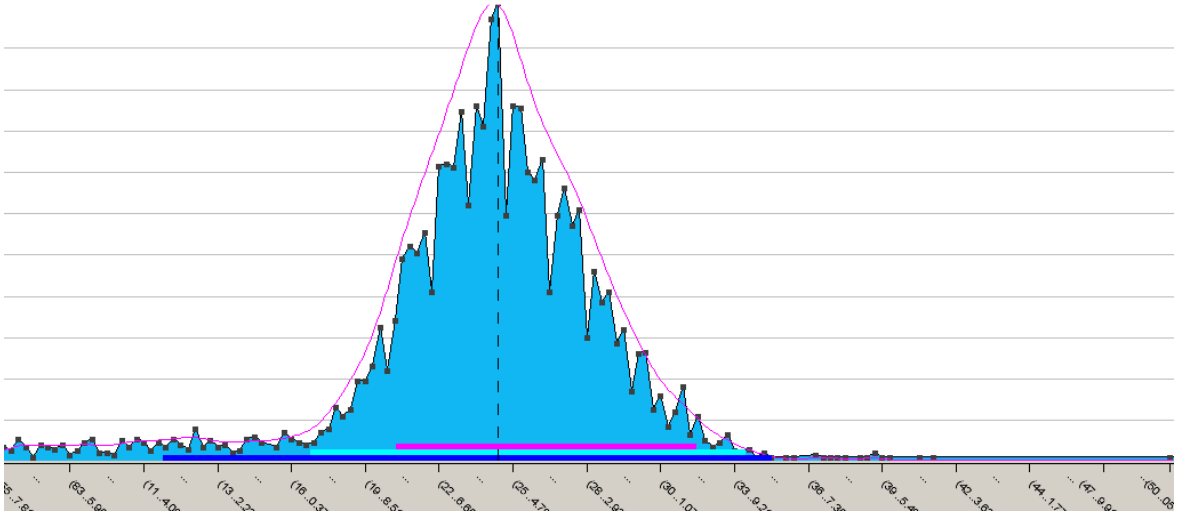


Figure 41 - A closer look at the median being represented by a dotted line

This dotted line can be placed wherever the user wishes, and the percentile bars will be recalculated. This allows users to receive information about any peak they wish.

Additionally, a cumulative plot was added. This was done for three reasons:

- Additional information to interpret the percentiles
- Easier to properly position the dotted line
- Generally to provide more information to the user.

As the name suggests, the plot shows the accumulated density of the value and all values up to that point. An example of a cumulative line added to the graph can be seen below.

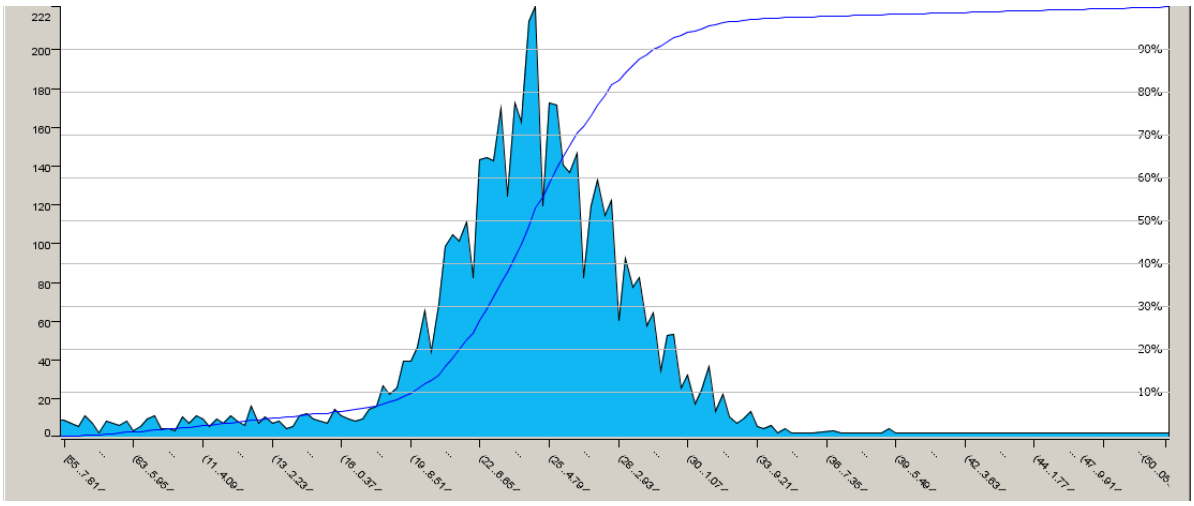


Figure 42 – An example of a cumulative plot

5.1.4 Population Hierarchy

It is indispensable to retrieve statistical information concerning the populations within a dataset. These populations are essentially determined by (combinations of) the gates we have defined.

A population can be defined as either a:

- Positive Gate population: When we refer to the data that falls within the boundaries of a gate
- Negative (Remainder): When we refer to the data that falls outside of the boundaries of a gate

This node will show an informational frame listing the populations in a hierarchical manner, supplemented with class and class colour information for the leafs. These are retrieved from the database.

Furthermore, the node adds two types of percentages:

The parent percentage defines how much of this population falls within the boundaries of its parent population.

For example, in the image below, the parent percentage of the GFP population is the percentage of GFP data that falls within the Living gate. Only the Living Cells class is defined in the database, therefore only this leaf is able to receive extra information about its class.

Population Hierarchy - 2:70 - Population Hierarchy

File

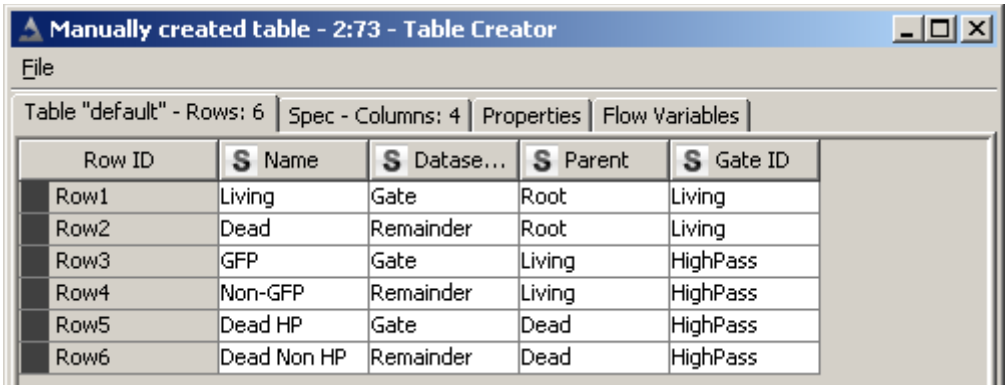
Name	Class	Bitpattern	Color	Description	% Parent	% Total	Count
Root					100	100	5000
Dead Cells					56.68	56.68	2834
Dead Non HP					58.958	33.418	1670
Dead HP					41.043	17.78	889
Living Cells		11.....		Living Cells	43.32	43.32	2166
Non-GFP					29.872	12.94	647
GFP					70.129	30.38	1519

Figure 43 - Example of a population hierarchy node. Only relevant leafs receive class info

The total percentage defines how much of the total data belongs within the population boundaries.

However, this node had a specific difficulty. Both the dataset and the gating information did not contain any kind of hierarchical structure. It was with this data impossible to tell which gate should be seen as a parent and which as a child.

Since the hierarchy was something specific to a workflow, it was entirely possible to use a separate input table which contained such data, as pictured below.



Row ID	Name	Datase...	Parent	Gate ID
Row1	Living	Gate	Root	Living
Row2	Dead	Remainder	Root	Living
Row3	GFP	Gate	Living	HighPass
Row4	Non-GFP	Remainder	Living	HighPass
Row5	Dead HP	Gate	Dead	HighPass
Row6	Dead Non HP	Remainder	Dead	HighPass

Figure 44 - Example of a table used to represent a hierarchical structure

Because this hierarchy is specific to a workflow, the developer setting up the workflow already has this information and can easily create this table itself. An example of such a table that defines that hierarchical structure can be seen in the example above.

The *Name* column in the table is purely aesthetic and defines which label text is set for the leaf.

The *Dataset Type* column defines whether this leaf is actually linked to a gate or if it defines a remainder. Both are linked to an actual gate through the *Gate ID* column.

If a *Dataset Type* is set to Gate, it will use the data that falls within the boundaries of the linked gate for its calculations.

If the *Dataset Type* is set to Remains, it will use the data that falls outside of the boundaries of the linked gate for its calculations. All data within the gate is thus ignored.

The *Parent* column of the table is set to either a *Name* set in the *Name* column or it is set to *Root* when the leaf has no actual parent. This column is the core of the hierarchical structure.

In order to observe the strength of the Population Hierarchy node, look at the above example in figure 43 and 44. In the above example, two actual gates exist: Living and HighPass.

With just these two gates, the Population Hierarchy node is able to provide information about 7 possible classes within the data. One of the classes, *Living Cells*, has already been defined in the database. Therefore, the node is able to retrieve information about this class and immediately show it (without the user actually having to define it).

This node was originally created with a `JFrame`. However, since it lacks actual interactivity and is purely informational, I changed it to the KNIME-specific `View` class.

This allows the window to be re-opened as much as the user wishes and sets the window to be on top by default.

5.1.5 Writing features to the PHAEDRA database

One of the aims within a workflow is the calculation of *features* on the data, which can be used for further analysis.

Features can be seen as a number of statistics on the data that can be interesting. For instance: the percentage of all living cells within a dataset. The calculation of these features is done either by use of existing nodes (often math and transformation nodes) or a custom tailored node which handles the calculation of averages, mean intensities and other general statistics.

The calculation of features themselves are outside the scope of this thesis.

Once calculated, these features are gathered in a data table similar to the one pictured below. Note that test data was used during the analysis in the example and as such the calculated features are not representative of actual data.

The values in a data table are subsequently extracted and converted into an *XML CLOB*, then saved to the PHAEDRA database. CLOB is an abbreviation of a **C**haracter **L**arge **Ob**ject. As the full name suggests, this is a data type which is able to contain very large chunks of data. In this case an XML structure.

The connections to the database are handled by functions within the overarching PHAEDRA system.

Eventually, the saved data looks as illustrated in the image below.

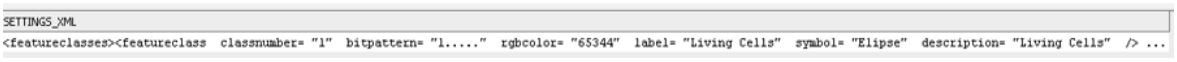


Figure 45 - Example of a database entry containing XML CLOB data

Because a CLOB is often too big to be handled as a string (varchar in the database), the data has to be retrieved with the help of a CLOB object, which contains the data in the form of a character stream.

The code snippet below shows how data is generally retrieved from this structure.

```
/**
 * Executes a select query on the database and retrieves the XML
 * @return settingsXML string
 */
private SerialClob getSettingsXML(){

SerialClob resultClob = null;
String query = "SELECT SYS.XMLType.getClobval(SETTINGS_XML) xml FROM
PHAEDRA.HCA_CELLFEATURE WHERE ...";

SqlRowSet result = Connector.select(query);

while(result.next()){
      resultClob = (SerialClob) result.getObject(1);
}

return resultClob;
}
```

```
/**
 * Parses the string xml into cellfeature (custom class) objects
 * @param settingsXML
 */

private void parseXML(SerialClob settingsXML) {

try {

DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder db = factory.newDocumentBuilder();
InputSource inStream = new InputSource();

inStream.setCharacterStream(settingsXML.getCharacterStream());

Document doc = db.parse(inStream);

doc.getDocumentElement().normalize();

NodeList nList = doc.getElementsByTagName("featureclass");

for (int temp = 0; temp < nList.getLength(); temp++) {
      Node nNode = nList.item(temp);
      if (nNode.getNodeType() == Node.ELEMENT_NODE) {

            Element eElement = (Element) nNode;

            String label = eElement.getAttribute("label");
            String classnumber = eElement.getAttribute("classnumber");
            String rgbColor = eElement.getAttribute("rgbcolor");
            String description = eElement.getAttribute("description");
            String bitpattern = eElement.getAttribute("bitpattern");

cellfeatures.add(new Cellfeature(label, classnumber, description,
bitpattern, rgbColor));

}
}

} catch (Exception e) {
      e.printStackTrace();
}
}
```

Code snippet 9 - Retrieving an XML CLOB from the PHAEDRA database

5.2 Compensation Workflow

Besides the previously mentioned Cell Filtering Workflow, I worked on two more flows.

The compensation workflow is a workflow created to take raw data and filter out the *spill over* that might have occurred. Basically, raw data can be contaminated with remnants from other sources. The contamination is called the spill over.

This workflow is pictured in the image below. An enlarged version can be found in the appendices.

A spill over matrix is calculated, which can be applied to any type of raw data. The matrix is created from analyzing two sets of data. The first set contains *unstained* data. This is data that does not contain any contamination. The second set of data contains *single stained* data. This is data that contains large amounts of contamination in one particular channel. For instance, data that is marked as 'single stained PE-A' will have a PE-A channel that is highly contaminated.

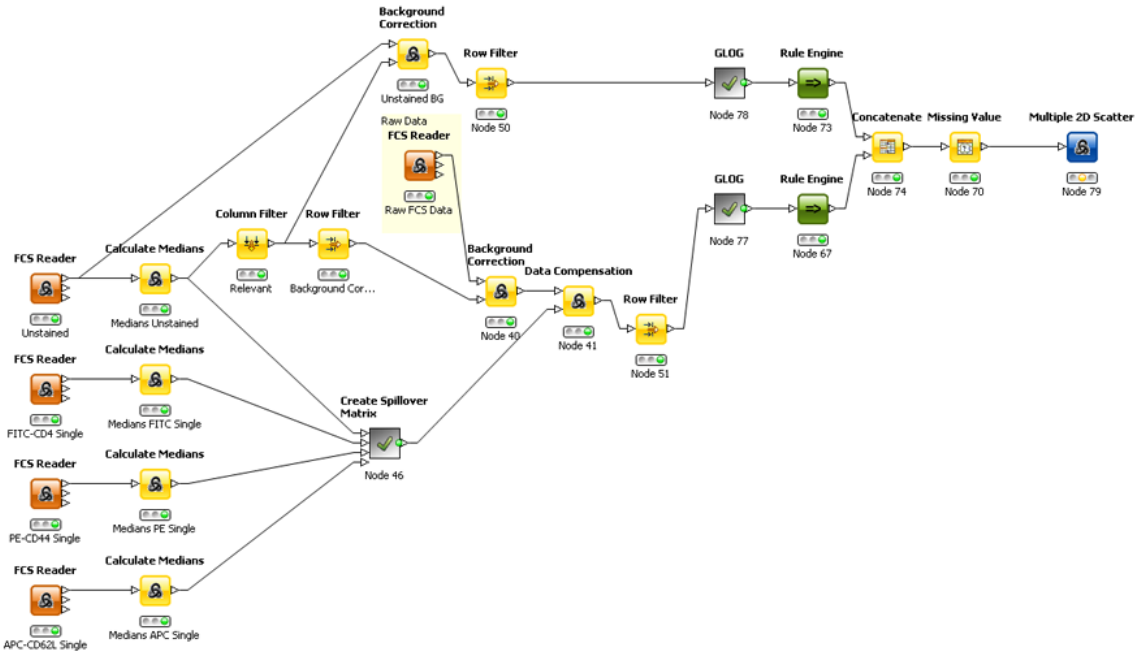


Figure 46 - Compensation workflow

5.2.1 Calculating Medians

From these datasets, the medians are calculated. Every set will create one row of median data. This is a separate node with a very basic task, which makes it very fast. All rows will then be concatenated with the existing *Concatenate* node, creating a basic matrix as seen below.

Row ID	D FITC-A	D PE-A	D APC-A
Row 1	41.31	35.19	33.36
Row 1_dup	[illegible]	[illegible]	[illegible]
Row 1_dup2	52.02	-81.09	230.74
Row 1_dup_d...	78.03	166.77	2.78

Unstained

Single stained

Figure 47 - A median matrix represented in a data table

The matrix contains a top row, which are all the medians from the unstained dataset. All other rows are single stained datasets.

5.2.2 Creating the spill over matrix

Starting with this matrix, calculations can be applied to eventually create a spill over matrix.

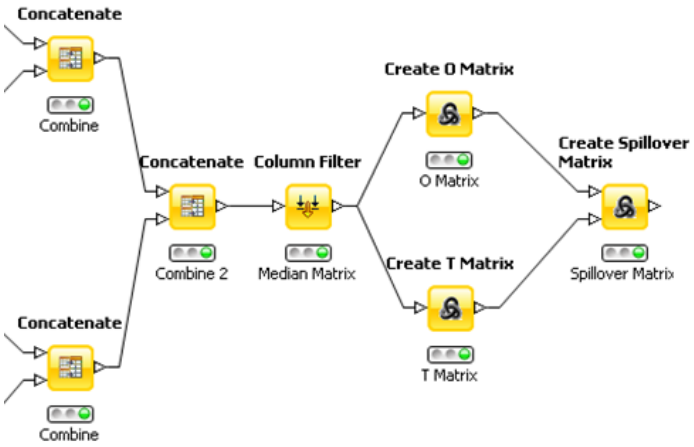


Figure 48 - Nodes required to create a spill over matrix

The 'O Matrix' is created by taking the top row from the median matrix, thus the median values of the unstained dataset. These values are then subtracted from the respective single stained values.

The 'T Matrix' is created by taking the O Matrix and filtering out only the diagonal matrix from this.

Concretely, this simplified example shows what happens during each mentioned step:

Median matrix

$$\begin{bmatrix} 1 & 2 & 3 \\ 5 & 5 & 5 \\ 4 & 4 & 4 \\ 3 & 3 & 3 \end{bmatrix} - \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

O Matrix

$$\begin{bmatrix} 4 & 3 & 2 \\ 3 & 2 & 1 \\ 2 & 1 & 0 \end{bmatrix}$$

T Matrix

$$\begin{bmatrix} 4 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Using these matrices, a spill over matrix can be created by applying the following formula, described in the article 'Scalable analysis of Flow Cytometry data using R/Bioconductor':

$$F = O^{-1} \cdot T$$

For all matrix interactions, I opted to use the Efficient Java Matrix Library (EJML) library, a library optimized for small matrices, which allows operations to be performed really quickly.

The spill over matrix created is then applied to raw data, resulting in 'compensated' data.

5.2.3 Applying the GLOG

In order to better distribute data over the plot and create a visually more intuitive graph, a series of nodes were set up to create the GLOG transformation. I based my implementation of this on the article 'Alternatives to Log-Scale Data Display' by Joseph Trotter.

I created a series of nodes (either custom made or Math nodes) which apply the following formula:

$$f_g(z) = ln(z + \sqrt{z^2 + \lambda})$$

In this formula, *z* is the variable that should be transformed. The lambda (λ) value represents a percentile of the negative values from the dataset. The 5th percentile was advised by the article.

Take note that the logarithmic function called here is the natural logarithm (base E). Depending on the platform in which you wish to call this function, pay careful attention to the implementation of this function.

For example: the natural logarithm function (base E) in Java is called by `Math.log()` and the base 10 logarithm is called by `Math.log10()`. However, in the Math Expression nodes the `log()` function calls the logarithm with base 10. Instead, the function `ln()` is used for a natural logarithm.

5.2.4 Visualisation

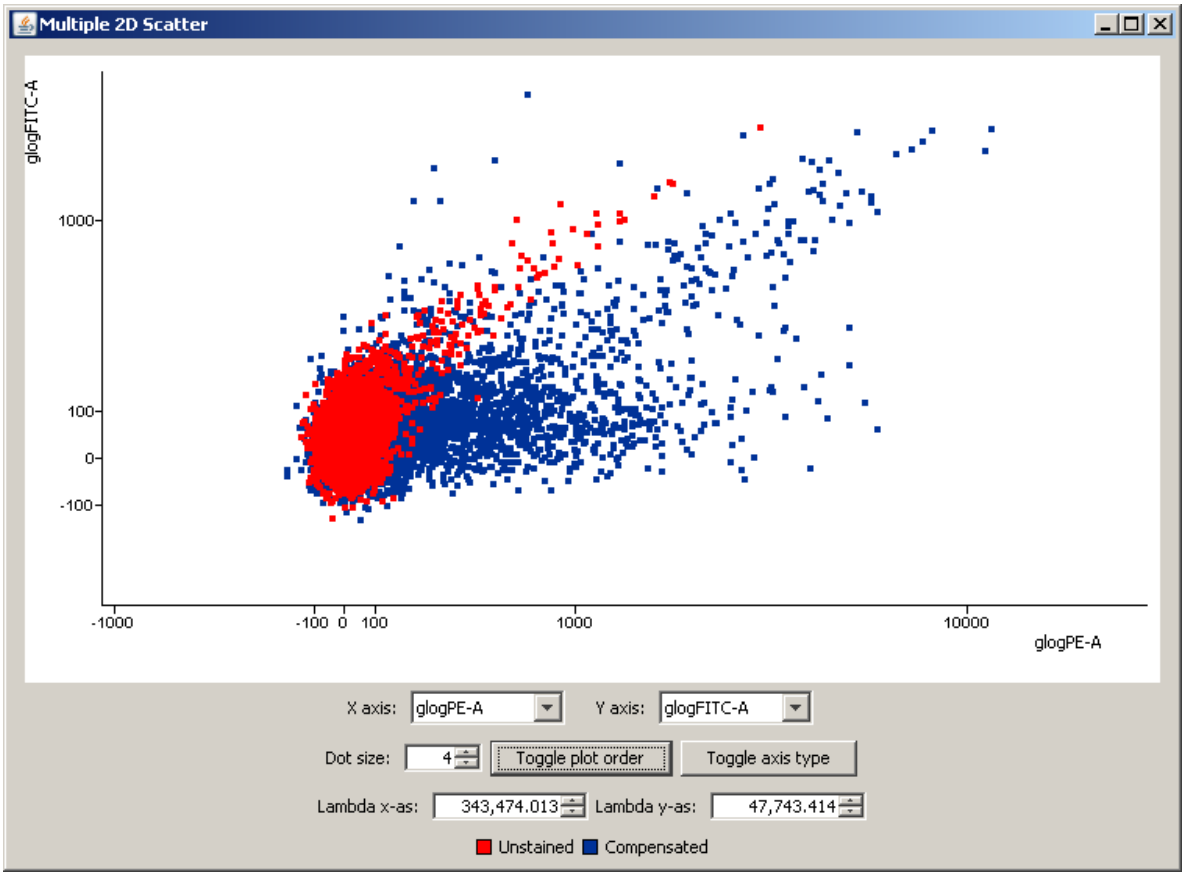


Figure 49 - Visualisation node used within the compensation workflow

As a final step, the changes are visualized in a node. While originally created with only compensation visualization in mind, this node was modified to become a generic multiple 2d scatter plot. As the name suggests, this node can receive multiple datasets as input, and is able to plot all of these on the same chart. In the compensation workflow example, the unstained data is often plotter together with the compensated data. This allows for visually more useful information.

It uses jGrial to visualize its plots, a Java library that adds a more easily manageable layer on top of the JOGL (**J**ava **O**pen**GL**) library.

It has various visual options, such as changing the dot size, the plot order (which dataset lays on top of which dataset) and a colour chooser for each dataset.

Furthermore it checks if a GLOG value is plotted, and if so, enables the option to edit the lambda on the fly. This allows for fine-tuning the GLOG as required.

5.3 Purified Workflow

The final workflow was created only a few weeks before the end of our internship. The goal of creating a flow this late was to make sure the nodes we created covered enough bases to allow for basic variations of analysis methods.

Because the majority of gating nodes used were the 1 dimensional gating nodes which I created, this workflow was assigned to me. Therefore, I can explain the entire workflow process.

The workflow is pictured below. An enlarged version can be found in the appendices.

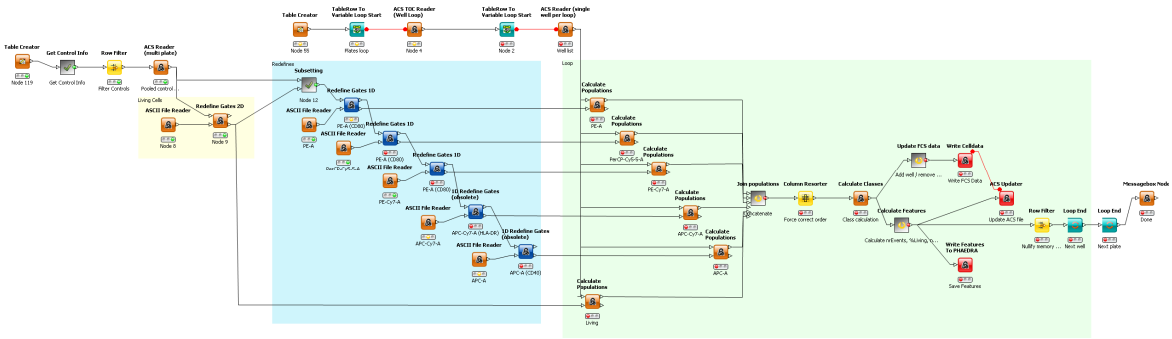


Figure 50 - Purified workflow

First of all, the node 'Initialize PHAEDRA Workflow' allows for the connection between the workflow and the overarching PHAEDRA system. With PHAEDRA, the user specifies certain wells which will act as 'control' wells. These wells are retrieved from their respective files and pooled together. Such a process is shown in the example below.

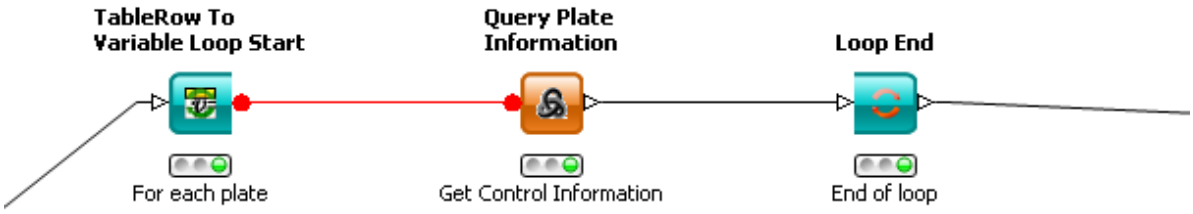


Figure 51 - Pooling of control data into a single data table

In the example above you can see two cyan nodes. These nodes cause all nodes within them to be iterated for a certain amount of times. The amount is defined by the input the *TableRow To Variable Loop Start* node receives.

In this case, this will be a list of all wells that are labelled as a control well. As a reminder, control wells are wells that represent the entire data. By using such a specific selection of data, users only have to provide user input once. As the control wells are representative for the rest of the data, the calculations and gates set on this data can simply be forwarded towards the rest of the data and applied without user input.

When the iteration in figure 51 executes, the *Query Plate Information* node will iterate over the entire list. Every time information about the respective plates will be retrieved. The *Loop End* node appends all this information to its output.

Once the above mentioned process is finished, the *Loop End* node forwards its output. This output will now consist of all plate information for all control wells. This information can be send to a I/O node that retrieves the required information.

A control is chosen so that user input is only performed on a limited amount of data, increasing performance and decreasing the need for user input. Once defined on the control data, the gate data will be applied to all real data automatically.

On this collection of controls, a 2-dimensional gate is applied on the channels FSC and SSC (front and side light scatter). This should provide the user with a data overview containing two large clusters. Only one of the clusters contains living cells. A polygon gate is applied around the living data. An example of this can be seen in the image below.

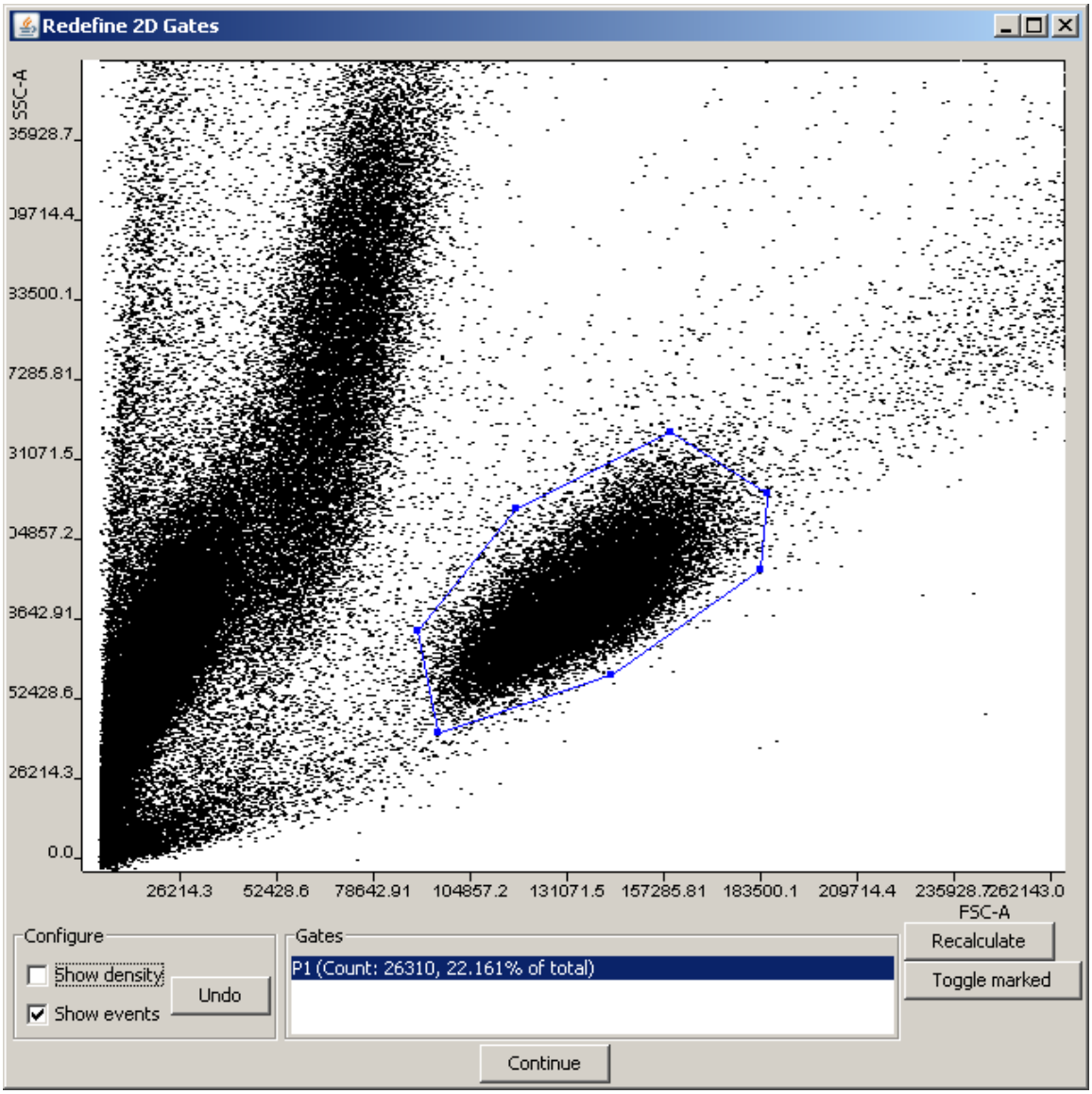


Figure 52 - An example of a gate being applied to a living cells cluster (here labelled as 'P1')

The dead cells are filtered out of the dataset in the process we call subsetting. Essentially, the gate is applied to the dataset, creating a *membership column*. This is a column that is added to each row. If the data represented by one such row is contained within the gate, the data is a 'member' of the gate. If not, the gate is not a member.

A filter is then applied to filter out only that data that is seen as a member of the living gate. The resulting, filtered, data is what we refer to as a 'subset'.

On the filtered control data, we will apply a series of one-dimensional gates on each available channel. This part of the flow is connected this way so that, when executed, each gating node will wait for its preceding one to finish. This is easier for the user, who only has to deal with one input at a time.

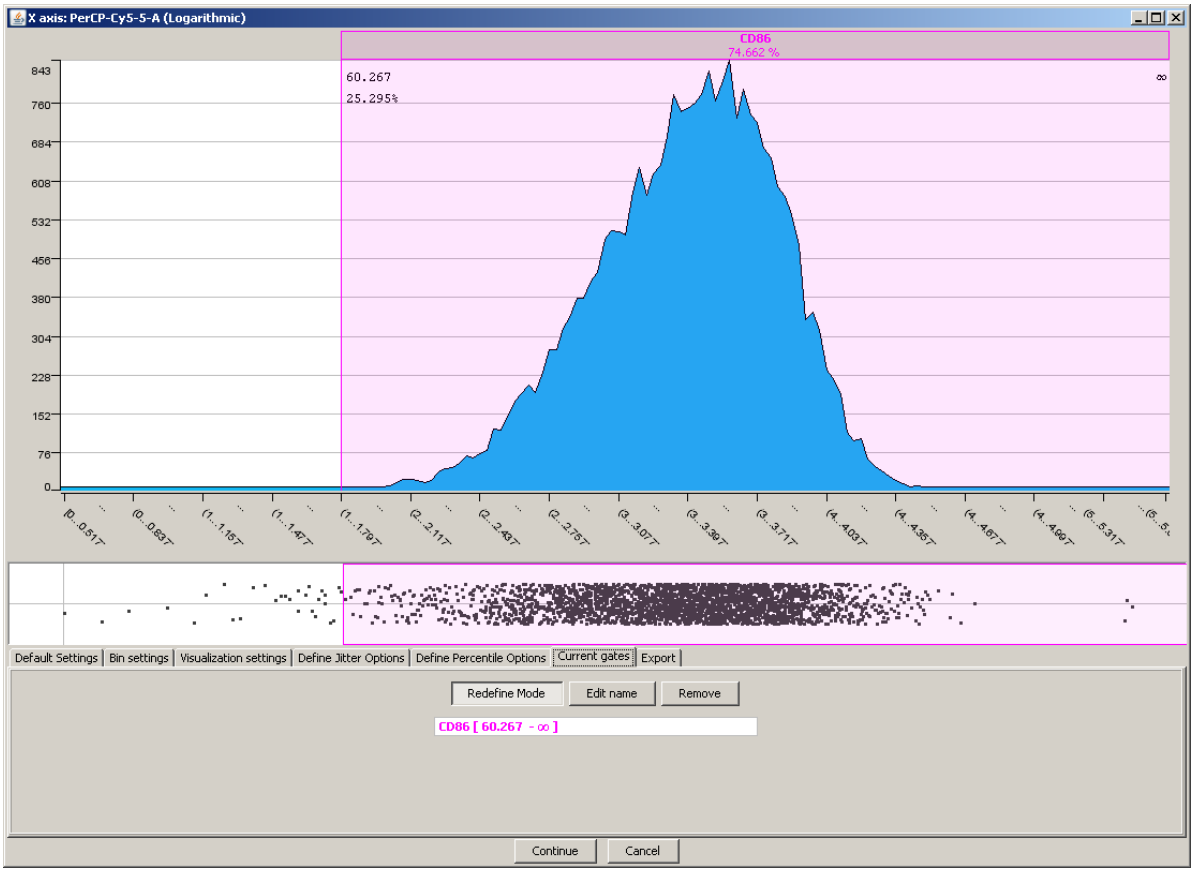


Figure 53 – An example of a Redefine Rectangle Gate node being called

Once through this section, each node will send its gating data to the main part of the workflow.

In the meantime, the I/O nodes will have done their work and will have retrieved the table of content from the ACS file containing the raw data. A list will have been created of all raw data wells within the file.

The well IDs in the list will be fed through the main loop one by one. Within the loop, the well ID will be used to read in the data and perform all the calculations on the raw data.

This was a situation which we had not anticipated in our node design and the nodes were as such modified. While originally needing a configuration, they are now automatically configured by flow variables set by the loop nodes.

The result is that, instead of reading an entire plate at once and then performing calculations on this massive chunk of data, the loop makes sure that the I/O and calculations are done well per well. When dealing with large data plates, this method makes sure that the user does not need enormous amounts of RAM to store all the data.

Once calculations are finished, features are calculated and output is generated. Output exists out of:

- A CSV file, which contains the raw data per well
- Features written to database
- Update of the ACS file with new data

6 INTEGRATION WITH PHAEDRA

This chapter will provide you with more details on the integration of our workflows with the overarching research application PHAEDRA. You will notice that our workflows have become a genuine part of the PHAEDRA system.

During our internship, the integration of our workflows was not our task but that of a PHAEDRA programmer. The following information is an excerpt from a demonstration.

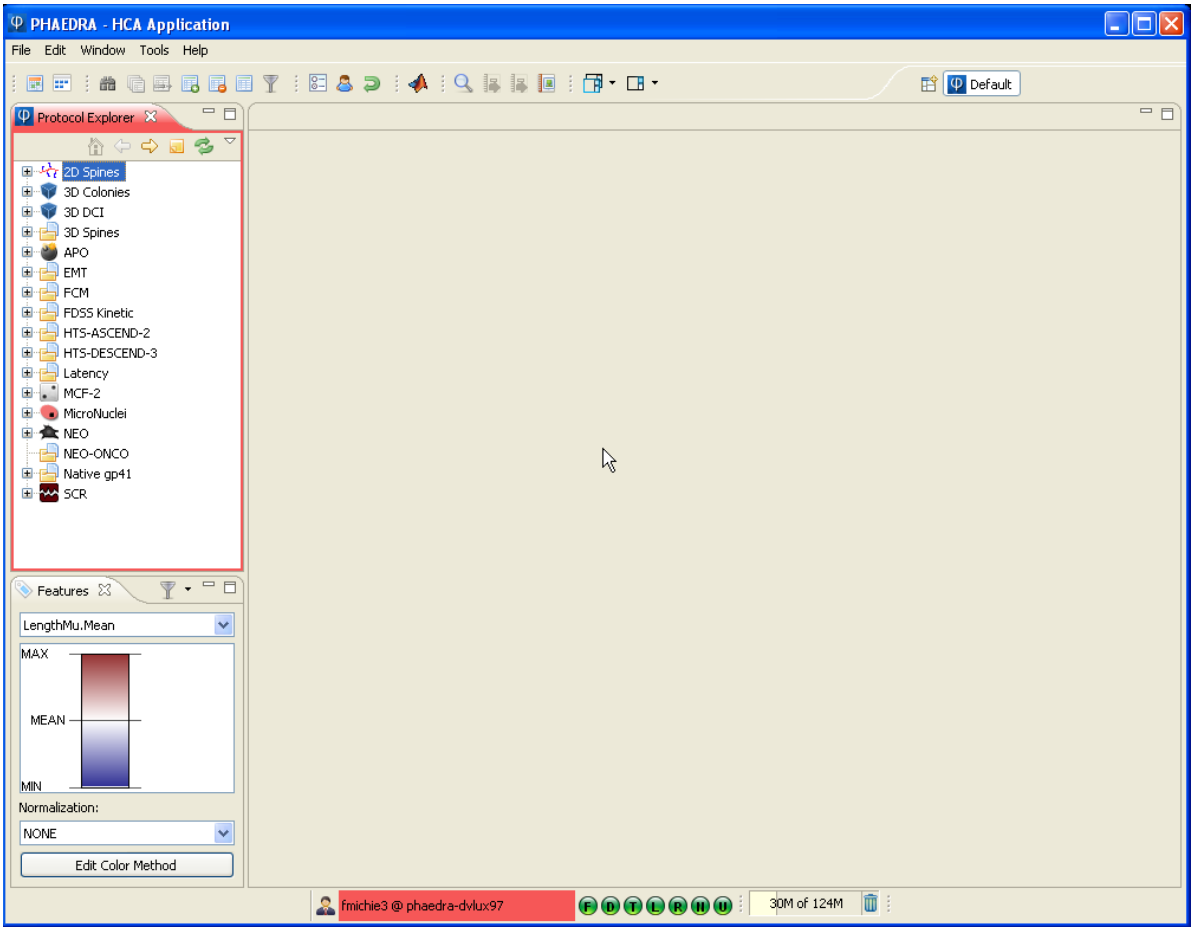


Figure 54 - The default PHAEDRA system

As mentioned before, PHAEDRA's functionality is not limited to Flow Cytometry. The platform gathers its plate information from other existing platforms.

The demonstration provided to us assumed that this step had already been taken, and that the FCS data was already imported.

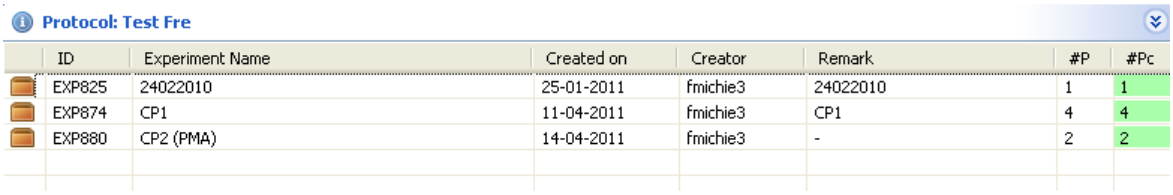


Figure 55 - Showing all experiments within the FCM 'Test Fre' protocol

On the left we see the protocol explorer, which contains a list of all available protocols. In this case, only the Flow Cytometry (FCM) protocol is interesting to us. Once opened, the above picture screen opens up, containing all experiments. The #P column represents how many plates are used within the experiment.

During the demonstration, the CP1 Experiment was used.

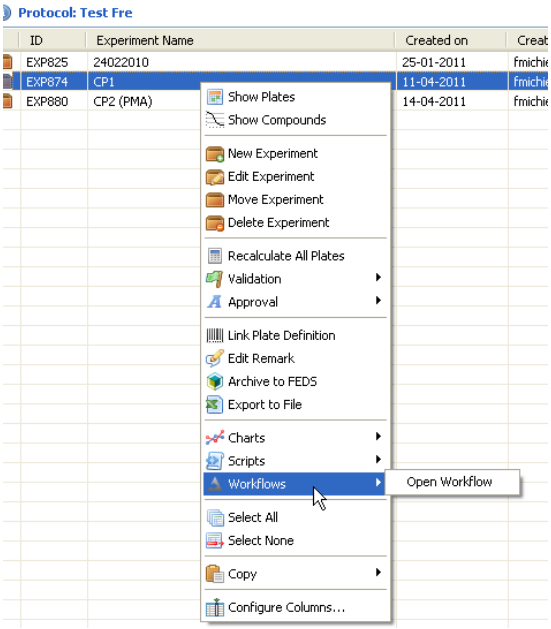


Figure 56 - A context menu allows users to start up KNIME workflows within PHAEDRA

Once an experiment is selected, a context menu pops open containing various options on further handling the experiment data, as pictured above. One of these is called 'Workflows' and contains variations on our KNIME Projects.

Once clicked, a new window will open listing all the workflows that are possible to perform on the experiment data, as seen in the image below.

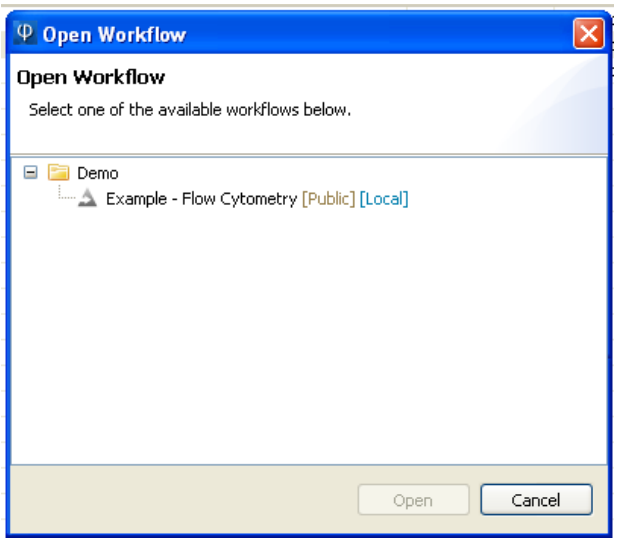


Figure 57 - Different workflows might be available, depending on what calculations should be performed

Once the requested workflow is selected and opened, the KNIME Workflow will open as a new tab within PHAEDRA. Take note that the user never leaves the overarching PHAEDRA application, as pictured below.

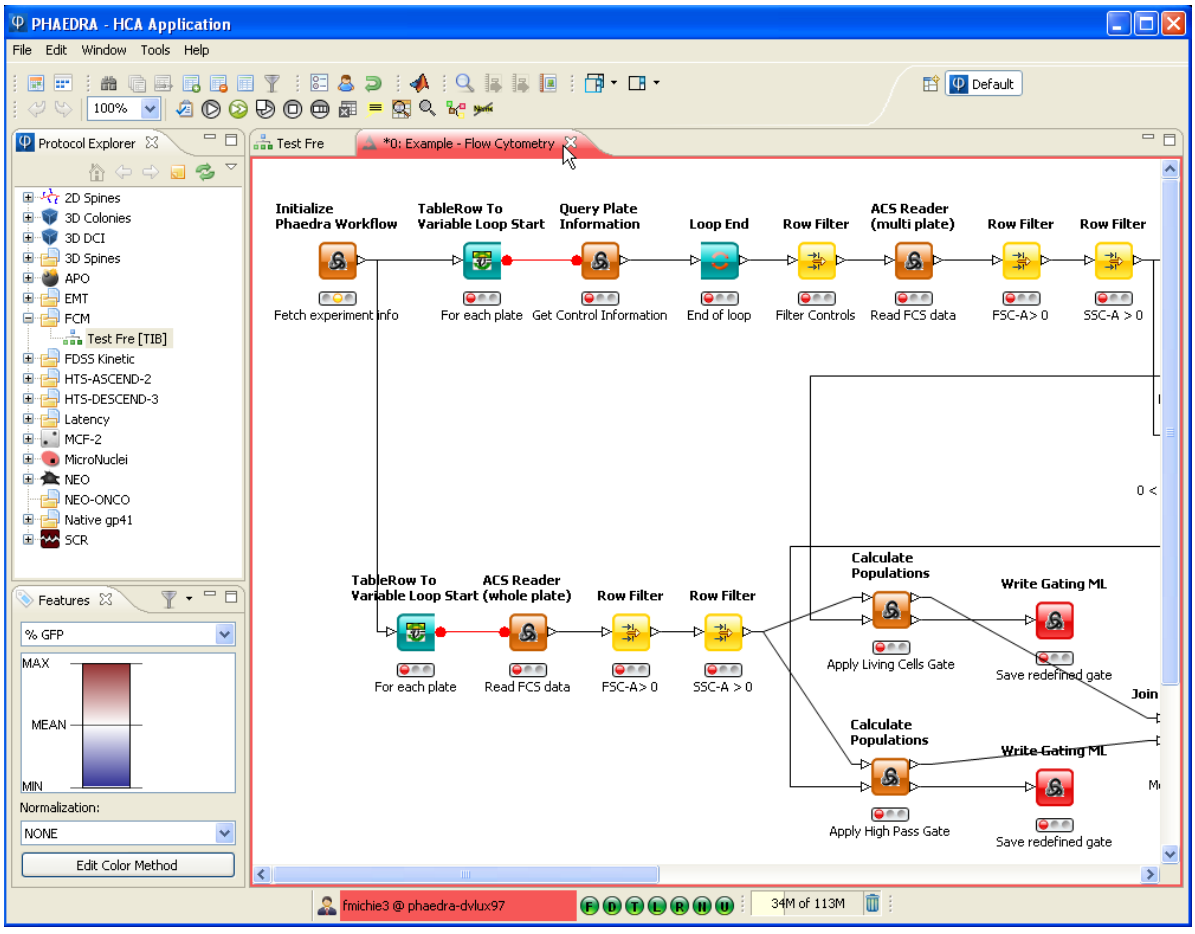


Figure 58 - The Cell Filtering (GFP) Workflow opened within PHAEDRA

The very first node in finished workflows will always be the 'Initialize Phaedra Workflow' node. This node was created by a PHAEDRA programmer and retrieves all data from the previous perspectives and converts them into usable flow variables. This data includes file location and plate IDs, allowing our I/O nodes to retrieve the requested data.

The user is only required to press the 'Execute All' button. The workflow takes care of the required analysis.

During the workflow process, the original files and relevant database entries will be updated with the new information. As such, once the workflow is completed, the user can return to PHAEDRA and work with updated data.

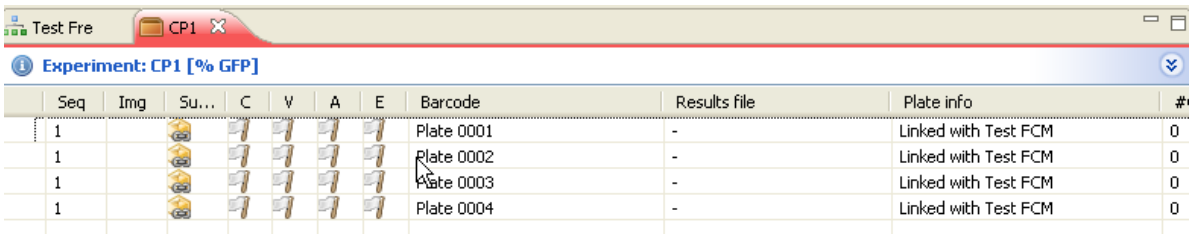


Figure 59 - Experiment information updated with workflow calculations

In the above picture, the workflow was finished and the user proceeded to open the experiment, now with feature data that was calculated in the KNIME Workflow.

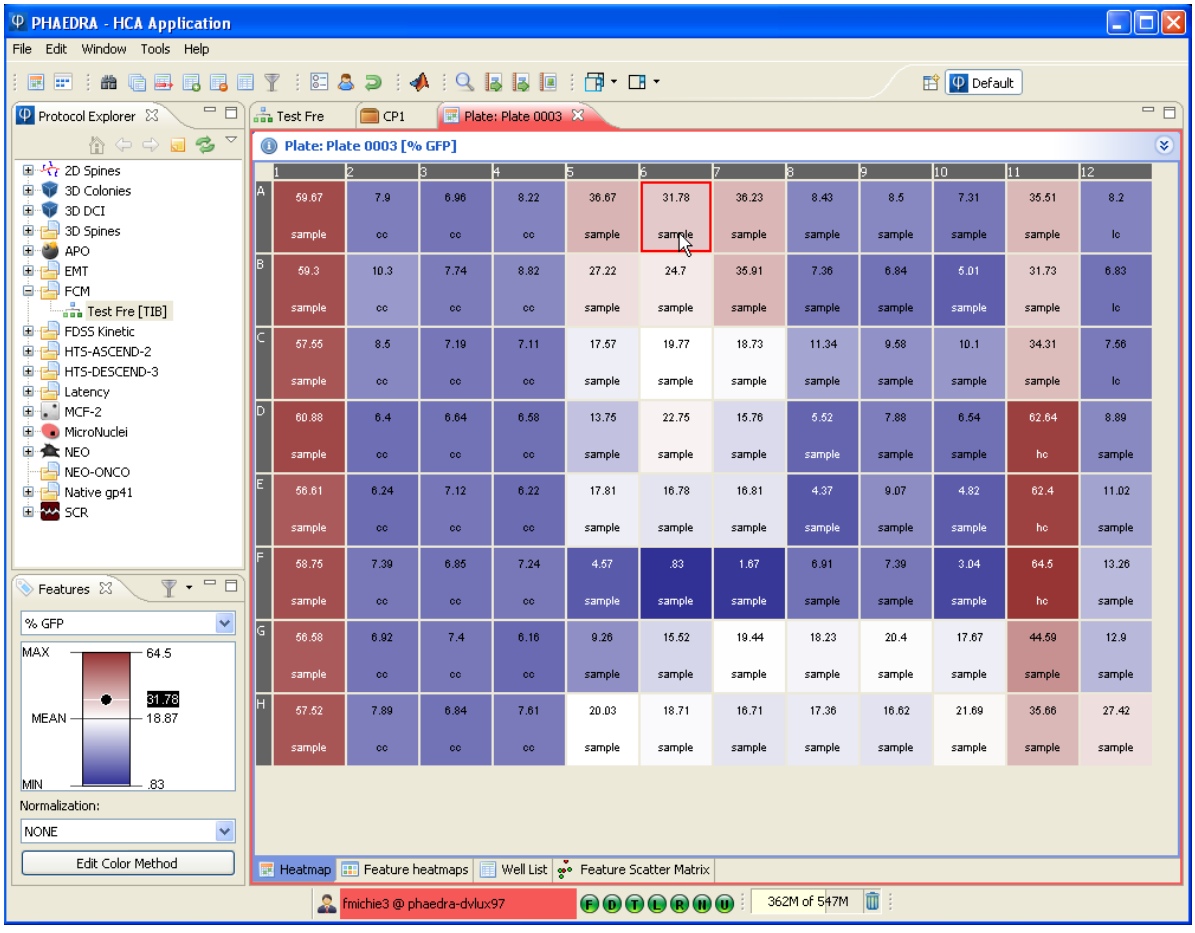


Figure 60 - A heat map generated with calculated feature data from the workflow

When a plate is now opened, a heat map is generated with the data that was created by our workflows. With the 'Features' combobox on the left, users can view different calculated features. Every square on the heat map represents a well within the plate.

With this data, further analysis can be performed within PHAEDRA, however this reaches outside of the scope of this thesis.

As shown, our KNIME Workflows are integrated solidly within the overarching system and automate a piece of the analysis that can be performed by scientists.

7 CONCLUSION

During my internship I have learned a great deal and while the project was sometimes challenging, I believe this has made me more confident in my worth as a programmer.

Aside from JAVA expertise, I have also had a very good experience with teamwork during this internship. Having a work environment where questions can be asked at any time is a very good learning environment.

This internship was also a good look at how a job within the IT sector works, especially within a large company. This experience is invaluable to my perception on the work field.

I also believe that our team of students was able to genuinely help a project get started within Janssen Pharmaceutica. While they still have a long way ahead of themselves, I feel we were able to save them a considerable amount of time and effort.

8 REFERENCES

Useful sites

Stack Overflow. (s.a.). Available at 26 May 2011 on the internet: http://stackoverflow.com/

Java 2S. (s.a.). Available at 26 May 2011 on the internet: http://www.Java2s.com/

Tutorials used

KNIME Developer tutorial. (s.a.). Available at 26 May 2011 on the internet: http://tech.knime.org/developer-guide

Papers used and referenced

Flowcytometry Sourceforce Community (2009, September 17). *Analytical Cytometry Standard (ACS) Gating-ML component: International Society for Advancement of Cytometry (ISAC) standard for representing gating descriptions in flow cytometry.* Candidate for an ISAC Recommendation, version 090917. Online version available at http://flowcyt.sf.net/gating/latest.pdf

Wiley Interscience (2007, October). *Current protocols in Cytometry (Alternatives to Log-Scale Data Display).*

Scott, David W. (1979). *Biometrika 66: On optimal and data-based histograms*. Oxford, England : Biometrika Trust.

Scalable analysis of flow cytometry data using R/Bioconductor (s.a.). Available at 26 May 2011 on the internet: http://onlinelibrary.wiley.com/doi/10.1002/cyto.a.20746/full

Third party JAVA Classes and implementations used and referenced

TrueZip - Archiving files as virtual file directories. (s.a.). Available at 27 May 2011 on the internet: http://truezip.Java.net/

Compressing and Decompressing Data Using Java APIs (s.a.). Available at 27 May 2011 on the internet: http://Java.sun.com/developer/technicalArticles/Programming/compression/

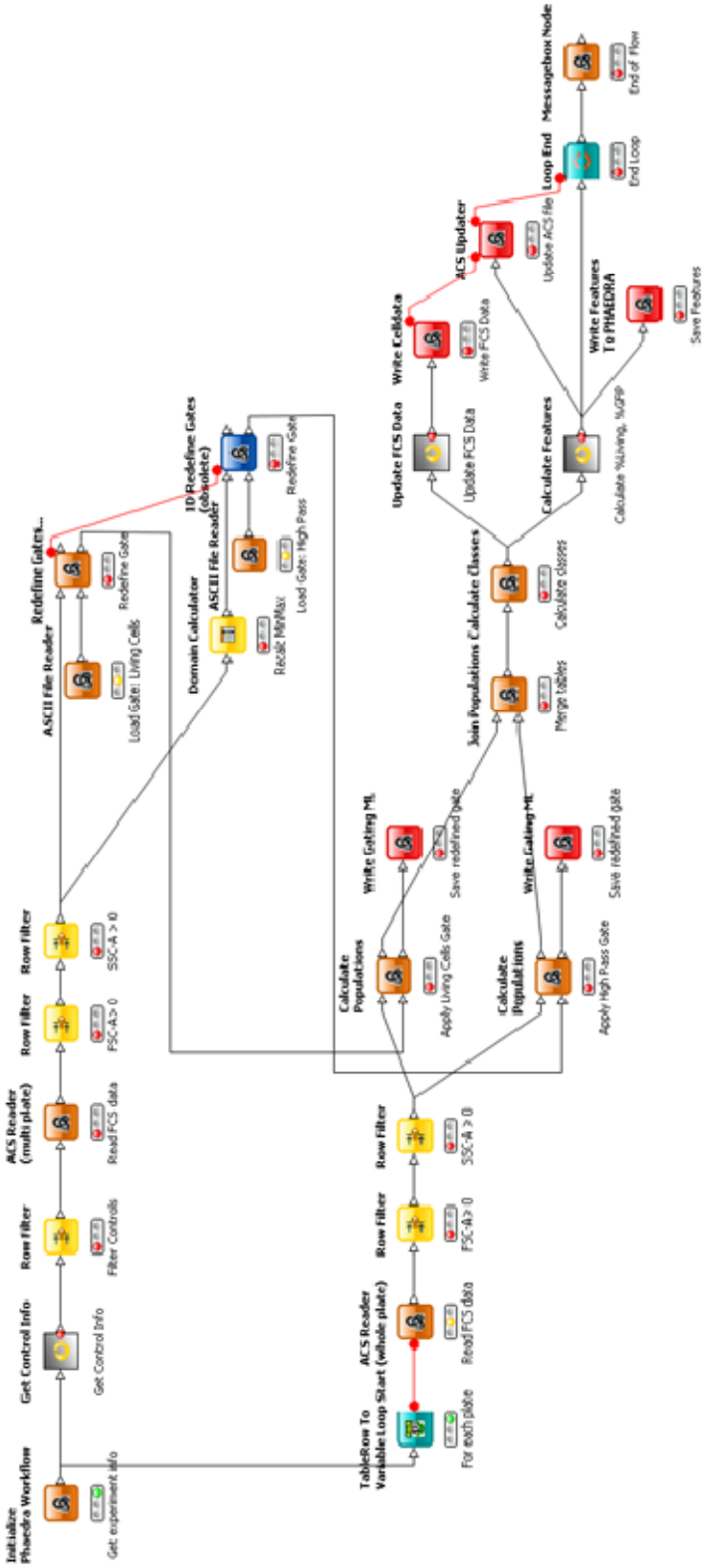
Natural Cubic Splines (s.a.). Available at 27 May 2011 on the internet: http://www.cse.unsw.edu.au/~lambert/splines/natcubic.html

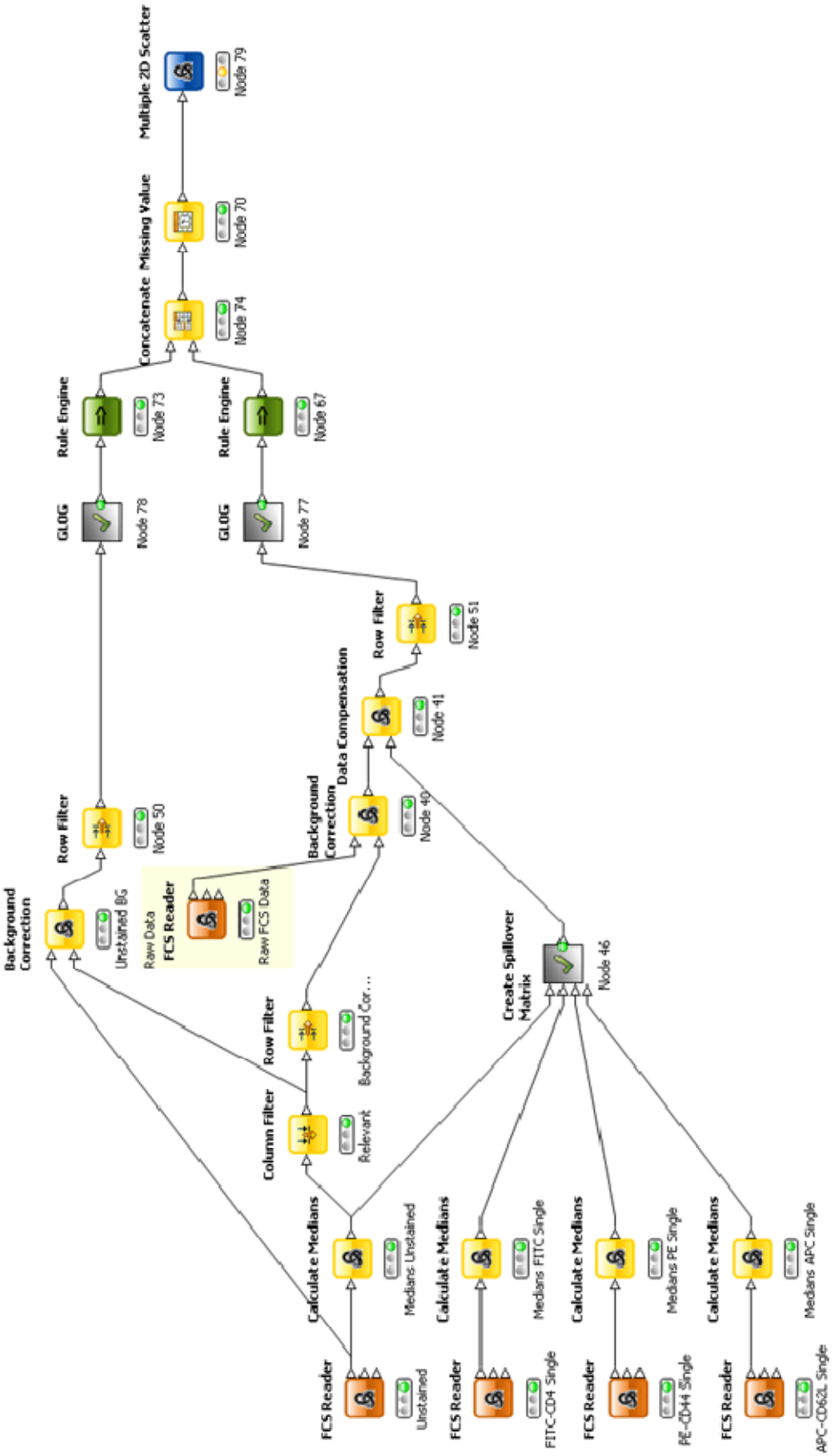
Efficient Java Matrix Library (s.a.). Available at 27 May 2011 on the internet: http://code.google.com/p/efficient-Java-matrix-library/

9 APPENDICES

As appendices, I have chosen to include large size versions of the workflow images provided within the thesis. They follow the same order as discussed in the text:

- Cell Filtering Workflow (GFP)
- Compensation Data Workflow
- Purified Cells Workflow


Initialize Phaedra Workflow
Get experiment info
Get Control Info
Get Control Info
Row Filter
Filter Controls
ACS Reader (multi plate)
Read FCS data
Row Filter
FSC-A> 0
Row Filter
SSC-A > 0
Redefine Gates...
ASCII File Reader
Redefine Gate
Load Gate: Living Cells
Domain Calculator
1D Redefine Gates (obsolete)
ASCII File Reader
Redefine Gate
Recalc Min/Max
Load Gate: High Pass
Calculate Populations
Apply Living Cells Gate
Write Gating ML
Save redefined gate
Join Populations
Calculate Populations
Apply High Pass Gate
Write Gating ML
Save redefined gate
Table Row To Variable Loop Start
ACS Reader (whole plate)
For each plate
Read FCS data
Row Filter
FSC-A> 0
Row Filter
SSC-A > 0
Update FCS Data
Update FCS Data
Write Celldata
Write FCS Data
Calculate Features
Calculate %Living, %GFP
Write Features To PHAEDRA
Save Features
Join Populations
Merge tables
Calculate Populations
Calculate classes
ACS Updater
Update ACS File
Loop End
End Loop
Messagebox Node
End of Flow


Multiple 2D Scatter
Node 79
Concatenate
Missing Value
Node 70
Node 74
Rule Engine
Node 73
GLOG
Node 78
Rule Engine
Node 67
GLOG
Node 77
Row Filter
Node 50
Row Filter
Node 51
Background Correction
Unstained BG
FCS Reader
Raw Data
Raw FCS Data
Background Correction
Data Compensation
Node 40
Node 41
Row Filter
Background Cor...
Column Filter
Relevant
Create Spillover Matrix
Node 46
Calculate Medians
Medians Unstained
Calculate Medians
Medians FITC Single
Calculate Medians
Medians PE Single
Calculate Medians
Medians APC Single
FCS Reader
Unstained
FCS Reader
FITC-CD4 Single
FCS Reader
PE-CD44 Single
FCS Reader
APC-CD62L Single

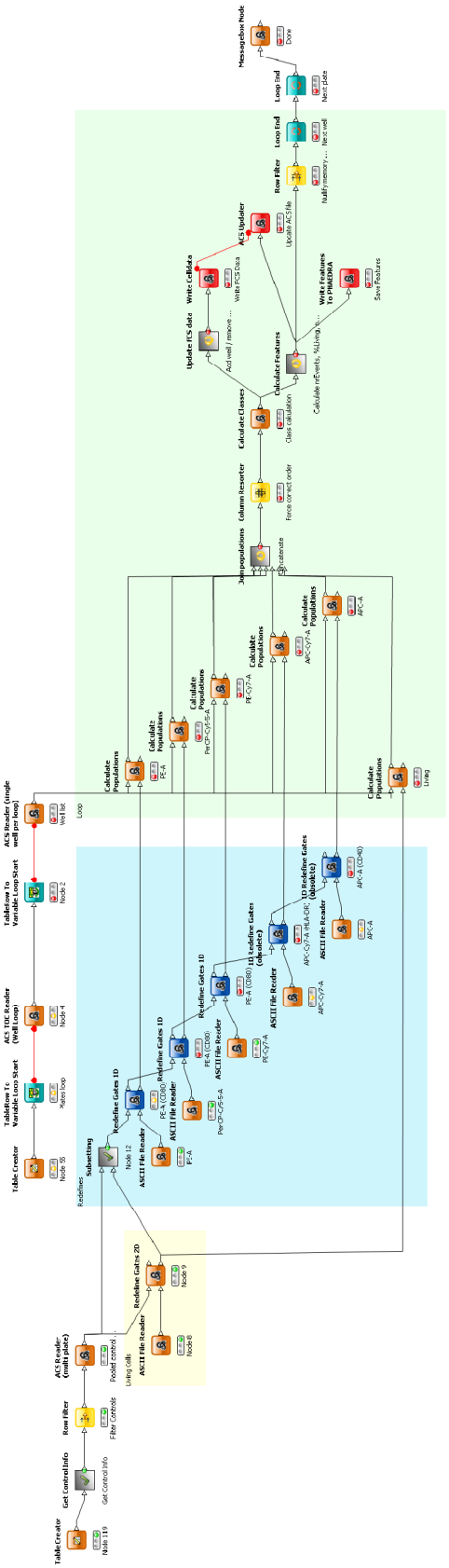


Table Creator
Get Control Info
Row Filter
ACS Reader (multi plate)
Redefine Gates 20
Redefine Gates 1D
ACS TDC Reader (Well Loop)
TableRow To Variable Loop Start
ACS Reader (single well pos loop)
Calculate Populations
Join populations
Column Resorter
Calculate Classes
Calculate Features
Write Cellidea
ACS Updater
Row Filter
Loop End
Loop End
Message Box Node