Computational Methods for High-Throughput Transcriptomic Data

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von

Florian Battke

aus Stuttgart

Tübingen
2012

Tag der mündlichen Qualifikation:	25. September 2012
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatterin:	Dr. Katja Nieselt
2. Berichterstatter:	Prof. Dr. Daniel Huson
3. Berichterstatter:	Prof. Dr. Joachim Selbig

Zusammenfassung

Transkriptomanalysen sind ein wichtiges Werkzeug für die Erforschung der biologischen Mechanismen, mit denen sich Organismen an Veränderungen in ihrer Umwelt anpassen, sowie für die Identifikation von Genen, die für die Entstehung von Krankheiten wichtig sind. Darauf aufbauend können molekulare Angriffspunkte für neue Medikamente bestimmt, die biochemische Produktion optimiert, und vor allem Einblicke in die fundamentale Arbeitsweise biologischer Zellen gewonnen werden.

Microarrays waren die ersten Hochdurchsatzmethoden zur parallelen Bestimmung der Expression tausender Transkripte. Sie werden durch neue Methoden der RNA-Sequenzierung ergänzt, die Daten von neuer Art und in viel größerem Umfang erzeugen. Bioinformatiker sind dadurch mit neuen Herausforderungen konfrontiert: Daten verschiedener Typen müssen integriert werden, eine große Zahl von Methoden für die jeweiligen Analyseschritte müssen kombiniert werden, und Visualisierungen der Daten im Zusammenhang mit Annotationen müssen mit statistischen Verfahren zusammengebracht werden. Zusätzlich sind in Anbetracht der großen Datenmenge spezialisierte Datenstrukturen für effiziente Berechnungen notwendig.

In dieser Dissertation werden Ansätze zur Bewältigung einiger dieser Herausforderungen vorgestellt. MAYDAY, ein Programm zur Visualisierung und Analyse von Microarray-Daten, wurde zum großen Teil neu entwickelt, um eine umfassende Anwendung für Expressionsanalysen zu schaffen. Das neue MAYDAY baut auf einem flexiblen Plug-in-Management auf, kann Annotationen mit Transkripten, Experimenten und Datensätzen verknüpfen, enthält ein interaktives System zum Filtern anhand einer Vielzahl von Kriterien und bietet interaktive, miteinander verbundene Visualisierungen, die für die Analyse und Erkundung von hochkomplexen Datensätzen unerläßlich sind. Darüber hinaus erlaubt die Integration interaktiver Scripting- und Abfragesprachen, darunter die Statistik-Sprache R, auch die Durchführung sehr spezieller Analysen. Die Anbindung von MAYDAY an Gaggle ist ein erster Schritt in Richtung kollaborativer Analysen über das Internet.

Auf dieser Grundlage wurde SEASIGHT in MAYDAY entwickelt, womit sich Rohdaten aus Microarray-Experimenten, sowie Daten aus den neuartigen RNA-Sequenzierungs-Experimenten normalisieren und gemeinsam verarbeiten lassen. Die Entwicklung dieser Erweiterung stellt einen der Hauptinhalte der Dissertation dar.

Desweiteren wird ein Algorithmus für die effiziente Berechnung von Expressionswerten aus RNA-Sequenzierungs-Daten vorgestellt, mit dem diese neuen Verfahren auch ohne bekannte Genomsequenz angewendet werden können, was ihren Anwendungsbereich auf Proben von nicht kultivierbaren Organismen erweitert.

In der Verbindung mit SEASIGHT stellt das neue MAYDAY die erste frei verfügbare Software dar, die den gesamten Analyseprozess der Transkriptomik abgedeckt, beginnend beim Import von Rohdaten, über Normalisierung, Filterung und statistische Tests, bis hin zu komplexen Analysen und interaktiver Visualisierung.

Neue Entwicklungen auf dem Gebiet der Transkriptomik sind auf dieser soliden Basis leicht zu integrieren. Insbesondere für die Systembiologie wird MAYDAYs integrativer Ansatz immer wichtiger, um die Vielzahl unterschiedlicher ‘omics’-Daten in einem gemeinsamen Analyse-Framework zu vereinen.

Abstract

Transcriptome analyses are an important tool for studying the biological mechanisms behind the ability of organisms to react to changes in their environment, as well as to elucidate which genes play important roles in diseases such as cancer. They can be used to find targets for drug design, to optimize the output of biochemical production, and, most importantly, to gain an understanding of the fundamental functioning of living cells.

Microarrays have opened the door for high-throughput expression experiments of thousands of transcripts. Recently they have been complemented by RNA sequencing methods which produce new types of data and a significantly larger data volume. Bioinformaticians are confronted with many challenges of integration: Data of different types need to be integrated, many methods for different analysis steps have to be put together, and visualizations of primary and meta data need to be combined with statistical approaches to derive meaningful results from the data. In addition, specialized data structures are required for efficient computations.

In this dissertation, solutions to several of these challenges are presented. Mayday, a framework for visual inspection and analysis of microarray data, was largely redesigned to create a strong platform for transcriptome analysis. The new Mayday includes a flexible plugin system, a framework for handling meta information associated with transcripts, experiments, or whole datasets, as well as an interactive system for filtering lists of transcripts according to a large variety of criteria. A new visualization package was implemented as a basis for the highly interactive, linked views which are vital for the analysis and inspection of complex datasets. Furthermore, interactive scripting and querying possibilities were added based on different programming languages, most notably the statistical computing language R. With these, bioinformaticians can quickly test ideas and perform non-standard analyses directly inside Mayday. A first step in the direction of on-line collaborative analysis is presented with Mayday's integration into the Gaggle communications system.

With the new Mayday as a solid foundation, the SeaSight extension was developed, which is the main focus of this dissertation. It provides a generic framework for raw data processing both for the new RNA-seq data types as well as for data generated by different microarray platforms.

In addition, an algorithm for the efficient processing of RNA-seq data is presented which allows for the application of this new technology to samples from species where a genome reference sequence is currently not available, adding a further method to the transcriptomics researcher's toolkit.

Together, the new Mayday and SeaSight provide the community with the first software tool which offers a one-stop solution for transcriptome data analysis, spanning the whole pipeline from raw data import, via filtering and statistical testing, to higher-level analyses and interactive visualization, and provides a solid foundation for further development in the transcriptomics area in particular, and in the Systems Biology field in general where the multitude of 'omics' data increase the need for integrated approaches to data interpretation.

Acknowledgements

I am extremely grateful to Dr. Kay Nieselt for introducing me to this highly interesting field of research, giving me large freedom in choosing projects and exploring ideas, for discussions and comments which greatly improved the shape of this dissertation, and, most importantly, for being an outstanding supervisor. In addition, my special appreciation goes to Prof. Daniel Huson for co-supervising my dissertation.

For efficient collaboration on publications, memorable parties and social activities, I owe gratitude to Alexander Herbig, Stephan Symons, Günter Jäger, and Aydın Can Polatkan. In particular, I am grateful to Alexander Herbig for constructive and delightful discussions, shared conference experiences, as well as late-night paper submission activities. Stephan Symons' unparalleled theoretical and practical work on Mayday was an invaluable contribution to the project, and is only exceeded by his positive outlook on doom which raised my spirits many times. The project is taken towards new destinations by Günter Jäger who was a most pleasant office co-inhabitant and whose expertise and efficiency I appreciated immensely. Sabine Gebert's cheerfully applied organizational skills were a great help in planning and realizing conference visits, as well as local teaching and additional events.

I very much enjoyed working in the Integrative Transcriptomics group and in the larger context of the Center for Bioinformatics and hereby thank all who contributed to this feeling, choosing as representatives of this group Lena Feldhahn, Marc Rurik, Daniel Lehle, and Stefan Raue whose prolonged stay in our group was highly welcome. I am also grateful to Prof. Oliver Kohlbacher for engaging discussions on and off topic. Though not part of the Center for Bioinformatics, Ralph Holz has been a great support during his time in Tübingen.

Furthermore, I am obliged to our partners in the SysMO Stream project consortium, with whom we produced and interpreted the data on the metabolism of *Streptomyces coelicolor*, most notably Prof. Trond Ellingsen and Øyvind Jakobsen (SINTEF Materials and Chemistry, Norway), as well as Dr. Alexander Wentzel (Norwegian University of Science and Technology), and Prof. Wolfgang Wohlleben (Tübingen). I value Dr. Michael Bonin and Dr. Michael Walter of the Microarray Facility Tübingen as excellent collaboration partners in the SysMO project and beyond. Together with them, I thank Stephan Körner, Joachim Zühlke, and Dr. Steffen Hüttner (Hölle & Hüttner AG, Tübingen) for successful cooperation on the Passage project. For help with implementing my improvements to the Rank Product implementation, I am indebted to Dr. Fangxin Hong (Harvard University).

Thanks are also due to Lucía Spangenberg-Torre and Dr. Hugo Naya, both now at the Institut Pasteur de Montevideo, Uruguay, for interesting questions we addressed together both in Tübingen and on-line, as well as for the good time we spent together when organizing and realizing the "Advanced Transcriptomics" block course in Uruguay. I am grateful for the experience of working in the field of visualization with Julian Heinrich, Corinna Vehlow, and Prof. Daniel Weiskopf (VISUS, University of Stuttgart). I also learned a lot during the analysis of *Parhyale hawaiiensis* expression data which was joint work with Peter Nestorov (Friedrich Miescher Institute, Basel) and Dr. Matthias Gerberding (University of Hohenheim).

The Mayday project is the work of many people over several years. Nils Gehlenborg and Janko Dietzsch deserve praise for initiating the project and continuing its development. Important parts of the core were developed by Matthias Zschunke and Stephan Symons, and Christian Zipplies played a major role in the implementation of the genome browser visualization. Claudia Broelemann is to be mentioned for her work on the Gaggle integration, as well as Tobias Ries for his outstanding work on the JavaScript console. The project has greatly benefited from contributions by Günter Jäger who will continue the development of Mayday's core as well as new applications beyond the original scope of expression analysis. Contributions from Anna Jasper, Christian Sieber, Frederik Weber, Kirsten Heitmann, Markus Riester, Matthias Munz, Nastasja Trunk, Michael Piechotta, Philipp Bruns, Roland Keller, Sina Beier, Stephan Gade, and Dilek Tuncbilek in the form of plugins or prototype implementations of new methods, and feature requests and bug reports submitted by users of our software have helped move the project forward and are also gratefully acknowledged.

Finally, and extending far beyond my PhD period and the writing of this dissertation, I thank my family and friends with all my heart.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

All living organisms react to external and internal stimuli on a cellular level, altering their cells' state to adapt to changes in their environment, or in the course of the cells' live cycle, for instance. Such adaptations, which are usually tightly regulated, involve changes to the abundances of functionally active molecules such as proteins and non-coding RNA. As protein levels depend on the transcription (and translation) of genes stored in the organism's genome, and because the quantification of active proteins is a non-trivial endeavour, RNA abundances are often used as a proxies for protein levels, as well.

Transcriptome analyses are performed to obtain a so-called expression level for each transcript, and to compare these levels between samples taken from different populations, for instance representing healthy and diseased individuals. On a larger scale, *Systems Biology* studies aim at integrating these data with information about protein and metabolite concentrations, epigenetic modifications, and a range of meta data describing knowledge about cellular processes, among others, in order to ultimately create a full model of the functioning of living cells.

Starting with the *genome* as the complement of an organism's gene, new terms were coined for comprehensive sets of (most generally) observable items of a certain class: The complement of all transcribed RNA molecules forms the *transcriptome*, the set of all proteins the *proteome*, the metabolites make up the *metabolome*, and the collection of all observed phenotypes is called the *phenome*, to give only a few examples. The corresponding research fields are *transcriptomics*, *proteomics*, *metabolomics*, *phenomics*, etc.

Where molecules are studied, these relatively young fields in biology are based on so-called *high-throughput technologies* which enable researchers to identify and quantify thousands of different molecules in parallel. Depending on the field, these molecules can all be of the same type, or cover a wide range of chemical compounds. In transcriptomics, the RNA molecules studied are very similar, and microarray or RNA sequencing experiments can be applied to quantify all RNA transcripts in hundreds or thousands of samples. A much higher chemical diversity is exhibited by proteins, and metabolites such as nutrients are the most diverse set of compounds.

With the growing number of experimental methods and the increasing volume of data that results from high-throughput experiments, bioinformaticians are faced with several challenges. Firstly, raw data come in a variety of formats which software tools need to be able to process and relate with associated meta data. Secondly, each experimental technique has its own advantages and limitations which need to be taken into account during pre-processing, as different sources of technical bias require appropriate normalization methods. Thirdly, human researchers are no longer able to inspect each individual data point acquired in their experiments. They need methods which offer meaningful, visual *summaries* of the data based on which interpretations and new hypotheses can be formulated.

Solutions to these challenges are being developed as bioinformaticians are working with the new data types. Many of these solutions are specific to a particular study and only cover one or a few aspects of interest in the context of that project. In general, however, researchers do not want to assemble a large library of highly specialized programs before trying to answer their research questions. As most programs have quite specific requirements regarding data formats, a lot of time is spent on data conversion which can range from trivial re-formatting to sophisticated computational transformations. Apart from being very costly in terms of time and posing the danger of introducing errors in the analysis, such ‘manual pipelines’ also demand a high willingness to learn about each program’s peculiarities.

Thus, a combination of useful methods from different areas, such as normalization, data processing, statistical testing, and visualization in one *analysis framework* is highly important for efficient research. In part, this has already been realized in the past: The combination of statistical and visual analyses, often called *Visual Analytics*, has been found to be a powerful method enabling researchers to dissect complex datasets, formulate and test hypotheses, and gain an understanding of the underlying biological process.

However, the benefits of integration extend beyond processing and combination of heterogeneous primary and meta data into a common view subject to the methods of visual analytics. Today, as large studies are usually performed by several research groups, collaboration between researchers (working with data form different experimental procedures) should be made as simple as possible, and integration of their results becomes an important factor. Thus, integration should start with the integration of raw data from different sources, cover the integration of different analysis and visualization methods, and extend up to the integration of results from different researchers.

In this dissertation, a unified approach towards solving the challenges of integration is presented. The application platform chosen for this endeavour is the microarray data analysis program Mayday, developed by Kay Nieselt’s group at the University of Tübingen. Mayday focuses strongly on visual analysis, offering a range of highly-configurable visualizations. These are complemented by statistical and machine-learning methods for supervised learning, and by unsupervised clustering algorithms which try to find structure in the vast amount of data produced by modern high-throughput transcriptomics experiments.

In the course of my PhD studies, the existing application was fundamentally rewritten and extended, transforming the original microarray analysis software into an extensible and powerful environment for visual analytics of omics data. The new Mayday *integrates* a large number of methods into one user interface, *integrates* statistics and visualization, *integrates* meta information with primary data visualizations, *integrates* data from different sources and of different types, *integrates with* other Systems Biology analysis programs and supports the *collaboration between* researchers.

The thesis is organised as follows: Chapter two introduces the biological and statistical basis for the problems addressed in this work. Chapter three presents the new Mayday which is the foundation for developments in the fields of visualization (described in chapter four), interactive bioinformatics analyses (chapter five) and collaborative data analysis (chapter six). The new Mayday core also prepared the ground for the integration of RNA-seq and raw microarray data, the purpose of Mayday SeaSight, whose design, implementation and application are presented in chapter seven. A special protocol and algorithm for efficient generation of transcriptomics data using high-throughput sequencing is presented in chapter eight. The final chapter offers a discussion of the achievements presented here as well as an outlook on possible future work.

2. Transcriptomics

Organisms can react to external and internal stimuli at a cellular level. For unicellular organisms, examples of such stimuli are changes in nutrient availability (external) and cell-cycle dependent changes during growth and replication (internal). If cells are part of tissues and organs in a multi-cellular organism, further stimuli can be signals (external to the individual cell) mediated by ligands such as hormones and by cell-cell contacts.

As a result, the internal state of a cell changes. Knowledge about cellular states and how they can be influenced is of interest

- in medicine, where states are associated with health and sickness and effecting change in the state is the aim of treatment,
- in biotechnology, where states are associated with e.g., productivity and the compounds produced by microorganisms, and environmental as well as genetic changes are used to increase yield or to produce modified compounds, and, most importantly,
- in biology, where the comparative analysis of different states provides insights into regulatory relationships and researchers try to uncover *how cells work* by carefully observing how disruptive changes (e.g., nutrient limitation) affect the measured state of a cell.

The 'state' of a cell is mostly defined by the presence of functional molecules and by their activity. To gain an insight into these activities, the turnover of *metabolites* could be measured, which is done in *metabolomic* studies. Another approach is to measure the abundance of each functional molecule. Most of them are proteins or protein-RNA complexes and their study is the subject of *proteomics*. However, proteomics faces several challenges: Proteins have very different functions, and very different chemical properties. Membrane-spanning proteins, for instance, have lipophilic surfaces, while other proteins active in the cell soma generally have hydrophilic surfaces. Extraction, identification and quantification of the full protein complement of a cell is therefore non-trivial and most studies are confined to between a few hundred and a few thousand proteins.

One solution to this problem is to change the focus from the direct identification and quantification of proteins to identification and quantification of (protein-coding) RNA molecules.

2.1. Gene expression

The information specifying how the cell's functional molecules are synthesized is stored in the cell's DNA sequence. The region storing information about a particular molecule is called a *gene* (a word which first was used to refer to the abstract concept of an inheritable trait [85]) and the complete set of all genes in an organism is called its *genome*, the study of which is the field of *genomics*. The following introduction focuses on so-called protein-coding genes, i.e., genes which provide information on how to synthesize a specific protein by concatenating different amino acids. Another important class are the *non-coding* RNA genes which contain templates for functionally active RNA molecules, and are also of interest in studies of gene expression.

Depending on environmental conditions and the internal state of the cell, certain (protein-coding) genes are *expressed* in the process of *transcription*: A DNA-dependent polymerase (interacting with a number of other factors and working in a larger complex [137]) attaches to the start of a gene's *coding sequence* and creates an RNA molecule by ligating nucleotides (adenine, cytosine, guanine, uracil) such that the growing RNA strand's sequence is *complementary* to the sequence of nucleotides on the DNA, i.e., for each adenine on the DNA a uracil is added to the RNA, for each cytosine a guanine is added (and vice-versa) and for each thymine an adenine is added. Gene transcription is regulated by the presence of so-called *transcription factors* which bind to specific DNA sequences (*binding motifs*) and direct the polymerase to a gene location, often as a result of long *regulatory cascades* that start with the recognition of some environmental or internal condition (e.g., a ligand binding to the cell surface) and ending in the activation or inactivation of a specific transcription factor [36]. Nucleotides in the DNA can be modified (methylated), as can DNA-packaging molecules, so-called *histones*, resulting in DNA that is inaccessible to the transcription machinery, and thus inactivating the genes in the respective region [92].

A transcribed RNA molecule is subjected to *post-transcriptional processing* [44], which can be simple additions at either end of the molecule (5'-cap of methylated guanine, 3'-poly-adenine sequence), the replacement of some nucleotides by others (*RNA editing* [19, 35]) or the rather complex process found in eukaryotes which includes the removal of non-coding subregions, *introns*, in a process called *splicing* [21, 43] by the *spliceosome* [145, 170] or by RNA autocatalysis [34].

The processed mRNA molecule forms the template based on which the ribosome synthesizes the corresponding protein sequence. The *genetic code* [112] maps nucleotide triplets (sequences of three consecutive nucleotides, *codons*) to amino acids. The process is as follows: mRNA binds to the ribosome subunits [126], with the *start codon* `AUG` bound to (modified) methionine-bound transfer RNA at the active site of the ribosome. Transfer RNA (tRNA) molecules with specificity for one (or several) codons are loaded with specific amino-acids [141] and bind to the mRNA at the active site of the ribosome. This binding depends on (partial) sequence complementarity between the mRNA codon and the tRNA *anticodon* loop. The new amino-acid is ligated to the growing protein, the empty tRNA molecule is released, the ribosome moves to the next codon, and the process continues.

During translation, the nascent protein molecule starts assuming a three-dimensional structure which is required for its function. Further processing can be necessary to create an active protein, such as the introduction of disulfide crosslinks and the removal of *signal peptides*, short amino acid sequences which serve as localization signals and guide the protein to its target location within (or outside) the cell [169], for instance into the nucleus (nuclear localization signal [107]). The 'final' product is a protein which might be a structural element of the cell, or might be able to perform an enzymatic function. Many proteins can change their conformation (and, as consequence, their activity) through *activation*, by binding to a co-factor or by phosphorylation [42, 39], for instance.

Summing up, the process of *gene expression* consists of up to five steps: transcription of DNA into RNA, post-transcriptional RNA modification, translation of RNA into amino-acid sequence, post-translational modification of the protein, and activation.

As a result, RNA molecules can be used as proxies for functionally active proteins, as their abundance is correlated [103, 102] with the abundance of the final gene product, the active protein. Some caveats remain [54], considering that some expressed proteins require further modification to be activated (or inactivated), or their activity depends on co-factors which might not be present in the cell.

The advantage of RNA molecules over protein molecules from the point of view of identification and quantification is that all protein-coding RNA molecules have the same function, namely the transmission of information from long-term storage (DNA) to the protein-production machinery (ribosome). This functional identity is reflected in the fact that all RNA molecules share the same chemical properties, which renders their extraction and purification much less complicated. Furthermore, RNA is constructed from four nucleotides, while proteins are constructed from more than twenty different amino acids, which can carry a large number of modifications.

Transcriptomics is the scientific field concerned with the study of RNA molecules that are transcribed from a genomic template. These transcripts fall into several classes: protein coding messenger RNA (mRNA), large functional RNA such as ribosomal RNAs (rRNA), transfer RNA that play a vital role in protein biosynthesis (tRNA), as well as several other classes of RNA which do not code for proteins but have (presumed) regulatory functions (so-called non-coding RNA). Among the latter class are small nucleolar RNAs (snoRNA), small nuclear RNA (snRNA) involved post-transcriptional processing (e.g., splicing), and a growing number of RNA molecules with still unknown function [89].

Besides individual molecules, transcriptomics deals with whole *transcriptomes*, the complete set of all RNA molecules present in a tissue or organism at given point in time. In this work, the focus is on gene expression, i.e., on the set of protein-coding RNA molecules, their abundances under different conditions and at different times, and the regulatory mechanisms that lead to their generation (*expression*).

2.2. Transcript quantification

Methods for transcript quantification can be divided into two classes depending on how transcripts are identified: *Hybridization* methods are based on the fact that complementary single-strand DNA or RNA molecules spontaneously hybridize to form a double-stranded molecule, and quantification is performed by measuring the amount

Table 2.1.: Timeline of transcript identification/quantification methods

Hybridization		Sequencing
*†Southern Blot	1975	Sanger plus/minus sequencing*† [134]
	1977	Sanger/Maxam-Gilbert sequencing*†
	1986	Polymerase Chain Reaction*†
	1990	Automated capillary sequencing*
	1990	Quantitative PCR†
Photolithographic arrays	1994	
Spotted DNA arrays	1995	Serial Analysis of Gene Expression
Ink-jet in situ arrays	1996	Quantitative Real-Time PCR†
Random bead arrays	1998	
Maskless photolithographic arrays	1999	
	2005	Second-generation sequencing
	2010	Third-generation sequencing

Methods can be roughly divided into two classes, those that use hybridization and molecular labeling of DNA/RNA molecules for identification and quantification, and those that determine nucleotide sequences directly and count identical sequences to obtain a quantitative expression estimate. *identification only; †only one or a few molecules in parallel.

of light emitted by fluorophores attached to hybridized molecules. The more recently introduced *sequencing*-based methods first determine (part of) the sequences of expressed transcripts (often also using hybridization and/or fluorescence as part of the process) and then count the number of occurrences of each transcript. A short history of all presented methods is given in table 2.1.

One very popular hybridization-based method for the detection of expressed transcripts is the *Southern Blot* [149]. Very briefly, restriction enzymes and gel electrophoresis are used to create fragments from a DNA/RNA sample and to separate fragments by size which are then transferred to a membrane. A probe molecule labeled with a fluorescent dye is added and allowed to hybridize with its complementary fragment (if present). This technique allows for the identification of a single molecule of interest (or a few different molecules, using different fluorophores), in a semi-quantitative fashion. It was the basis for the development of *microarray* technologies, as described in section 2.3.1.

Based on the *Polymerase Chain Reaction* (PCR, [110]) which allows for the amplification of known DNA/RNA fragments without the need of cloning fragments into bacterial genomes for multiplication, quantitative methods (qPCR, [61]) were developed, culminating in *quantitative real-time PCR* [67], a method with which the abundance of a single molecule of interest can be established in relation to another molecule with very high accuracy. Quantitative real-time PCR uses fluorescent dyes that intercalate into double-stranded DNA molecules, or fluorescently labeled single-stranded probe molecules for quantification.

Sequencing-based approaches rely on methods that can determine the sequence of a given DNA/RNA molecule. Two such methods were introduced in the same year, namely the Sanger method [135] which uses chain-terminating nucleotides with specific fluorescent labels, and the method by Maxam and Gilbert [106] which uses radioactive labels and specific cleavage of DNA molecules. In both methods, the resulting labeled fragments are separated by size using gel electrophoresis and the sequence is read manually from the gel bands.

While the radioactive labeling required four parallel experiments to determine the DNA sequence (one for each nucleotide), the Sanger method with four different fluorophores required only one experiment. These two alternative concepts of nucleotide-specific markers in a single experiment as opposed to a single marker in four nucleotide-specific experiments (separated by time rather than spatially) are still present in modern third generation sequencing methods (see below). Furthermore, the fluorescent dyes require less effort in handling than radioactive ^{32}P markers. As a result, Sanger sequencing was further developed into an automated sequencing method by replacing the gel electrophoresis with capillary electrophoresis and manual sequence read-out by automated detection of fluorescence [153]. This method allowed for the identification of amplified, identical DNA molecules. Amplification is performed either by cloning or by PCR.

Quantification was not possible with these first sequencing methods due to their low throughput, but based on sequencing techniques, the quantification method SAGE (Serial Analysis of Gene Expression, [167]), was developed. The protocol was designed to create short DNA fragments from each expressed gene and ligate them into a long molecule (separated by linker molecules) which was then sequenced. The short fragments were then sorted and counted to obtain abundance estimates for the expressed genes. The modern high-throughput sequencing methods discussed in section 2.3.2 are based on the same concept of sequencing and counting, with a much higher parallelisation and simpler sample preparation protocols.

All mentioned methods have in common that they are only feasible for the identification resp. quantification of a relatively small number of distinct transcripts. Living cells, however, express several thousand different (protein-coding) transcripts at the same time, out of a pool of several tens of thousands of possible transcripts. Thus, new high-throughput methods were needed to study whole transcriptomes in a time- and cost-effective way. These were introduced with an extension of the hybridization methods to whole-genome gene expression microarrays, and, more recently, of the sequencing-based methods to so-called *next generation* or *second generation* sequencing platforms and their application to RNA sequencing (RNA-seq, [173]).

2.3. High-throughput transcriptomics

2.3.1. Microarrays

Using hybridization of DNA/RNA fragments from a sample to known complementary *probe* molecules is the basic principle of DNA microarrays. While in Southern

Blotting, the sample is immobilized and a labeled interrogation probe is added to find its complement, these roles are reversed in microarrays:

Known probe molecules are either deposited (‘*spotted*’) on a substrate (usually a solid surface such as a glass slide), or synthesized directly in place. Both methods result in a regular arrangement of so-called *spots*, locations where millions of copies of identical probe molecules are attached to the substrate. If existing molecules (e.g., molecules from an EST library or presynthesized oligos) are used, the result is called a *spotted array* [140], otherwise the term *in situ (oligonucleotide) array* is used, referring to the fact that in situ synthesized probe molecules are usually relatively short (25-75 bp).

From each sample, RNA molecules are reverse transcribed into *complementary DNA* (cDNA) and amplified using PCR. They are labeled with fluorescent dye molecules and washed over the array surface. When a labeled sample molecule encounters a complementary probe molecule, a two-stranded hybrid forms and the sample molecule is thus bound to the array (given appropriate hybridization conditions). The more abundant a transcript was in the original sample, the more of its molecules can bind to the millions of complementary probe molecules in the respective spot (up to the level of complete saturation). After washing off unbound sample, the dye molecules are excited using laser illumination, the emitted light is captured in a photographic image (‘*scanning*’) and the light intensity computed for each spot is taken as an estimate of the abundance of the respective transcript. Some platforms use a single fluorescent dye and can thus analyze one sample per array, yielding *absolute expression values*, others use two (or more) different dyes to allow competitive hybridization of two (or more) samples, yielding *relative expression values.*

The first commercially available *in situ* arrays were produced with a photolithographic method [53, 118]. Chemically modified nucleotides attached to a protection group are combined into growing strands of DNA by selectively removing the protection group of the existing chain before adding the next nucleotide. The protection groups are cleaved off through irradiation with ultraviolet light. To control the removal of the protection groups, different masks are used that prevent the UV light from reaching all spots. One mask is required for each nucleotide and position in the sequence (e.g., for a probe length of 25bp, up to 100 masks are required). An alternative was later introduced with *maskless photolithographic* array construction [147] using digital micro-mirror devices. Other *in situ* synthesis approaches use piezoelectric deposition of nucleotides, similar to ink-jet printing technology [23].

Microarray experiments rely on the mapping of locations to known probe molecules to identify the transcripts found in the sample. All platforms establish this relationship during the array design phase, but for one: The BeadArray platform [108] uses probe molecules synthesized on the surface of small beads, each of which carries millions of copies of only one probe molecule. Beads are randomly distributed over the array surface and the location of each known probe is established by hybridizing a set of well-defined interrogation molecules (‘*decoding*’) that are complementary to *barcode sequences* added to each probe’s sequence.

Irrespective of the technology used for array production and spot-to-probe mapping, a very important step in microarray design is probe design (*in situ* arrays) resp.

probe selection (spotted arrays). Probe molecules must be complementary to (part of) their *target sequence* and should not be (partially) complementary to another target. If two different probe molecules are too similar in terms of their nucleotide sequence, *cross-hybridization* will occur and confound the expression signal. Furthermore, as the temperature at which two complementary DNA molecules form a hybrid depends on their nucleotide content (with sequences rich in `G` and `C` having a higher 'melting' temperature T_m), probe molecule selection should be optimized with respect to the T_m values. An optimal design thus contains *sensitive*, *specific (unique)*, and *isothermal* probes. This is usually achieved by using more than one probe for each target (sometimes called 'probesets') and computing expression values by summarizing over the signals of all probes specific to a given target. Finally, microarray design requires previous knowledge of the expected transcripts, either explicitly to guide the process of oligonucleotide probe design, or given implicitly by the contents of the probe library used for spotting.

Present-day high-density microarrays contain several hundreds of thousands of spots ('features') which are summarized into several tens of thousands of expression values. To give an example, the Affymetrix Human Genome microarray 'HGU-133 Plus 2.0' contains about 1.35 million features representing about 54,000 probesets [2].

2.3.2. RNA sequencing, RNA-seq

The *next* or *second generation* sequencing methods use a combination of highly-parallel PCR reactions, either in emulsion droplets (emPCR, [180]) or on solid substrates [1] to create millions of copies of each fragment. Highly-parallelized sequencing reactions are used to determine the sequence of each cluster of amplified sequence.

- **'Sequencing by Synthesis'** (SBS) methods [20, 47] use reversible termination and a ('wash-and-scan') cycle of adding (labeled) nucleotides, incorporation into a growing DNA strand by a (modified) polymerase molecule, washing of excess nucleotides and scanning to identify the incorporated nucleotide (using different dyes for the four nucleotides) resp. to detect the incorporation event (using a single dye and only one nucleotide per cycle).
- **'Sequencing by Ligation'** (SBL) methods [122] use well-defined single-stranded *probe* molecules labeled with a fluorescent dye. If the probe is able to hybridize to the template strand in the correct position, it is ligated to the growing strand and identified in the same manner as used in SBS methods.

The result of all methods are so-called *reads*, short (35-750 bp) nucleotide sequences. These are *mapped* to a known reference sequence to identify the transcripts that they derive from (for an alternative strategy, see chapter 8), and the number of reads per transcript is computed.

Recently introduced *third generation* sequencing methods remove the need for fragment amplification completely. Single molecules are sequenced directly, reducing experimental bias (such as the sequence-specific efficiency of PCR [121]) and allowing researchers to sequence even extremely small sample volumes. To achieve this high sensitivity, different approaches are used, some using *optical* detection as in

Table 2.2.: Third generation sequencing technologies

System	essential parts[1]	detection[2]	process[3]	volume limited by
– Currently commercially available systems				
PacBio SMRT [49]	P, N (lab)	fluorescence	synth.	zero-mode waveguide
Ion Torrent [132]	P, N	pH change	synth.	pico-titer plate well
– Systems under development				
Life Tech FRET [81]	P (mod), N (lab)	fluorescence	synth.	FRET radius <10nm
Oxford Nanopore [38]	EN, protein pore	ion flow	degr.	pore diameter
IBM Nanopore [120]	transistor pore	nuc. properties	direct	pore diameter

[1] P, polymerase; N, nucleotides; mod, modified; lab, labeled; EN, exonuclease. [2] nuc. properties, electrical properties of each nucleotide. [3] synth, synthesis of the complementary strand; degr, degradation of the template molecule; direct, sequencing without synthesis or degradation.

second-generation methods, others using *chemical sensors* or measurements based on *physical (e.g., electrical) properties*. Common to all methods is the reduction of the observed sample volume, often down to few picolitres, which is necessary to detect minute events such as the light emitted by a single fluorescent dye molecule or the positive electrical charge of a single hydrogen ion.

Methods can also be grouped as to whether they rely on the use of biological molecules, such as polymerases or exonucleases, or whether they are based purely on artifical parts. A third grouping is possible based on the biochemical process used into methods that synthesize a complementary strand, methods that degrade a single stranded molecule and methods that sequence a molecule directly without degradation or synthesis. Table 2.2 lists several methods with their properties. For more details, see for example the review by Schadt *et al.* [139].

The main advantages of third generation sequencing methods, beside the elimination of the amplification step and direct sequencing of single-molecules, are increased read length compared to second generation methods, and, for some methods, the detection of nucleotide modifications (e.g., methylations).

In this thesis, the focus is on data from microarray and second-generation sequencing experiments.

2.4. Transcriptomics data analyses

2.4.1. Naming conventions

The data processed in transcriptomics research can be seen in the first approximation as being two-dimensional: The first dimension is that of the transcripts (genes, ncRNAs, intergenic regions) under study. The second dimension is spanned by the biological *samples* that are obtained. While the first dimension is easy to describe, the second has more structure and thus more terms that are sometimes used interchangeably, or even with different meanings in different settings.

Table 2.3.: Structure of a transcriptomic study

Study	→ condition a, condition b, ...
Condition[1]	→ time-point t_1, time-point t_2, ...
Time-point[2]	→ biological replicate 1, biological replicate 2, ...
Biological replicate	→ technical replicate 1, technical replicate 2, ...
Technical replicate	≡ 'sample'

Samples can be grouped in several hierarchical levels. [1]Conditions can be multi-dimensional attributes, e.g., different mutants on different media at different temperatures. [2]When only a single time-point is studied, the 'time-point' level in the hierarchy is not used. Instead, one condition comprises several biological replicates.

As the input of the computational analysis of transcriptomics data results from microarray hybridizations (or RNA-seq experiments), the term 'sample' is usually employed to refer to a single data acquisition act (hybridization or sequencing) instead of a single biological sample, as one biological sample may give rise to several *technical replicates*, each of which results in a 'sample'.

Samples can be grouped according to technical and/or *biological replicates* they represent. They can be further grouped by the *condition* that was studied (e.g., healthy samples vs. disease samples). Conditions can be multi-dimensional, for instance when several treatments are evaluated on several different mutant strains of an organism. *Time specifications* are usually not seen as part of the 'condition'. Rather they are seen as a sub-structure of each condition, such that several samples representing different time-points under the same condition constitute a *time-series*. Finally, all samples in all conditions resp. series, that were studied within a specific project, form a 'study' (see table 2.3).

The term '*experiment*' is rather ambiguous. It can be used to describe all samples referring to one time-series, or all samples acquired under a certain condition (or pair of conditions). In microarray literature, the term is often used interchangeably with the term 'sample', to denote a single column in the expression matrix, or to distinguish the concept of an expression matrix column from that of a biological sample.

In the following, the term 'experiment' will be used in the conventional sense to represent one column of the expression matrix, i.e., transcriptomic data obtained by one wet-lab data acquisition act. Several experiments form an expression matrix, or *data set*, (representing one or more conditions, or one time-series) of which several can be analyzed within a larger study. In Mayday, each measured transcript is represented by one `Probe`, irrespective of the type of transcript (gene, noncoding RNA), the technology that was used to produce the numeric value (e.g., summarized Affymetrix probesets are also represented by 'probes', as are RNA-seq quantified expression values for a given locus), or whether the data represents transcription at all (e.g., when microarrays are hybridized with genomic DNA samples or when Mayday is used for proteomic or metabolomic studies).

Table 2.4.: Data pre-processing and normalization steps

	microarrays	RNA-seq
	scanned array images	*scanned images*[1]
	↓	↓
Image analysis	spot detection	spot detection
	pixel value aggregation	sequence determination
	↓	↓
	fore-/background intensities	*read sequences*
	↓	↓
Pre-processing	background correction	mapping against reference
	probeset summarization	counting
	↓	↓
	raw expression values	*read counts*
	↓	↓
Normalization	intra-array (dye bias)[2]	library size
	inter-array (hyb. conditions)	transcript length
	↓	↓
	normalized expression values	*normalized read counts*

Comparison of pre-processing and normalization steps for microarray and RNA-seq experiments. [1]All second-generation sequencing methods are based on optical detection of fluorescence and require a spot detection step to locate each 'cluster' of identical sequences being sequenced. [2]Intra-array normalization is mainly needed for multi-channel arrays. Intra- and inter-array normalization also correct for different sample concentrations.

2.4.2. Data pre-processing and normalization

The first dry-lab step in expression studies is quantification, where the raw data files (usually image data) are analyzed to quantify the expression strength for each transcript of interest. For microarrays, this includes image alignment and spot detection, delineation of spot boundaries and summarization of pixel values inside and outside of the spot area of each spot location. In RNA-seq experiments, image files are analyzed to obtain read sequences and per-base quality values (see table 2.4 for a comparison of the steps required for the two technologies).

Raw data from biological experiments not only reflects the *true* biological phenomena under study, but also contains unwanted components. These are (more or less) random *noise* due to the technical properties of the system used for data acquisition (e.g., optical noise in scanned microarray images) and *experimental bias* due to differences in sample processing (environmental conditions, different experimentators, different labs, different batches of chemicals used, etc.).

Before meaningful analyses can take place, these unwanted effects have to be removed from the obtained signals as much as possible to uncover the *relevant biological signal.* This is typically a two-step process:

1. **Sample pre-processing** is necessary to remove technical noise from the data and to obtain the initial expression estimates for the next step.

For microarray experiments, *background correction* methods are used, to separate the optical signal due to probe hybridization (inside the spot area) from the background noise (outside the spot area) due to unspecific hybridization and scanner noise. Methods range from simple subtraction of background values to modelling signals based on separate foreground and background distributions [148]. Spatial effects (especially for multi-printtip spotted arrays) can also be corrected [183]. Further steps, such as probeset summarization can be necessary to generate the per-transcript expression values.

Read sequences from RNA-seq experiments are mapped against a reference sequence to assign them to transcripts, resulting in a *counts per transcript* value. A separate 'background correction' step is not required for RNA-seq data, as no 'unspecific sequencing' can take place.

2. **Inter-sample normalization** aims at removing technical bias that overlays the biological differences between different samples.

   Normalization of microarray data [124] is strongly dependent on the platform used. If multiple samples are hybridized to one array (dual- or multi-channel arrays), *intra-array* normalization methods are first employed to remove bias introduced by different concentrations of the two (or more) samples, as well as by different 'strengths' (labeling efficiency, light output, scanner sensitivity) of the fluorescent dyes used (see [142] for an overview of possible sources of bias in microarray experiments). After intra-array normalization, or if only one sample is hybridized to each array (single-channel arrays), so-called *inter-array* correction methods are used, with one example being quantile normalization [27] which aims at mapping both samples expression values to the same distribution. These methods try to account for technical bias introduced e.g., by different scanner brightness settings, hybridization conditions or different sample concentrations.

   For RNA-seq data, two factors are considered to influence the count values: The *library size* (total number of reads) has to be accounted for to make a gene's count values from different samples comparable [105], and the *length of the transcript* influences comparisons between different genes' count values within one sample [114]. The RPKM (reads per kilobase of exon model per million of mapped reads) measure [109] corrects for both. Alternative methods have been proposed for library size normalization, such as the more robust quantile method of Robinson and Oshlack [131, 33]. Another source of bias is sequence `GC` content which can even interact with variation in library preparation [129] (see [119] for a discussion of sources of bias in RNA-seq data).

2.4.3. Differential expression & statistical testing

With normalized data at hand, on of the fundamental aims of expression studies is to determine which transcripts are *differentially expressed* between different conditions (or time-points). For simple studies which include only two conditions (e.g., treated vs. untreated), this could be (and has often been) determined by simply computing the mean expression for each gene in each condition and then ranking genes according to the absolute difference between their two mean values.

Instead of deciding on a difference threshold above which genes are deemed sufficiently differentially expressed, a less arbitrary approach is to use a *statistical test* on each gene's values. Such tests (e.g., Student's t-test [150]) rely on the availability of several replicates per condition to take within-condition variance into account when evaluating between-condition variance. The result of applying a statistical test is a *p-value*, expressing the probability that the observed between-condition variance is the result of random sampling from *one distribution* instead of from a true difference between the means of *two distributions*, one for each condition. In other words, the t-test p-value gives the probability of finding a similar or more extreme difference between the conditions when drawing both conditions' expression values from the same (normal) distribution (or, equivalently, from two normal distributions with the same mean). Genes with a p- value smaller than a predefined threshold (most often 0.05) are called *statistically significantly differentially expressed.*

If several tests are conducted, correction for multiple testing is necessary to limit the amount of *false positives* (i.e., genes deemed significantly differentially expressed even though they are not). This is especially important in high-throughput experiments, where tens or hundreds of thousands of tests are performed and validation of predicted positives (e.g., using qPCR) is costly and time-consuming. Several correction methods have been published with different levels of stringency, falling into two categories. The family-wise error rate (FWER) is controlled by methods such as Bonferroni's [28, 29], Holm's [70] or Šidák's [165] which are very strict and often lead to a high false negative rate up to the point where no gene is considered significant at all [48]. Methods controlling the *false discovery rate* (FDR [17, 18]) are less stringent and are often used in expression studies.

The t-test is a *parametric* test statistic, meaning that it makes assumptions about the underlying data, namely that data follows a normal distribution. *Non-parametric* test statistics make no such assumptions, and thus are more suited for data where assumptions might be violated. One example for a non-parametric test is the *Rank Product* method [30], which automatically controls the false discovery rate and also takes dependencies in the data into account. Such dependencies are very likely in expression data, as genes are usually up- resp. downregulated in groups, for instance comprising all genes common to a metabolic pathway.

The list of (statistically significantly) differentially expressed genes is then used for higher-level analysis.

2.4.4. Higher-level analyses

Two points of entry are commonly used to dissect expression data sets, one being the list of differentially expressed (DE) genes obtained by applying some statistical test (i.e., a gene list derived from the data), the other being lists of 'interesting' genes (i.e., lists based on *a priori* information). The latter include lists of genes implicated in the same (metabolic or regulatory) process, genes suspected to be of relevance to the research question under study, genes found to be significant in other studies, etc.

These gene lists can be used as the basis for a large number of computational as well as visual analyses. Very common are unsupervised *clustering* methods which try to group genes together based on their expression profiles to discover potentially co-regulated genes (i.e., genes whose transcription is regulated by the same transcription factors), *pathway analyses* and *enrichment studies* with the aim of finding a pathway or gene category which is (statistically significantly) over- resp. underrepresented in the list of DE genes, and analyses of the genomic location of putative co-regulated genes. An example analysis covering most of these points is presented in section 3.7.

Meta-information, such as functional gene annotations, is often included in the analysis process, for instance to assign transcripts to functional categories for enrichment calculations. It can also be used to enhance data visualizations by introducing additional data aspects and allowing researchers to intuitively perceive relationships between different experimental (and meta) variables which might be unexpected and would not have been picked up by algorithmic approaches. Repeatedly displaying data in rich visualizations based on primary and meta data, generating hypotheses and testing them with further visualizations as well as statistical methods is the scope of *visual analytics*.

In this dissertation, a fundamental redesign and several large extensions of the expression analysis framework Mayday are described which implement the concept of visual analytics for expression data, add and/or improve important visualizations for large-scale data, integrate sequencing-based high-throughput transcriptomics data, and lay the foundation for further developments within the Mayday framework.

3. The new Mayday as a solid foundation

3.1. Mayday's Evolution

In 2003, MAYDAY [46] was initiated by Kay Nieselt, Janko Dietzsch and Nils Gehlenborg who implemented version 1.0 as part of a student project, and was quickly followed by version 1.1. The aim of the project was to fill the need for a visual analytics application for microarray studies. The project became the basis for many other students' work who designed and implemented plugins for the core application. A major step was the release of MAYDAY 2 in 2005 which was the starting point for the work described here. The next release was version 2.5 in 2008 which included a redesign of large parts of MAYDAY's core (see below), with regular releases continually incorporating changes and additions to the core, introducing new functionality (often as a result of students implementing new plugins), as well as improving the user interface [16].

The evolution from MAYDAY 2.0 to the current version 2.12 was motivated by two developments: The rise of Systems Biology where additional aspects (metabolome, proteome) of an organism were described as numerical data similar to the matrix format ('expression matrix') used for transcriptome analyses, and the introduction of powerful high-throughput sequencing technologies which resulted in a completely new form of expression data. Both of these needed to be integrated into the program.

Together with many changes necessary to make MAYDAY more flexible and to allow for efficient analyses of large-scale data, these developments not only significantly changed the user interface (see figure 3.1) and brought the project to well over 300,000 lines of code (see figure 3.2), but also lead to the establishment of a more structured basis on which plugins can be implemented. The loosely coupled pair of a small core and a relatively simple plugin management system in version 2.0 was replaced by the combination of the original core adapted to allow for multi-threaded computations, a large number of data structures (such as flexible vector and matrix classes, structures for the efficient storage of genomic positions, graphs, trees, data structures built around native types as well as memory-mapped structures), a very powerful plugin system currently providing more than 60 extension points (see below), a framework for persistent settings and automated GUI creation as well as a task management system. A common foundation for visualizations was established which greatly simplifies the development of new visualizations and their integration with other plots and MAYDAY's data structures. Figure 3.3 shows how these elements relate to each other. The visualization framework is presented separately in chapter 4.1, the other elements are described in more detail below, followed by a short example of how MAYDAY can be used to analyze complex Systems Biology datasets.

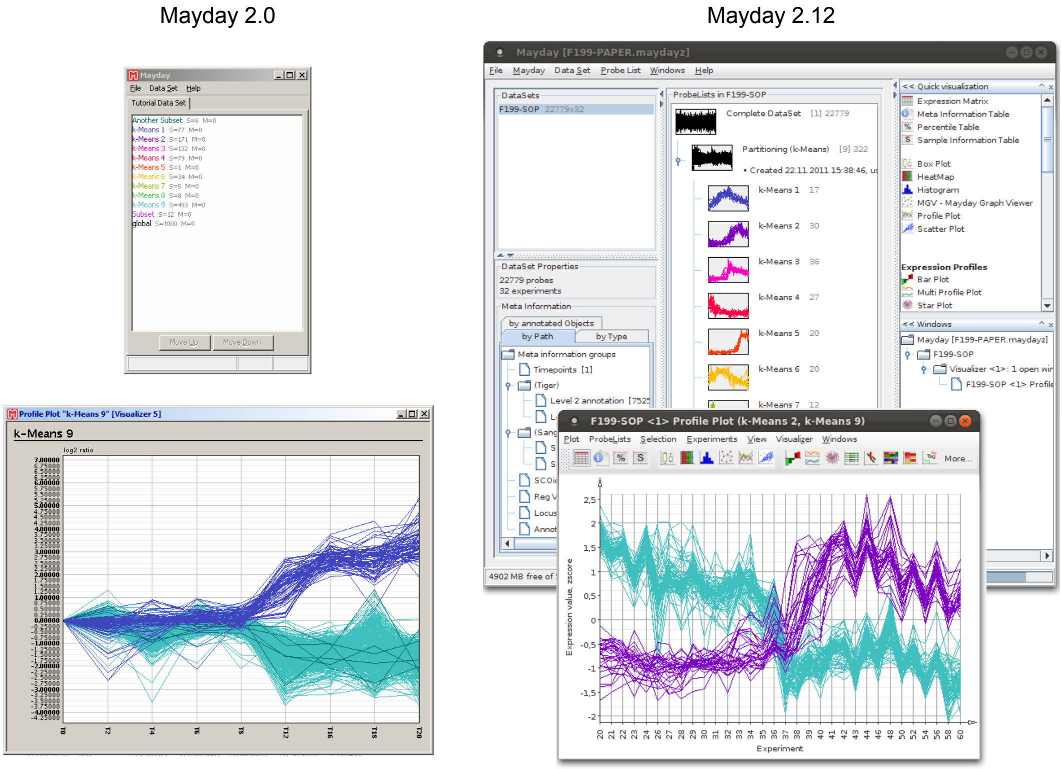


Figure 3.1.: User interface comparison: Compared to the Mayday version 2.0 released in 2005 (left), the current version (right) has a much richer user interface which displays all important elements of Mayday's data model directly in the main window (top part of the figure): If multiple datasets are opened, they are listed at the top left of the window. For the currently selected dataset, meta information is shown on the bottom left side, while the probe lists are displayed in the central area of the window, together with their hierarchical ordering and small preview images which make it easier for users to remember what each probe list contains. Users can include additional view elements (rightmost column) such as a list of visualizations for quick access, or an overview of currently opened visualization windows. The visualization plots (for example the profile plot, bottom part of the figure) also offer many more configuration and interaction possibilities than before.

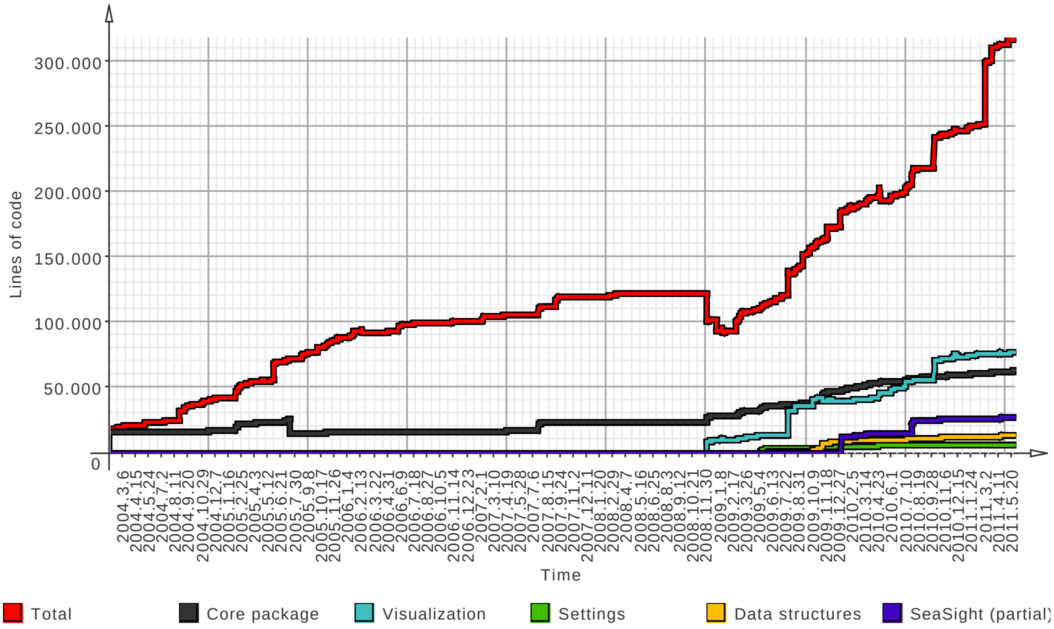


Figure 3.2.: Size of the MAYDAY project, given in lines of code (LoC) for the total package as well as several subsystems over the lifetime of the project. The figure is based on all packages included in the "experimental" release version, excluding unfinished code in the "incubator" repository. The curve for SEASIGHT only shows the main SEASIGHT components, excluding data structures, settings, and other components developed for SEASIGHT but placed into core packages. The decrease in total lines of code in November 2008 coincides with MAYDAY 2.5 and the switch from the old code repository to the new one.

3.2. Plugin Management

Prior to version 2.5, MAYDAY was built around a small core extended by plugins. The plugin management was kept simple such that all plugins extended MAYDAY at a single predefined point, namely the processing of `ProbeLists`. This was optimal for accommodating such methods as clustering or machine learning. Other extensions to the core functionality, especially those that did not work on a selection of `ProbeLists`, nevertheless had to be presented to the user in the same place. This resulted in a situation where plugins that had nothing to do with `ProbeList` processing could only be included at a semantically inappropriate place, at times confusing users as to why these plugins did not respect the current selection of `ProbeLists`. In addition, plugins extending MAYDAY at internal points where direct user interaction was not required, such as plugins implementing distance measures, were not possible in this system.

In the course of this work, MAYDAY's plugin system was rewritten from scratch. The new system incorporates three very important changes which will be described in more detail below:

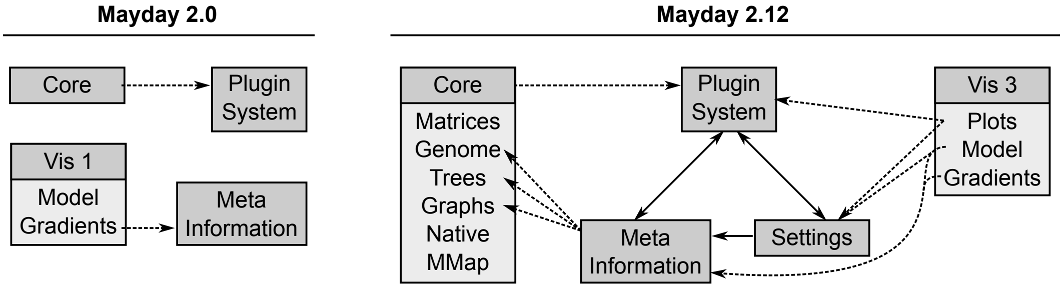


Figure 3.3.: Major elements of the Mayday core package and the changes in their integration between Mayday 2.0 and the current version 2.12. Arrows indicate dependencies. MMap, memory mapped data structures (see section 7.6.3); Vis1 (Mayday 1.0) and Vis3 (Mayday 2.12), visualization framework implementations (see 4.1).

- Unique plugin identifiers as part of each plugin's descriptor
- Multiple extension points
- The abstraction of file resources

3.2.1. Unique Plugin Identifiers

The new plugin management system defines a standard set of information required for each plugin, encapsulated in the `PluginInfo` class. It contains human-readable information regarding the plugin's author, the author's email address, a short description of the plugin's purpose and the name of the plugin. Further information required by the plugin manager are a unique plugin identifier and a list of dependencies.

The unique plugin identifiers are used throughout Mayday to refer to the plugin, to describe dependencies between plugins and for serialization purposes. When Mayday is started, the plugin manager begins by scanning for available plugins (Java classes extending the `AbstractPlugin` base class). Candidate classes are instantiated and queried for their plugin descriptor. In the second phase, plugin dependencies are resolved and all plugins are initialized in the order defined by their dependencies. From this point on, the unique plugin identifier can be used to refer to a certain plugin.

Plugin identifiers are also used for serialization purposes, equipping Mayday with a universal system for storing information about which plugin is capable of parsing certain data. Through this system, Mayday's file formats (most notably the `maydayz` snapshot format) can be used to embed additional information which is serialized and deserialized while the main file parsing and storing classes can defer finding parsers for embedded data items to the plugin manager.

3.2.2. Multiple extension points

As described before, the original Mayday version only allowed plugins to extend Mayday at one given point, i.e., the `ProbeList` context menu. By adding multiple points for extensions, the utility of the plugin system could be greatly improved and Mayday could be transformed into a more generic platform for data processing.

This was done by introducing a further information field into the plugin descriptor, the so-called *master component*. The plugin manager groups plugins by their master component and makes them available, e.g., for creating context menus. Many features of today's Mayday are only possible due to these changes.

As an example, all plugins working on `ProbeLists`, including most plugins already available in Mayday prior to the introduction of the new plugin system, specify `MC_PROBELIST` as their master component. Other extension points have been defined for plugins working on datasets and meta-information, as well as for internal plugins such as distance measures, statistical tests, data transformations and meta-information types, among others.

In terms of programming patterns, the plugin manager implements the factory pattern and, by the system of extension points, is turned into a plugin super-factory. Secondary factories, e.g., for distance measures, provide simplified access to some groups of plugins and/or adapt pre-existing structures in the original Mayday version to the new plugin management system.

3.2.3. Abstraction of file resources

As Java application, Mayday was designed to be run from a local installation or by using the Java WebStart mechanism which allows starting the program with a single click on a website link. A third deployment option is used by developers, who usually start Mayday from within their development environment (e.g., Eclipse). Each option results in different location of resources such as icons, help text files, filter specifications etc.

- In local installations, these files are usually part of `JAR` archives which, since they are part of the local file system, can be opened and scanned by the plugin manager. Additional plugins and resources may be available as individual files.
- During development, plugin classes as well as resources are available as individual files in a folder structure depending on the respective developer's choice.
- When using WebStart, classes and resources are hidden away in `JAR` containers which, though they are cached in the local file system, are not accessible to the plugin manager. (Prior to Java version 1.5, these resources were at well-defined locations in the user's home directory. Starting with Java 1.5 (Java 5 SE), the files are placed into Java's WebStart cache with names randomly assigned by the Java installation. In any case, the placement of these files is undocumented and subject to change without notice). Furthermore, users may have additional plugins installed locally that they want to have incorporated into their Mayday WebStart session.

Depending on the deployment option, the plugin manager chooses the appropriate methods to find available plugins and file resources. `JAR` files for webstart deployment contain an extra file as part of their meta-information which gives a list of all resources as well as a list of included plugins as part of the `JAR` manifest. The plugin manager enumerates available resources and makes them available to Mayday classes in the form of a virtual file system based on the `mayday/` prefix. Thus programmers can access icons etc. based on a simple file path irrespective of their actual position in the file system.

3.2.4. Surrogate plugins

With the addition of the `R` console RLink (see 5.2) and the JavaScript console (see 5.3) plugins, Mayday can make use of functions available in `R` or written in JavaScript to further extend its functionality. The flexibility of the new plugin system allows functions in these (and other) languages to be registered as Mayday plugins. Thus, a distance measure written as an `R` function and registered with the plugin manager will be available to all methods using distance measures, such as for example clustering methods.

These extensions are made possible via the `SurrogatePlugin` class which defines a Java adapter for non-Java plugin methods. The plugin manager recognizes these plugins and, instead of instantiating the plugin using a Java class constructor, uses a factory method of the surrogate plugin in conjunction with an extra object (e.g., the function name for `R` or the JavaScript source code). Plugins defined via this system can extend Mayday at any of the existing extension points.

3.3. Meta information

Expression analyses are usually performed in the context of *a priori* information about the genes under investigation. Such *meta information* or *annotation* can, for instance, assign genes to functional categories (such as Gene Ontology [5] terms) or metabolic pathways (such as KEGG [88] IDs). Meta information can also be derived from the *primary expression data*, an example would be the p-values obtained by computing a statistical test for differential expression on each gene's expression vector (given a class labelling of the experiments).

Meta information is an important part of Mayday's data model. Before Mayday 2.5, meta information was organized as collections of values (meta information objects, *MIO*s), grouped together in so-called `MIGroups`. Each group stored MIOs of a given type and associated `Probes` with MIOs. Different meta-information types were available (storing values of type `Integer`, `Double`, complex numbers, `String`, as well as `String` lists) and were statically included in all Mayday classes creating or using meta information. As a result, adding new meta information types required changing all classes that would potentially be able to work with the new type. The groups were stored in a flat structure, i.e., no relationships between `MIGroups` could be defined. Values stored in MIOs could not be visualized by the user, neither in a tabular view nor in specialized plots such as histograms. Furthermore, serialization was not part of the MIO definition, resulting in a situation where meta information could be imported into an opened dataset but could not be stored with that dataset. During the implementation of the "R interpreter" plugin, Matthias Zschunke implemented serialization support for the existing MIO types [186]. However, this was kept in

Table 3.1.: Differences between Mayday's old and new meta information framework

	old system	new system
Number of types	5	24
Pluggable	no	yes
Serialization	no	yes
Rendering	no	yes
MIGroup structure	flat	hierarchical
Used in visualizations	rarely	heavily

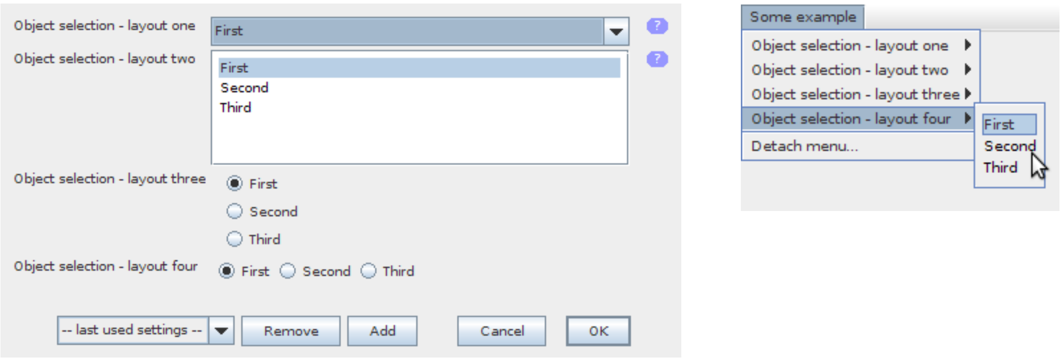
separate classes which were statically selected based on the type of information to serialize and, furthermore, were only used inside the R interpreter package.

The MIO system was completely rewritten for Mayday 2.5 (changes are listed in table 3.1). Firstly, all MIO types were implemented as Mayday plugins implementing the `MIType` interface (or extending the abstract `GenericMIO` class). Each MIO type needs to implement its own serialization and deserialization methods, creating a plain-text representation as well as an XML representation. These are used to persistently store all meta information associated with a given Mayday dataset in the so-called "Mayday Snapshot format", a compressed, fast-access file format which was also introduced with Mayday 2.5. A further interface `HugeMIO` was introduced with Mayday 2.11 for meta information types storing large amounts of data. These implement efficient serialization to Java `Input`/`OutputStream`s.

Furthermore, each `MIType` can implement special *rendering classes* for display of its content in Mayday's overview dialogs as well as *editor classes* that can be used to change the value of the object (if it is not read-only). Many of Mayday's visualizations (see also 4.1) can now use meta information either directly (e.g., displaying the distribution of numeric meta information in a histogram), or as a source of *relevance* information (e.g., influencing opacity to highlight relevant profiles in a profile plot), as the basis for coloring visual elements (based on numerical values mapped to a color gradient or an assignment of distinct colors to categorical meta information), as well as for sorting rows in tabular representations or Mayday's heatmap plot, for instance.

MIO types can implement a set of interfaces to indicate the kind of data they contain, such as numerical data (`NumericMIO`), categorical data (`CategoricalMIO`), or data that can be ordered (`ComparableMIO`). This allows other classes in Mayday to decide which `MIGroup`s provide valid input. Thus, adding new meta information types only rarely requires changes to existing code, and only in cases where specific handling of the new type is needed, e.g., a new numerical type with higher precision would not require any changes to visualizations using numeric meta information to assign colors.

The meta information groups relating MIOs to other objects were rewritten to provide a much larger set of access functions for programmers as well as to expand the applicability of meta information beyond only annotating `Probe` objects. Other



Object selection - layout one: First

Object selection - layout two: First

Object selection - layout three: ● First ○ Second ○ Third

Object selection - layout four: ● First ○ Second ○ Third

Second
Third
-- last used settings --
Remove
Add
Cancel
OK
Some example
Object selection - layout one
Object selection - layout two
Object selection - layout three
Object selection - layout four
Detach menu...
First
Second
Third

```
HierarchicalSetting mySetting = new HierarchicalSetting("Some example", LayoutStyle.PANEL_VERTICAL, true)
.addSetting(
        new ObjectSelectionSetting(
                "Object selection - layout one",
                "Help text ",
                0,
                new String[]{"First","Second","Third"}
        )
        .setLayoutStyle(ObjectSelectionSetting.LayoutStyle.COMBOBOX))
                    ⋮
```

Figure 3.4.: Graphical User Interface representations shown for the example of a `ObjectSelectionSetting`. Top left, different styles can be used for displaying the object selection in dialog windows. The automatically created user interface also contains control elements to store and load often-used configurations. Top right, the same selections as represented when embedded in a menu. Bottom, exemplary source code for creating one of the settings and adding it to the surrounding hierarchical setting.

objects, such as `ProbeLists`, `DataSets` and even other `MIType` objects can now be annotated. A hierarchical structure was introduced, assigning a *path* to each `MIGroup` similar to a file system, to allow the definition of relationships between `MIGroups`. This makes it easier to find `MIGroups` in strongly-annotated datasets, and also reflects logical relationships. For instance, a statistical test may produce a test statistic, a raw p-value as well as a p-value corrected for multiple testing. One might choose to have the corrected p-value at the root level of the `MIGroup` hierarchy, as it is arguably the most informative of the three values, while the raw p-value as well as the original test statistic could be added as children in the `MIGroup` tree.

3.4. Persistent Settings

3.4.1. Motivation and requirements

Most algorithms and analysis methods have some configurable parameters. As a consequence, programmers implementing such methods for MAYDAY have often spent considerable effort in creating user interfaces to allow users to change these parameters. Often, these user interfaces were very different from method to method, and many ‘lessons’ learnt during the development of previous methods were not known to those implementing new methods, or were not heeded due to time constraints.

However, there are some very desirable features that are almost universally useful. These are:

1. A **consistent visual appearance** of configuration dialogs and GUI elements. For example, if a color has to be selected, the GUI element for this purpose should look the same in all dialogs. In addition, if the same component is re-used, improvements need only be implemented once and automatically affect all instances of the component.
2. Context-sensitive **help texts** for different options, especially when their impact on the method's output is not obvious.
3. Automated **checking of parameter values** including appropriate notifications to the user in case a parameter was not specified correctly. For example, programmers should be able to define the acceptable value range for a numerical parameter. The implementation should check user input before accepting it and present a helpful message in case the input does not fulfill the requirements. Implementors using such parameter implementations will not have to spend any time validating the input, as they are guaranteed that any input is already tested.
4. **Serialization** of parameters for later re-use, not only during one analysis session but also in a persistent manner. Specifically, being able to store different sets of parameter combinations for later use can greatly simplify analysts' work, in addition to preventing errors in repetitive analysis tasks.
5. An **event-based system** to communicate parameter changes, e.g., to automatically update a plot when a visual parameter is changed.
6. **Integration** of parameters into different GUI elements such as dialog windows and menu bars with appropriate visual representation and event handling, as well as synchronization of multiple *views* visualizing the same parameter instance.
7. The possibility to **programmatically access** the values of parameters, also from outside the original implementor's code. This can be useful in the context of automating analyses, where manipulating the content of a GUI window would be very cumbersome. A more elegant solution is to directly configure the parameters via a common interface and then run the method with these parameters.
8. A **hierarchical representation** of parameters that allows for the aggregation of several methods' parameters into one parameter object. Thus, methods can be combined (for an example, see section 4.4.1) and the parameters for the combination are made up of the parameters for each of its parts.

3.4.2. Implementation

All the above requirements are all fulfilled by Mayday's `Settings` framework which was first implemented as part of SeaSight. Due to its usefulness, the framework was incorporated into and tightly integrated with the Mayday core and is now used by many other packages (e.g., the complete visualization framework) and plugins.

Table 3.2.: `Setting` types implemented in MAYDAY's core package (selection).

Base classes	
Setting (interface)	defines the common interface for all settings
AbstractSetting (abstract)	implements common functionality
GenericSetting	delegates storage and serialization to a `MIType`
Single values	
Setting classes are implemented for the types `boolean`, `double`, `integer`, `long`, `String`, for multiline `String`s, directories and files, colors and meta-information.	
Lists of values	
Setting classes are implemented for ordered lists of the types `double`, `integer`, `String`, for lists of files, as well as for a list of `Objects` based on a predefined set.	
Miscellaneous	
StringMapSetting	a set of pairs of `String`s, e.g., a mapping of names
ObjectOrderSetting	an ordering on a set of predefined objects
ClassSelectionSetting	a class labelling on a predefined set of objects
Object selections	
ObjectSelectionSetting<T>	one object from a predefined set
MultiselectObjectListSetting<T>	a selected subset from a predefined set
RestrictedStringSetting	one `String` from a set of predefined `Strings`
SuggestedStringSetting	one `String` from a set of predefined `Strings` or a `String` entered by the user
Plugin-related settings	
specific classes are implemented for certain plugin categories: distance measures, data transformations, *p*-value correction, statistical tests.	
PluginInstanceSetting<T>	a plugin instance selected from a list of available plugin instances, each potentially containing its own setting objects.
PluginInstanceListSetting<T>	a list of plugin instances that can be constructed based on a predefined set of available plugin instances, each potentially containing its own setting objects.
PluginMultiselectListSetting<T>	a selection of plugins from a predefined set of available plugins, each potentially containing its own setting objects.
Hierarchical settings	
HierarchicalSetting	an ordered list of settings (each possibly with subsettings)
BooleanHierarchicalSetting	encapsulates any kind of setting and a `boolean` value; if set to true, the sub-setting can be configured.
SelectableHierarchicalSetting	an object selected from an ordered list of objects, where each object can be itself a Setting
SortedExtendableConfigurable-ObjectListSetting<T>	An orderable list of objects composed from a set of available objects (dynamically constructed based on the current list), where each object can supply its own Settings.

All types implement the `Setting` interface and with the exception of `ComponentPlaceHolderSetting`, which does not store a value but can be used to integrate any SWING component in auto-generated user interfaces, derive from the `AbstractSetting` base class via `GenericSetting`.

The implementation is split into the storage, event-handling and validation part defined in the `Setting` interface and implemented largely in the `AbstractSetting` and `GenericSetting` classes (see table 3.2 for an overview of the core setting types), and into the user interface part defined in the `SettingComponent` interface and implemented largely in the `AbstractSettingComponent` class. Some settings offer different visual styles to allow programmers to influence how the setting is represented in user interface components (an example is shown in figure 3.4).

The same `Setting` can be represented by more than one GUI component. As an example take a `BooleanSetting` (storing one binary value) which can be included in a menu bar (using `JCheckBoxMenuItem`, a menu item with a check box next to its label) as well as in a dialog window (using an ordinary `JCheckBox`). By clicking on the menu item, users can toggle the value (true/false) of the underlying `Setting` which will be instantly reflected in the state of the `JCheckBoxMenuItem` as well as in that of the `JCheckBox` shown in the dialog window.

An important aspect of the Settings framework is its tight integration with Mayday's plugin system: Plugins can define their parameters based on the settings framework. If a method requires selection of a plugin, e.g., if a clustering method requires selection of a distance measure, the parameters of the selected distance measure plugin will automatically become part of the clustering method's parameter set, both for user interface creation as well as for serialization.

3.4.3. Benefits

Using the `Settings` framework, time that would have been spent on writing user interfaces, parameter validation and serialization (which was often omitted before) can now be spent developing and optimizing and, above all, testing the actual method. Furthermore, the definition even of very complex parameter sets no longer results in several hundred lines of code spread over several classes (GUI elements, validation, storage) with sometimes indistinct responsibilities but can be accomplished in a few lines (typically one single line for each parameter).

The serialization capabilities included in each setting type can be used to apply identical parameter values to a large number of objects. This is used in SeaSight to allow users to change the parameters of several selected experiments: First, a user interface is created showing the setting components for the settings common to all experiments. After they have been configured by the user, the new values are serialized and deserialized into the individual setting objects of the experiments. This in turn triggers change events within each experiment's setting which result in updates to the experiment's state as well as to the main SeaSight user interface.

And finally, user interface components (dialogs, menus) are automatically created for the parameters with intelligent and consistent layouting of components. No external dependencies or pre-processing steps are required.

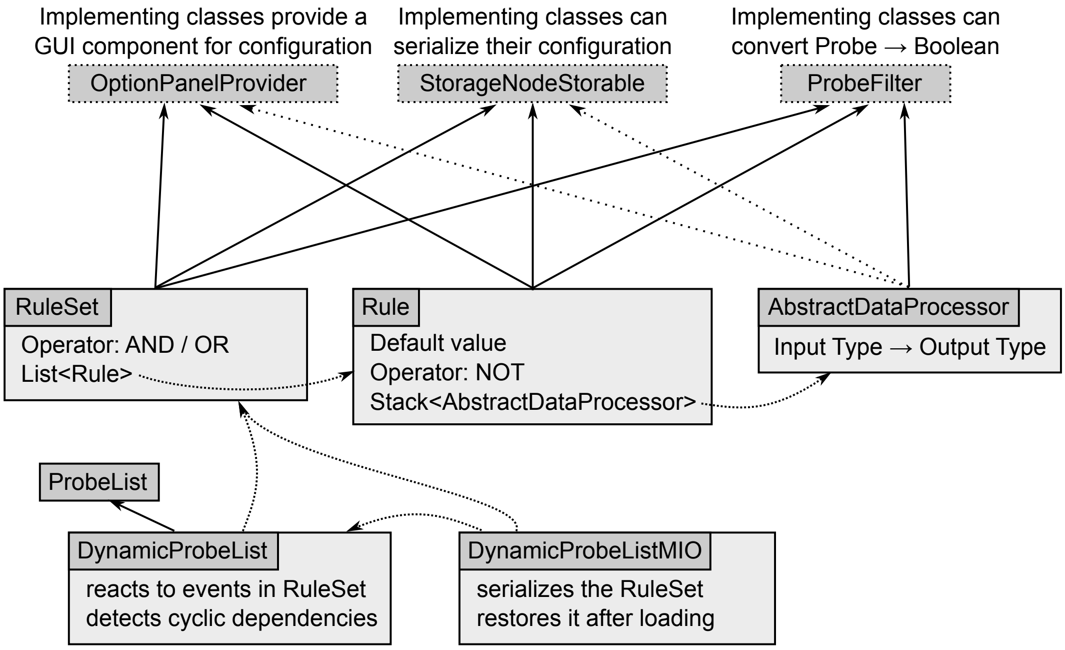

Implementing classes provide a GUI component for configuration
OptionPanelProvider
Implementing classes can serialize their configuration
StorageNodeStorable
Implementing classes can convert Probe → Boolean
ProbeFilter
RuleSet
Operator: AND / OR
List<Rule>
Rule
Default value
Operator: NOT
Stack<AbstractDataProcessor>
AbstractDataProcessor
Input Type → Output Type
ProbeList
DynamicProbeList
reacts to events in RuleSet
detects cyclic dependencies
DynamicProbeListMIO
serializes the RuleSet
restores it after loading

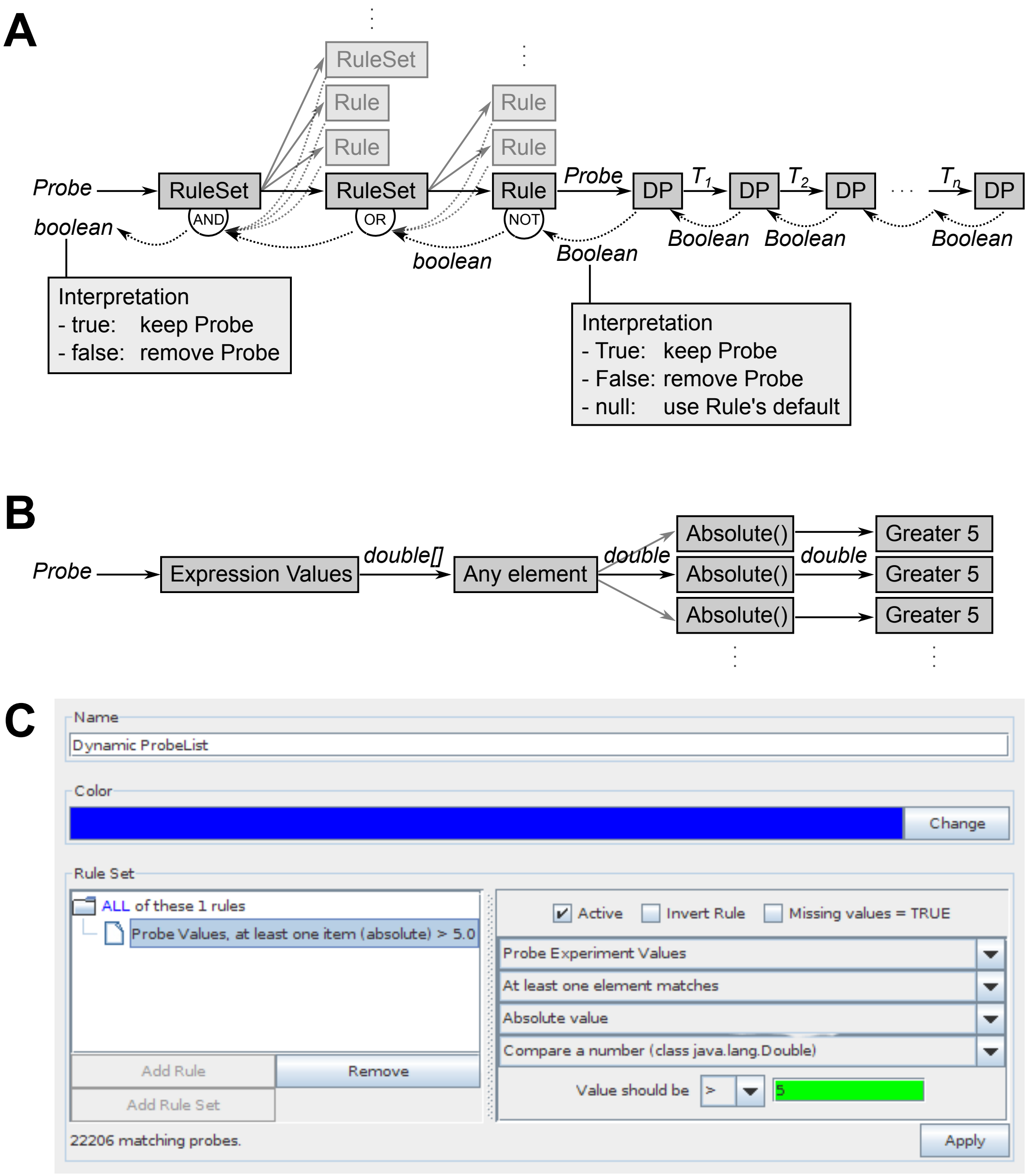
Figure 3.5.: Important classes in the dynamic `ProbeList` framework. Classes are shown with a solid border, interfaces with a dotted border. Solid arrows indicate interfaces that are always implemented, sparsely dotted arrows indicate that some of the respective classes implement a given interface. Finely dotted arrows indicate dependencies. The `ProbeList` class is part of Mayday's core.

3.5. Dynamic interactive filtering

Filtering plays one of the most important roles in the analysis of high-throughput datasets. The number of features studied is usually in the range between a few thousand and several hundred thousand, which makes visual inspection of the whole dataset infeasible. Since most features (i.e., genes) are expected to not be differentially expressed between different conditions, hiding uninteresting features allows to reduce visual as well as analytical complexity without sacrificing relevant information.

Usually, an analysis contains several rounds of filtering, visual inspection, statistical computations and hypothesis generation by the researcher. In Mayday, this is supported by a very flexible filtering framework, implemented under the name *Dynamic ProbeLists* (DPL, see figure 3.5 for an overview of the elements involved). The basic idea is to use filter criteria to define which probes (i.e., features, genes) to include in the resulting `ProbeList`, and which ones to exclude. Two elements of the DPL system are fundamental for its flexibility and power:

- **Small building blocks:** Similar to the GNU suite of tools, dynamic `ProbeList` filters are constructed from very simple parts, so-called `DataProcessors` which can be chained together (as shown in figure 3.6A). A `DataProcessor` converts



Name
Dynamic ProbeList

Color
Change

Rule Set
ALL of these 1 rules
Probe Values, at least one item (absolute) > 5.0

☑ Active ☐ Invert Rule ☐ Missing values = TRUE

Probe Experiment Values
At least one element matches
Absolute value
Compare a number (class java.lang.Double)
Value should be > 5

Add Rule | Remove
Add Rule Set

22206 matching probes.

Apply

Figure 3.6.: Processing path and user interface of a dynamic `ProbeList`. **A**, `RuleSets` aggregate the results from several independent tests either by an AND or OR operation. `Rules` encapsulate one test, providing a default value and, optionally, inverting the result (NOT operation). `DataProcessors` (DP) convert an input object into an output object, passing it on to the next DP, until the result is of type `Boolean`. This result is passed back the chain to allow further modification and aggregation operations. **B**, the data processor chain expressing the test *"at least one expression value should be larger than 5 or smaller than -5"*. The "All elements" processor uses an array as input (regardless of element type) and executes the remaining chain for each element in the array, aggregating the result. **C**, Dialog window for setting up a dynamic `ProbeList`, showing the setup of the same data processor chain (right panel), the automatically created name for the rule (left panel), as well as the number of `Probes` passing the filter (bottom left).

one type of data into another type and hands the result to the next element in the processing chain. A chain is *complete*, *iff* after passing through all processors, the input data (of type `Probe`) has been converted to a `Boolean` value, indicating whether to include or exclude said probe from the dynamic `ProbeList`. Each chain of `DataProcessors` is encapsulated by a `Rule` object, which takes care of cases where the chain can not come to a positive or negative result. In such cases, a (user-defined) default decision is made. Furthermore, the `Rule` can apply the unary Not operator. To build more complex filters, several criteria (chains) can be combined in a tree structure of `RuleSet` objects, each of which can be configured to either apply the And or the Or operation.

Using such small building blocks allows for very flexible specification of quite complex filters while keeping the implementation effort for each `DataProcessor` low. As an example, consider the criterion "*at least one of the probe's expression values should be larger than 5 or smaller than -5*". This very specific condition can be modelled using a chain of four `DataProcessors`. The first one converts a `Probe` object into an array of `double` values by extracting the `Probe`'s expression values. The second one applies the "any element of a collection matches" operation, made possible by the bidirectional data flow in the processing chain (as shown in figure 3.6B): Each `DataProcessor` converts its input value and hands the output value to the next `DataProcessor`. The final result of type `Boolean` is handed back in the reverse direction, allowing each `DataProcessor` to apply further computations before passing the result on. In this example, the "Any element" `DataProcessor` executes the remaining chain independently for each element in its input object and aggregates the result using the Or operation (taking care of correctly handling `null` values). The third `DataProcessor` in our example merely computes the absolute value of a number. Finally, the fourth and last element checks whether its (numerical) input fulfills a given equality resp. inequality with respect to a predefined number.

- **Dynamic updates:** A dynamic `ProbeList`'s `RuleSet` may contain criteria that depend on other `ProbeLists`, even other dynamic `ProbeLists`. If any of those lists changes, the dynamic `ProbeList` is automatically updated to reflect changes resulting from the change in the other list(s). The same is true for criteria depending on meta-information values, visualizer selections and so forth. The algorithm taking care of the updates can detect circular dependencies. In such cases, updates are stopped, and the user is informed about the problem. As soon as the circle has been removed by the user, updates are resumed.

  Furthermore, all updates to `ProbeLists`' content (regardless of whether they are dynamic or not) are automatically communicated to Mayday's visualizations (see section 4.1), providing real-time updates to all plots based on those `ProbeLists`. As a result, researchers can quickly change filtering criteria in one window, while observing the resulting changes in any number of linked visualizations.

Users can construct the `RuleSet` for a dynamic `ProbeList` using an intuitive GUI where the tree of AND and OR aggregations (`RuleSet`s) can be constructed (see figure 3.6C). For each `Rule`, the processing chain can be built by successively selecting `DataProcessor`s and adding them to the chain. MAYDAY automatically determines which `DataProcessor`s are applicable, based on the output type of the current last element in the chain. Input fields for configurable `DataProcessor`s (which implement the `OptionPanelProvider` interface) are also shown. During the construction of the `RuleSet`, the number of `Probe`s matching the dynamic `ProbeList`'s criteria is shown in the GUI and updated in real time. All `Rule`s that contain a complete processing chain, i.e., which can be applied to a `Probe` object to yield a `boolean` result are considered for this determination. A description of the complete `RuleSet` is automatically created. For the example described above, this descriptive text would be "*Probe values, at least one item (absolute) > 5*".

An interesting aspect of the dynamic `ProbeList` framework is that it is completely implemented outside of MAYDAY's core. From the core's point of view, dynamic `ProbeList`s are no different from ordinary `ProbeList`s. Configuration dialogs, processing modules, serialization and deserialization are all implemented based on MAYDAY's plugin and meta-information systems. This implementation approach relies on the use of different listeners to restore the 'dynamic' nature of the dynamic `ProbeList`s after loading a dataset containing such `ProbeList`s into MAYDAY, since they are stored as ordinary `ProbeList`s annotated with a meta-information object of a special type (`DynamicProbeListMIO`). After the dataset has finished loading, it is added to MAYDAY's `DataSetManager`, triggering an event. This event is received by the deserialized meta-information object, which creates the necessary `RuleSet` structure and replaces the ordinary `ProbeList` in the `DataSet`'s `ProbeListManager` with a dynamic `ProbeList` based on that `RuleSet`.

3.6. Task Management

Some calculations on expression data can be very time-consuming, such as hierarchical clustering of a large number of probes or samples. Originally, all operations in MAYDAY were implemented as *blocking* operations, i.e., after starting a computation, MAYDAY's user interface would become unresponsive until the respective task was finished. Most methods did not give any indication of their progress, leaving users to wonder how long they would have to wait before being able to continue their analyses. Some methods showed some sort of progress information in a method-specific user interface, some of them even allowed users to cancel the current task. However, pressing the cancel button often just dismissed the progress dialog (thus unblocking the user interface), and after the computation was completed, the result that was finally obtained was discarded.

This situation clearly was not satisfactory. In some cases, users might need to wait for one computation to finish before starting the next step because of a logical connection between these steps. However, analyses often involve several computations that can be performed in parallel. For instance, users might want to compute a hierarchical clustering using different distance measures. On modern multi-core computers, these different computations can be performed in parallel while the user can continue investigating the data.

Mayday 2.5 introduced a task management system that solves the problems of the original system while placing no additional implementation burden on the authors of Mayday plugins. The task manager automatically creates new threads when plugins are invoked on the data and provides a common user interface that shows the progress of all running threads (if plugin authors chose to report fractional progress state), the estimated remaining time, and messages produced during the computation, as well as a button to cancel any thread. Tasks can contain subtasks whose fractional progress contributes to the parent task's overall progress. Many Mayday plugins now implement their own cancelling behaviour, stopping computations, cleaning up data structures, etc. To allow cancelling of plugins that do not react to users' cancel requests, the task manager can forcibly kill any thread after a certain time has passed without the thread acknowledging the cancel request.

The introduction of concurrent computations mandated many small changes in Mayday's core data structures either to enforce synchronization or to safely allow unsynchronized concurrent access. Mayday now uses an *optimistic* approach to concurrency, which works in most cases. There are some situations in which concurrent access can fail, e.g., when one plugin fundamentally changes the structure of a dataset (for instance removing a column from the expression matrix) while another accesses the same dataset (for instance relying on the number of columns to be fixed during the whole computation), but these are extremely rare in practice.

In such a situation, one of the two concurrent tasks will fail and produce an error message, requiring the user to manually restart that task. The rarity of such situations is due to the fact that only very few plugins actually change the structure of a dataset, and also due to users' understanding that performing computations on a dataset that is in the process of being fundamentally reorganized not only might fail but will also most likely produce results that are of no interest in the context of the reorganized dataset.

3.7. **Application:** Expression profiling of the metabolic switch in *Streptomyces coelicolor*

Coinciding with the development of many of the new features described in this chapter, a large transcriptomics dataset studying the metabolic switch in *Streptomyces coelicolor* was analysed using Mayday [111]. In this study, the soil bacterium *S. coelicolor* was grown under controlled conditions in liquid medium fermenters with automatic adjustment of oxygen levels leading to highly reproducible data [11]. This organism undergoes a metabolic switch from primary metabolism (exponential growth) to secondary metabolism (stationary phase). The secondary metabolism, among other things, produces antibiotics, making *S. coelicolor* an interesting target for biotechnical engineering.

Five different strains (wild type and four mutant strains lacking the *phoP*, *scbR*, *glnR* and *glnK* genes, respectively) were cultured under different conditions (phosphate

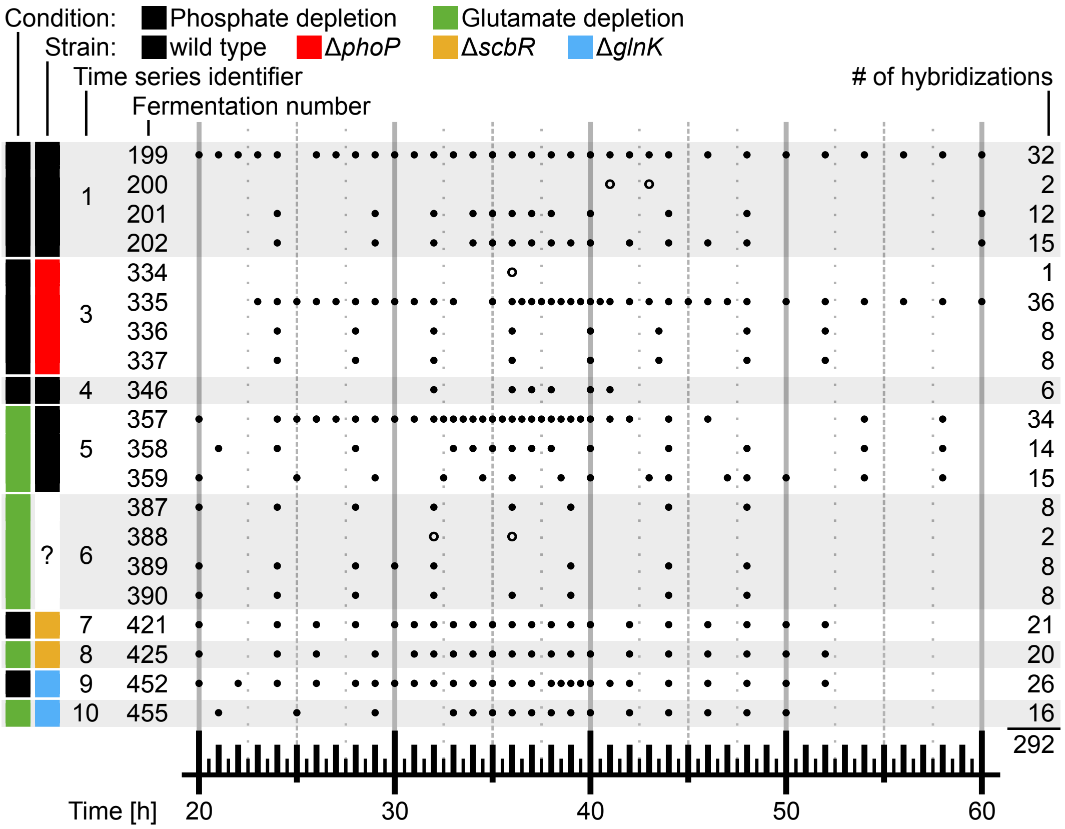


Figure 3.7.: Microarray hybridizations for all time-series datasets produced in the course of the SysMO Stream project. Filled circles indicate transcriptomics samples used for expression analysis, empty circles indicate genomic samples processed for mutant verification. Only samples selected for processing and hybridisation to microarrays are included in the table. Time-series 2 was cancelled due to a problem during fermentation, time-series 6 was not analyzed further due to a problem with the selected mutant strain (see text). Only 292 samples hybridized to arrays are shown; for all fermentations, samples were taken at regular (hourly, half-hourly) intervals. Missing fermentation numbers refer to preliminary trials for establishing media and mutants.

depletion and glutamate depletion) and samples were taken at different time-points between 20h and 60h after inoculation to cover the metabolic switch (see figure 3.7 for the detailed sampling scheme). Samples were used for transcriptomic analyses as well as for metabolomic and proteomic studies (performed in other groups).

Analyses were mostly performed by Kay Nieselt and Alexander Herbig, Mayday was used for all analysis steps. The clustering, dynamic filtering and visualization methods were used most often. Other tools, such as Mayday's genome browser ChromeTracks (see 4.3) were used for specific questions, especially for mutant verification. In the following, important steps in the analysis are highlighted. These can be considered relevant elements of any standard pipeline for projects of this type.

3.7.1. Data acquisition and normalization

For the transcriptomic analyses, RNA was extracted from the samples and, after the appropriate processing steps, hybridized to Affymetrix GeneChips® Custom-Express™ arrays containing 226,576 perfect match probes which interrogate coding sequences (8,205 probesets), intergenic regions (10,834 probesets) and predicted noncoding RNAs (3,671 probesets) [11]. Affymetrix CEL files were imported and normalized with the robust multiarray average (RMA) method [77] using Mayday SeaSight. Array identifiers were mapped to time-points by attaching meta-information to the experiments.

3.7.2. Mutation verification

Besides the *Streptomyces coelicolor* M145 wild type strain, several mutant strains were used to elucidate the role of key regulators in the context of metabolic switching induced by nutrient (phosphate resp. nitrogen) starvation. The Δ*phoP*, Δ*glnK*, Δ*glnR* strains were deletion mutants, the Δ*scbR* mutant produces a functionally inactive scbR protein. To confirm the gene deletions, genomic DNA was hybridized to the *S. coelicolor* microarrays and, instead of summarizing probe-level expression values into probeset expression values using the RMA algorithm, the probe-level data was visualized directly in Mayday ChromeTracks together with the genomic loci of deleted genes.

In the case of the Δ*phoP* mutant strain, the complete deletion of *phoP* (SCO4230) can clearly be verified (figure 3.8). In the Δ*glnR* mutant strain, *glnR* (SCO4159) was partially replaced by a gentamicin resistance cassette [50]. The fact that all probes designed for *glnR* show relatively high 'hybridization values' (which in this experiment indicate the presence of the respective sequences in the *genomic* DNA) indicates that the selected strain does not contain the desired mutation (figure 3.9). As a result, further processing of time-series 6 was abandoned.

3.7.3. Time-point clustering

The expectation in a time-series experiment is that expression changes between adjacent time-points are less marked than if comparing more distant time-points. The two most highly resolved time-series datasets (TS1, fermentation 199 and TS5,

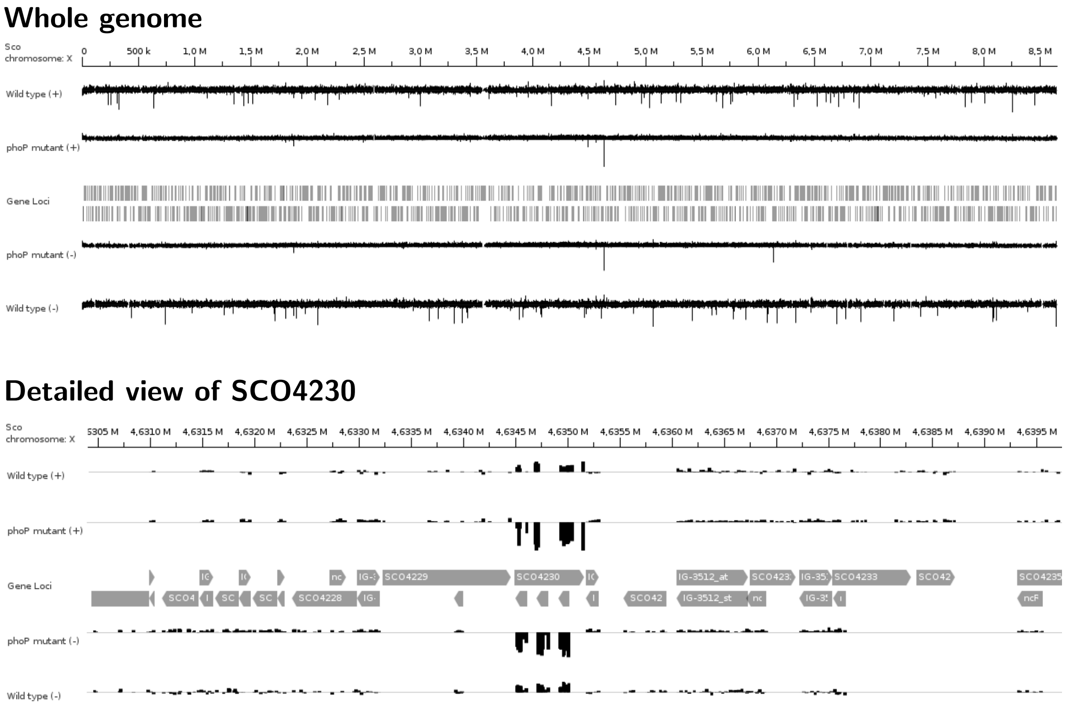


Figure 3.8.: Probe-level data aligned to the *S. coelicolor* genome. Outer tracks: wild-type, Inner tracks: Δ*phoP* mutant. Data was centered for improved visual clarity (the mean of wild-type and mutant hybridization value was subtracted from both). In the whole-genome view, the deletion at 4.6M is already apparent (top). Zooming in to show *phoP* (SCO4230) clearly shows that the whole gene is deleted, as probe-level hybridization values on both strands are significantly lower for the mutant strain.

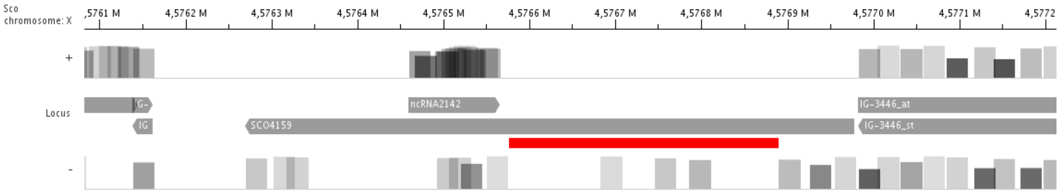


Figure 3.9.: Probe-level data aligned to the *S. coelicolor* genome. The deleted segment in *glnR* (red; leaving 97bp of the 5' region and 298bp of the 3' region intact) is covered by 3 probes with consistently high values (> 10) indicating that the strain used here was not the desired mutant strain lacking *glnR*.

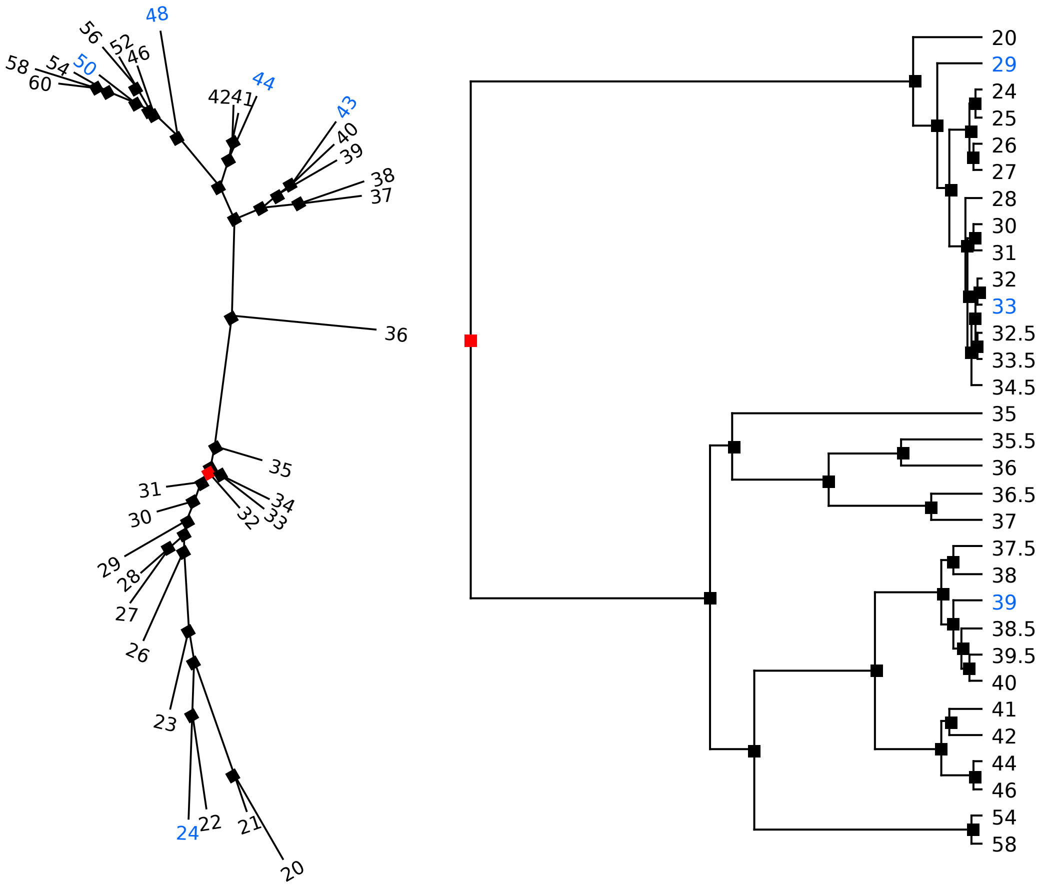


Figure 3.10.: Results obtained by hierarchical clustering of the samples in fermentation 199 (time-series 1, left) using the neighbour-joining method shown as an unrooted tree, and fermentation 357 (time-series 5, right) using the UPGMA method shown as dendrogram. Both clusterings were based on euclidean distances computed on the most variant protein-coding genes (regularized variance > 0.1). Only few experiments are not clustered perfectly according to their time-point label (highlighted in blue). The longest edges coincide with the time of nutrient depletion (after 36h in F199, after 34.5h in F335) indicating a major transcriptional event as response to changing nutrient availability.

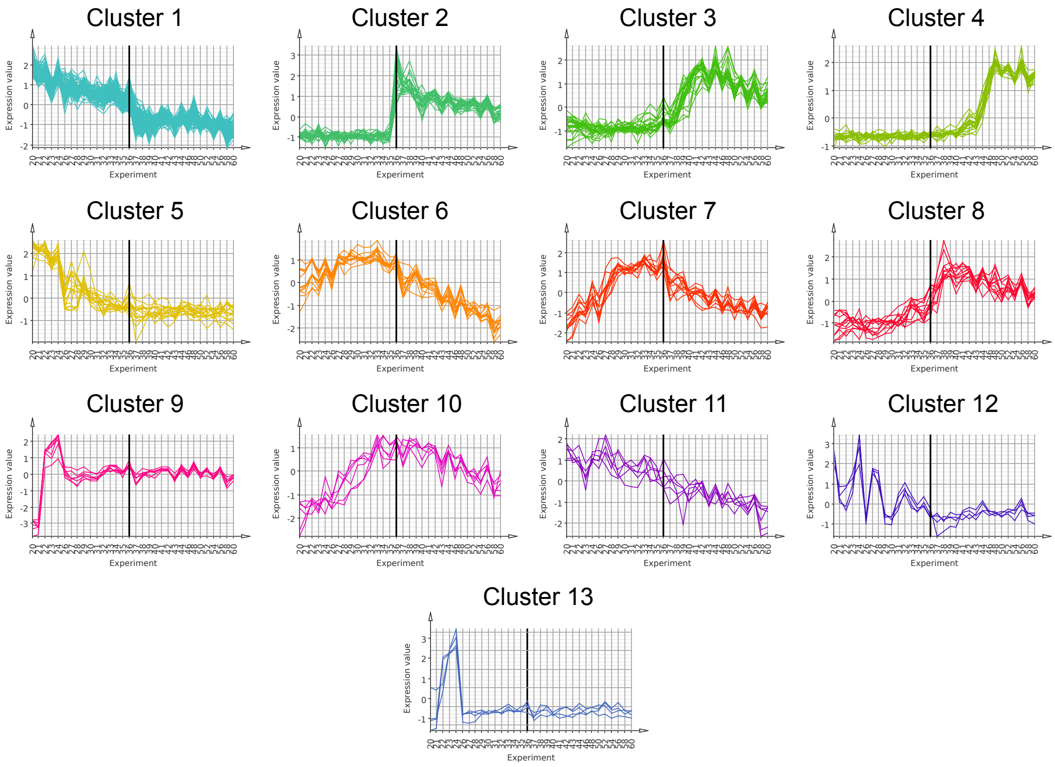


Figure 3.11.: Profile plots of clusters created by QT clustering (Pearson correlation, diameter 0.25, minimal cluster size 4) of the 322 most variant protein-coding genes (regularized variance > 0.1) of *Streptomyces coelicolor* wild type under phosphate-limiting conditions (time-series 1, fermentation number 199). The time-point of phosphate depletion is indicated in each plot by a strong vertical line.

fermentation 357) were selected to test this hypothesis. After importing the data and a mapping of Affymetrix probeset identifiers to *S. coelicolor* gene identifiers, dynamic filtering was used to select all probesets representing protein-coding genes with a regularized variance (variance divided by mean) larger than 0.1. These 322 (F199) resp. 223 (F357) genes were used as the basis for hierarchical clustering using the euclidean distance and the rapid neighbour-joining method [146] for F199 and the UPGMA method for F357.

The resulting clustering trees (shown in figure 3.10) nicely confirm the expectation. In addition, the longest edge in each tree is the edge separating the time-points *before* nutrient depletion and those *after* nutrient starvation. The few experiments that are not perfectly clustered according to the ordering defined by their time-point annotation can be explained by the fact that during the early, resp. very late periods, transcription does not change much. More importantly, no experiment is clustered on the 'wrong side' of the divide defined by the nutrient depletion event.

3.7.4. Gene clustering

The probesets representing highly variant protein-coding genes were also the basis for a clustering of the expression matrix rows (i.e., the genes). The QT (quality threshold) clustering method [69] as implemented in Mayday by Günter Jäger [78] was used with a minimal cluster size of four and the diameter set to 0.25 to cluster these genes according to the Pearson correlation [55, 117] distances of their expression profiles. 13 clusters were created (shown in figure 3.11).

Mapping these clusters to the *S. coelicolor* genome reveals that some of these genes with similar expression profiles are also clustered in terms of their genomic loci (figure 3.12), e.g., the genes in cluster 4 which are responsible for the production of the antibiotic actinorhodin. Cluster 3 is a combination of three chromosomal clusters with very similar expression. Interactively selecting these three clusters in the genome browser and using Mayday's *tag cloud* visualization (not shown) to summarize the functional annotations (Sanger protein classifications [136]) of these genes shows that one of the genomic clusters is responsible for the production of the antibiotic undecylprodigiosine, the second is comprised of proteins associated with membrane production, and the third is only generically annotated as related to secondary metabolism.

The genes of the largest cluster (cluster 1, comprising 147 protein-coding genes) are spread over the whole chromosome, indicating that the strong downregulation that coincides with phosphate depletion has a global effect on transcription in *S. coelicolor*.

3.7.5. Outlook: Systems Biology

The Systems Biology approach at the heart of the SysMO Stream project involved not only the analysis of transcriptome samples, but also of proteome and metabolome samples. As these data are also represented by a numeric matrix (relating proteins with their abundance, resp. metabolites with their concentration), Mayday can be used to analyze them just like transcriptomics datasets.

The time-series alignment tool Tiala (see section 4.4.2) was developed to enable visual comparisons between the expression profiles of clusters of genes under different conditions in different mutants. It was used extensively during the analysis of the SysMO Stream dataset for this purpose. Tiala is flexible enough to also allow for comparisons between datasets of different 'omics' experiments. In parallel to the transcriptomics data created for time-series 1 (fermentation 199), proteomics data was produced from the same samples by the group of Margaret Smith in Aberdeen [159].

The quantitative proteomics dataset used here covers 780 proteins in 8 experiments (time-points 20, 29, 32, 35, 41, 48, 54, and 60h after inoculation). Data points were averaged over two runs and imported into Mayday. The expression clusters found by the QT algorithm were used to create Tiala plots of aligned transcript and protein abundance profiles (figure 3.13). The proteomics and transcriptomics data agree very nicely. As expected, increases in the protein abundances (spectral counts) lag behind the transcriptional activation of the respective genes (e.g., visible in clusters 2

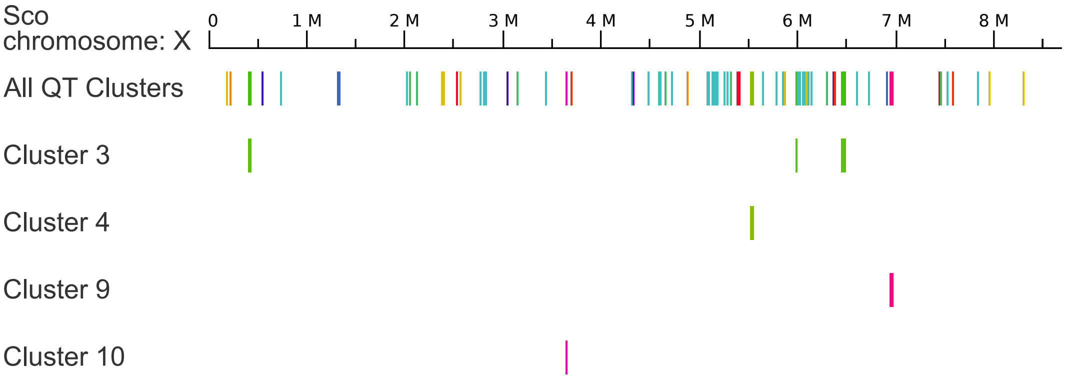


Figure 3.12.: QT clusters in Mayday's genome browser plotted according to the genomic positions of the genes contained. Expression clusters 4, 5 and 10 are single genomic clusters, expression cluster 3 is spread over three genomic clusters. Colors correspond to those in figure 3.11.

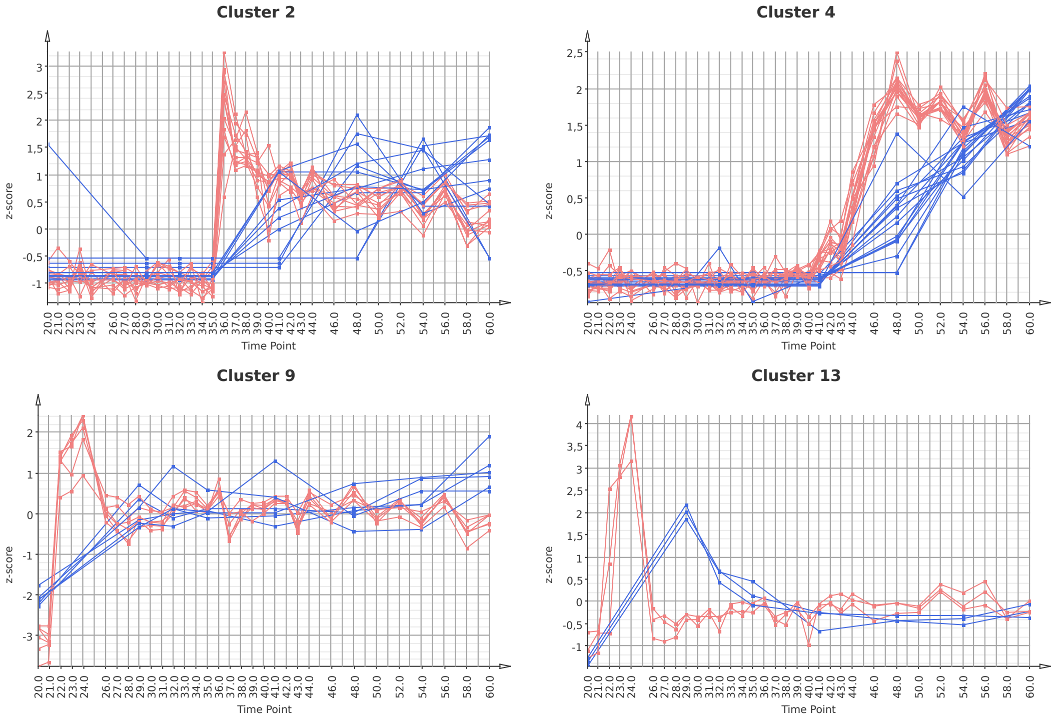


Figure 3.13.: Aligned profiles of transcription strength (red) and protein abundance (blue; measured as spectral units). Data was z-score normalized for comparison purposes. Protein abundances were only measured for some time-points, indicated by filled blue squares. Increases in protein levels lag behind transcription activation (clusters 2, 4), sustained transcription results in stable protein levels (cluster 9), while transient transcription is reflected in a short peak in protein levels (cluster 13).

and 4). The transient transcriptional responses also result in transient increases in protein levels (cluster 13), while sustained transcription leads to constant protein abundance (cluster 9).

The next step in a Systems Biology investigation could now be the inclusion of metabolite data and metabolic pathway information to gain insight into the complex interplay of transcription, protein abundances and activities, and metabolic fluxes. Combined analyses of this kind based on pathway information and graph visualization was a major focus of Stephan Symons' dissertation [154], part of which resulted in the development of the Mayday Graph Viewer [155].

4. Visual analytics in Mayday

4.1. Vis3 – Mayday's new visualization framework

Visualizations are the most important aspect of MAYDAY. They are immensely useful (if not essential) to understanding complex datasets (for some examples see [156, 111], as well as section 3.7). Consequently, the visualization subsystem has seen several iterations of development, starting with three individual plots (box plot, profile plot, enhanced heatmap) loosely linked by a simple data model. The second iteration was Philipp Bruns' [32] introduction of new and replacement of existing plots (scatter plot, profile plot) based on a scatter plot visualization library, using a different data model. The necessary integration of old and new plots lead to the development of the third visualization data model in combination with a very flexible framework for constructing plots from standard components. These two components shall be discussed in more detail below. The new visualization framework triggered the implementation of many new plot types. Currently, MAYDAY offers four tabular views and 31 visualizations, two of which will be presented in more detail in sections 4.2 and 4.3.

4.1.1. The View Model

Data content & interaction

The 'Vis3' visualization data model (shown in figure 4.1) provides a unified system that all visualizations are built on. Starting with a selection of `ProbeList`s, the `ViewModel` is created. It contains the selected ('original') probe lists together with the set of all `Probe`s contained in them as well as a list of 'optimized' probe lists. Based on an ordering of the original probe lists, which can reflect their native order in the `DataSet` or any user-defined order, each probe is associated with a *top priority probe list* by a simple algorithm: All probes contained in the first probe list (according to their ordering) are assigned to that list. The intersection of the second probe list and the remaining probes is computed and the result is the second 'optimized' `ProbeList`. The process is continued until all probes have been assigned to a list and for each original probe list, an 'optimized' list has been created. The purpose of these disjoint optimized lists is to help programmers write efficient visualization code. The profile plot, for instance uses the 'optimized' lists to make sure that each probe is rendered exactly once.

Plots use the `ViewModel` to access each probe's expression data, rather than directly requesting the data from the `Probe` object. This allows for the implementation of *online data transformations* that affect the way data is plotted but does not require changing the original data matrix or implementing special transformations for each

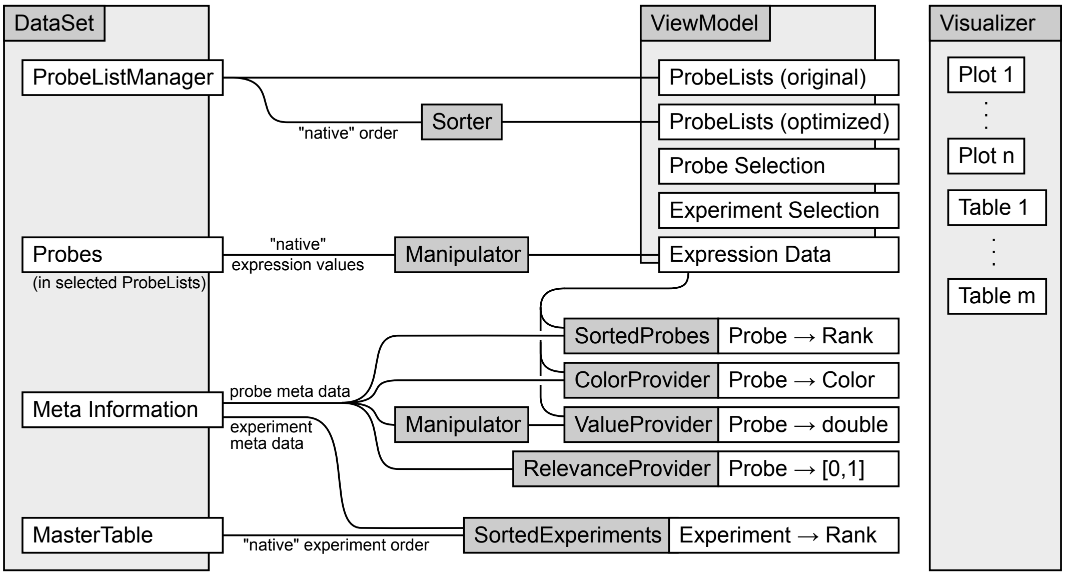


Figure 4.1.: Components of MAYDAY's visualization data model. All connected plots and tables are managed by a `Visualizer`. The `ViewModel` holds selections and supplies the plot with data from the `DataSet`. Further classes encapsulate often-needed functionalities such as mapping `Probe`s to colors (`ColorProvider`). These classes also provide their own `Setting`s for user interaction.

plot. Such transformations include the logarithm, centering each probe's values (such that each probe's expression vector's mean is zero), scaling the values (such that the expression standard deviation is one), applying the z-score transformation (centering and scaling), smoothing using a sliding window, etc. Transformations can also be combined ('stacked') and more transformations can easily be added as plugins.

Another important role of the visualization data model is the sharing of selections between connected plots (so-called 'linked views'). If several visualizations (e.g., one heatmap, one scatter plot and one profile plot together with an expression table) are opened within the same `Visualizer` (see section 4.1.2), selecting probes in one of them will automatically select and highlight the same probes in all other connected visualizations to facilitate understanding. For very complex analysis tasks, selection sharing can also be enabled between different `ViewModel`s, even when they are not visualizing the same `DataSet`. In the latter case, selections are matched based on the names of selected objects.

Additional data aspects

The primary expression data (resp. its transformed form) is often not enough to create helpful visualizations. MAYDAY's visualization data model offers a range of so-called *providers* that programmers can use to access different aspects of the underlying data.

These include the `ValueProvider` which maps each `Probe` object to a numeric value, which is either one of the (transformed) experimental values, or the content of a numerical meta information object attached to the `Probe`.

The `ColorProvider` assigns a color value to each probe, based on a user-selectable source which can be the top-priority probe list for the given probe, any of the probe's experiment values or any attached meta information. Numerical data is mapped to a predefined or user-defined color gradient using a user-defined linear or sigmoidal mapping function and various other parameters. Categorical data (e.g., from meta information) is mapped to an appropriate number of (configurable) distinct colors, and a legend is displayed that shows the mapping.

Meta information can also be used to assign a *relevance* value to each probe. This is accomplished by the `RelevanceProvider` which uses numerical meta information, for instance p-values from a statistical test, applies a user-defined mapping function (linear, logarithmic, exponential), and interprets the resulting value in the range $[0, 1]$ as the relevance of the probe with zero signifying low relevance and one signifying high relevance. This replaces the original system [57] where numerical meta information had to be converted to a special class of *relevance meta information* before it could be used for visualization. Plots can use this information to enhance the visualization, e.g., by using transparency and opacity to reduce the visual 'strength' of irrelevant probes and increase that of relevant ones, respectively.

Probes as well as experiments (rows and columns in the heatmap, for example) can be sorted according to several properties, such as their name, their *display name*, attached meta information, expression values (probes only), and trees obtained by hierarchical clustering.

Extensibility and integration

All components of the visualization data model offer `Setting`s (see also section 3.4) that allow users to change configurable properties. By using these classes, programmers can quickly implement rich and highly useful visualizations without dealing with the intricacies of how to compute relevance, how to correctly (re-)map values to colors when data transformations are suddenly changed by the user, or how to define color gradients, for example. Furthermore, using the same components for all plots ensures a 'smooth' user experience without unwelcome surprises. Configuring these data aspects is always done in the same user interface and with the same, well-defined results.

When using one of the *providers* or another building block of the visualization framework, programmers can add the component's setting to the overall `Setting` object of their visualization. Using the methods described in section 4.1.2, these settings are combined and integrated into the menu bar of the respective plot window (or another location if the outer container is not a `PlotWindow`). While the code generating these menus is quite complex to ensure it correctly works with changes in settings' values, as well as with the addition and removal of sub-settings (e.g., when a plot is detached, see section 4.1.2), the code needed to create highly-configurable plots is extremely simple.

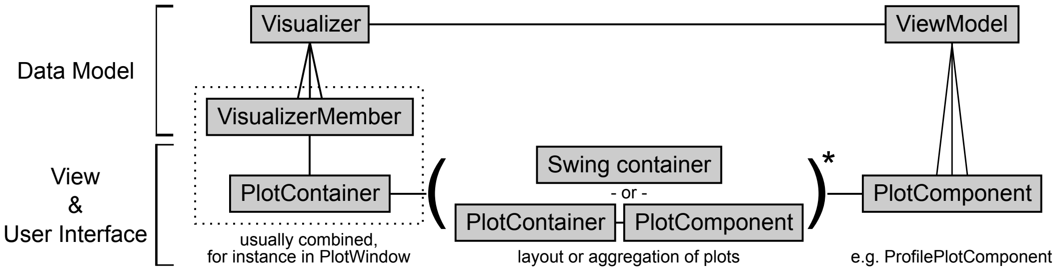


Figure 4.2.: Overview of Mayday's visualization framework. The `Visualizer` connects all visualizations based on a common `ViewModel` (sharing selections, data transformations etc.). Individual visualizations implement the `PlotComponent` interface and are connected to the `ViewModel`. Plots can be aggregated or laid out using any number of standard Swing components or intermediate wrappers of hybrid classes (implementing `PlotContainer` as well as `PlotComponent`). The top-level user interface component containing the visualization(s) implements the `PlotContainer` interface and is responsible for displaying interaction elements for the plot's settings. The `VisualizerMember` interface is implemented separately from the `PlotContainer` interface to also allow non-GUI implementations of plugins (e.g., export plugins).

4.1.2. Plot Components

Swing/AWT integration

Java already offers many components that can be used to create graphical user interfaces but can also be helpful for creating rich visualizations. Among them are the basic Swing/AWT classes for panels, scroll panes, and windows, but also the `LayoutManager`s which arrange groups of components according to different systems. These components also mediate user interaction by capturing mouse and keyboard events, changing their state accordingly (e.g., scrolling the viewport of a scroll pane) or redirecting them to the appropriate receiving components. Mayday's plotting framework allows visualization programmers to make use of all these existing components to build their plots. In fact, many of the helper classes available as part of the Mayday visualization package work directly on Swing components. The `ZoomController` class, for instance, which adds mouse wheel zooming capabilities to a component, can in principle be added to any existing Swing object.

Since the visualization framework is built on standard Swing components, the framework code was written as generic as possible. One example is Mayday's plot export functionality. Any plot can be exported into raster image formats (PNG, JPG, TIFF) as well as into vector formats (SVG, PDF) using the same classes and offering the same user experience without any need to implement such functionality separately for each plot. Additional export formats can easily be added as Mayday plugins.

Connecting plots with the data model

Two important interfaces are defined to integrate visualizations with the framework and ensure that data can be passed from Mayday to these plots. These are `PlotComponent` and `PlotContainer`. `PlotComponent` defines two methods, namely `setup(PlotContainer)` which is used to inform the plot that it has become part of a plot container, and `updatePlot` which instructs the plot to discard any buffers and redraw its content. `PlotContainer` defines methods with which any contained `PlotComponent` can register user interface elements (such as `Settings`, additional menus and a title for the plot) as well as a method to access the underlying visualization data model (see 4.1.1). The `PlotContainer` is responsible for aggregating settings of all contained components (e.g., via submenus) and presenting them to the user. This system allows for very flexible presentation of visualizations. `PlotContainers` are implemented by the "traditional" GUI plot windows (`AbstractVisualizerWindow`), by containers that can be used to create and manipulate plots in a programmatic fashion (`ScriptablePlotContainer`), by plots that can be detached from a GUI window (`DetachablePlot`), as a combination of several plots in one view (`MultiPlotComponent`, implementing both interfaces), as well as for embedding any plot in the Mayday Graph Viewer MGV (`Vis3Component`). `PlotContainers` that can be used as the *outermost* container, i.e., GUI windows, usually also implement `VisualizerMember`, an interface which the `Visualizer` associated with the `ViewModel` can access to organize all connected plots, e.g., to bring windows to front or to close them.

A `PlotComponent` does not have to be added as a direct child of a `PlotContainer` (such as a `JPanel` as direct child of a `JWindow`). Any number of standard Swing (and AWT) components can be nested inbetween (see figure 4.2). The methods of the two interfaces are used to connect the `ViewModel` with the `PlotComponent` as well as to bring the `PlotComponent`'s interface elements to the `PlotContainer`. Once the `PlotContainer` is activated (e.g., via `addNotify` when being added to a GUI window), it notifies all contained `PlotComponents` of itself. These can then access the `ViewModel`, set up their content and export their interface elements (settings, title, etc.). If a `PlotComponent` is removed from a `PlotContainer` and added to another (e.g., when a plot is transferred from one window to another), it will be notified of this change automatically by the Swing methods `removeNotify` and `addNotify`. It can thus remove any listeners that are no longer needed and re-export interface elements to the new `PlotContainer`.

4.1.3. Efficiency

When visualizing large amounts of data, naive rendering approaches quickly show that they do not scale well. Interactivity is key in visual analytics and Mayday uses several strategies to keep plots responsive even when showing large datasets.

- **Buffering** is an often-used strategy to avoid repeatedly rendering the same data and is used in almost all Mayday plots. Implemented in the `AntialiasPlotPanel` class, buffering is integrated with zooming (where upon zooming, a scaled version of the buffer is displayed while a new plot is prepared in

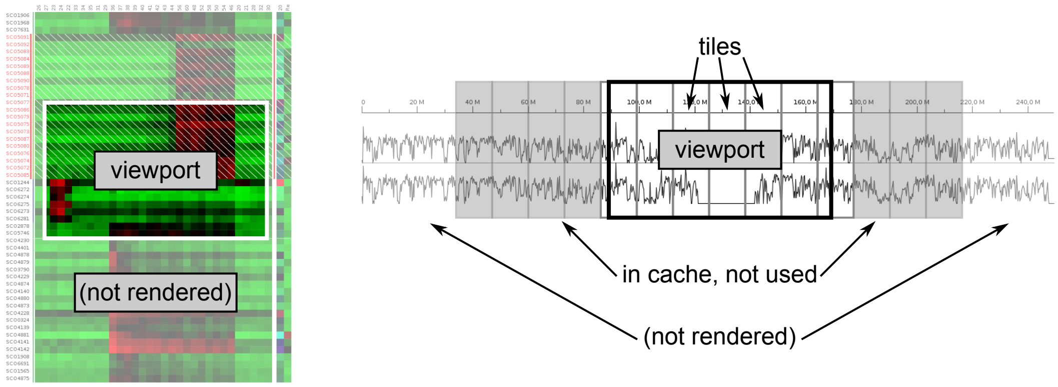


Figure 4.3.: Clipping (left) and partial, tiled buffering (right) are strategies to speed up rendering. Using clipping, plots only need to render elements that are within the viewport, meaning that they are currently visible to the user. Tiled buffering improves on this strategy by keeping small parts of the complete view (so-called *tiles*) prepared so that they can quickly be displayed when they are overlapping the visible area.

a background task). By adding their plot to an `AntialiasPlotPanel` or extending that class directly, programmers automatically add buffering to their visualization.

- **Event aggregation** can significantly reduce the amount of time spent for updating visualizations. Changes to the underlying data or the data transformation, for instance, require updating a visualization. These changes are also communicated to the *Providers* which need to update their state accordingly. For example, a `ColorProvider` needs to know the minimum and maximum of all data values to be able to map them to the color gradient. Thus, after recomputing the minimum and maximum, the `ColorProvider` will also create a change event and notify the visualization. Furthermore, changes to configurable properties of a *Provider* or of the plot itself will create change events. If the plot were to redraw on each such notification (possibly even blocking the thread which sent the event), many unnecessary updates would be performed and Mayday's user interface would become unresponsive for a long time. Instead, Mayday plots aggregate incoming update requests in a window of at least 100 ms, meaning that if an event is received, an update will be performed after 100 ms if no further event is received during that time. Otherwise, the timer is reset and the update is deferred for another 100 ms.
- **Clipping** (as shown in figure 4.3) is a very common strategy. Instead of rendering the whole visualization, only the parts actually visible are updated. Mayday's heatmap implementation (described in section 4.2), for example, determines which rows and columns are currently visible and only draws those cells falling into the intersection of visible rows and visible columns.

Based on these methods, plots can also implement more complex strategies to improve efficiency, such as partial and/or *tiled* buffering (see figure 4.3). Partial buffering can be a solution when buffering the whole plot is infeasible (as in Mayday's genome browser, see section 4.3) or when the plot consists of several parts some of which profit from buffering while others do not (as in the heatmap, section 4.2). Further improvements can be obtained by splitting the buffer into multiple smaller buffers, called ‘tiles’. Rendering a tile requires only a fraction of the whole buffer’s rendering time, and as only part of the whole plot area is represented by tiles (which also do not have to represent a contiguous area in the plot), this results in an overall reduction of the time spent for rendering. An example of the speed-up that can be achieved is given in section 4.3.

4.2. Reimplementation of the Enhanced Heatmap

A heatmap plot is a visual representation of a table of numeric or categorical values, where instead of displaying the table cell content in its textual representation, each cell is colored according to its content using a mapping of content values to distinct colors (categorical) or to a color gradient (continuous values). The concept was first used for population data [101] by Loua in 1873 (see [179] for a history of this visualization). The first application to large-scale expression data was presented in 1998 [175]. Improvements of the original concept include the addition of clustering trees [99] and meta information [57].

Originally, the heatmap in Mayday was one of four visualizations (heatmap, box plot, profile plot and multi profile plot). These four plots were statically included in menus and mostly consisted of a few large and complicated Java classes each. This first implementation was extended such that meta information could be included to *enhance* the heatmaps which can greatly increase the amount of information they convey and their usefulness in data analysis [57]. The second heatmap implementation was built as part of the new visualization framework (described in section 4.1). Problems in parts of Java’s table rendering classes on MacOS systems triggered the development of a third incarnation of the heatmap concept, this time independent of the table rendering code. (The problem has since been resolved in Mac Java).

The heatmap is built around a central data structure which connects to the `ViewModel` and keeps track of the columns, column groups (see below) and row headers that are part of the visualization, including the width of each column and the height of each row. Row heights can be scaled globally as well as individually based on *relevance* meta information. When updating the plot contents, this central structure computes the set of cells that need to be repainted and calls the rendering code of the respective delegate objects (columns or headers).

All on-screen elements of the heatmap are realized as plugins, each with its own `Settings`. On the first hierarchical level, the plot is organized into the main area where heatmap cells are rendered, an area to the left where row headers are placed, and an area above for column headers (see figure 4.4). This splitting of rendering responsibilities into small, well-defined elements allows for quick implementation of new rendering styles and additional column plugins (some shown in figure 4.5), such as the expression bars and expression box styles which are especially useful for conveying the heatmap information in black and white prints.

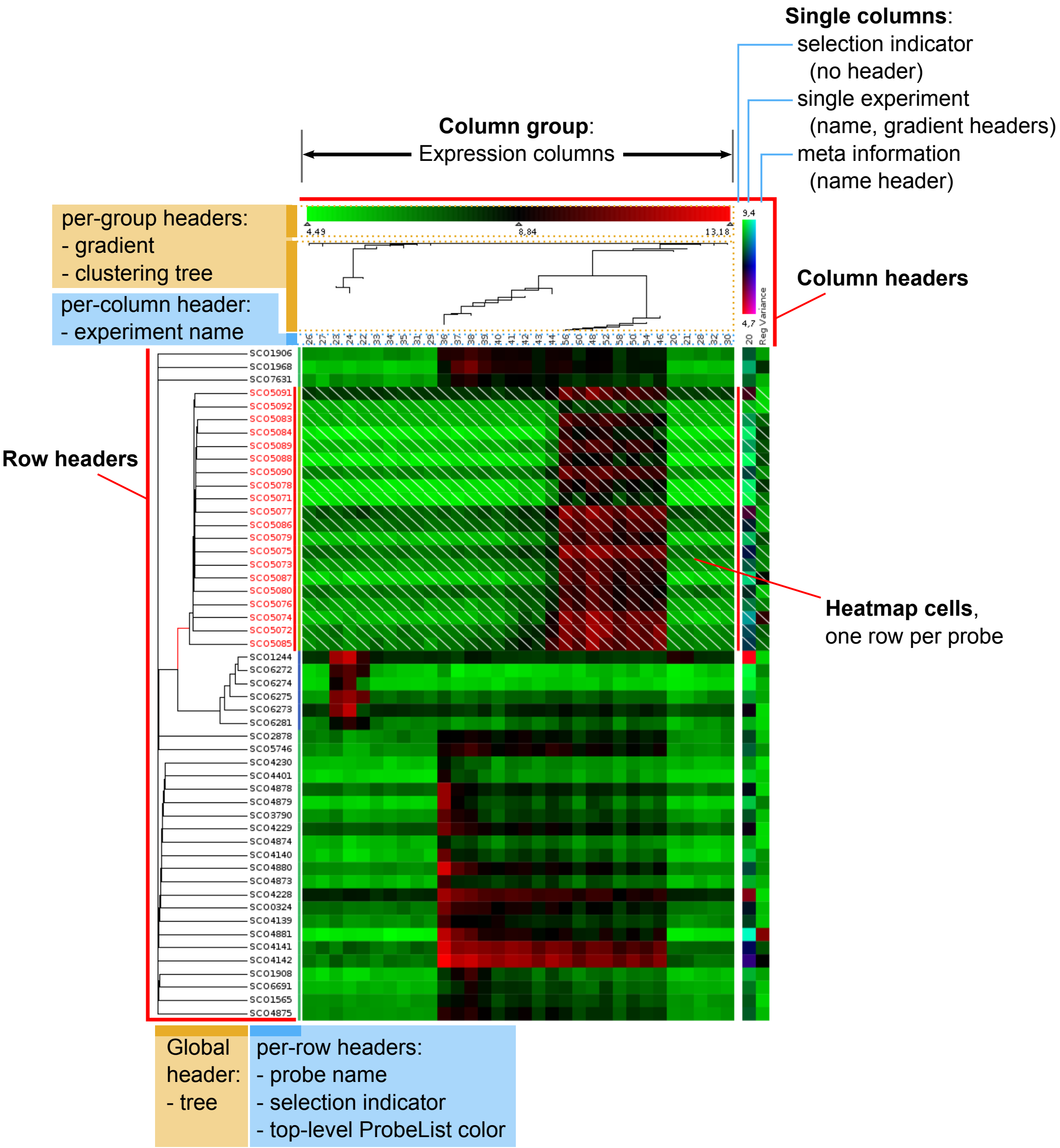


Figure 4.4.: Structure of a heatmap plot. The plot area is divided into three major parts: heatmap cells, column header area, and row header area. Each row represents one probe, each column visualizes one aspect. Several consecutive columns with identical properties are combined into column groups. Single columns are internally handled as single-member groups. Column headers are rendered either per column (e.g., names) or per column group (e.g., clustering trees). Row headers are rendered either per row (e.g., names, selections) or for the whole heatmap (e.g., clustering trees).

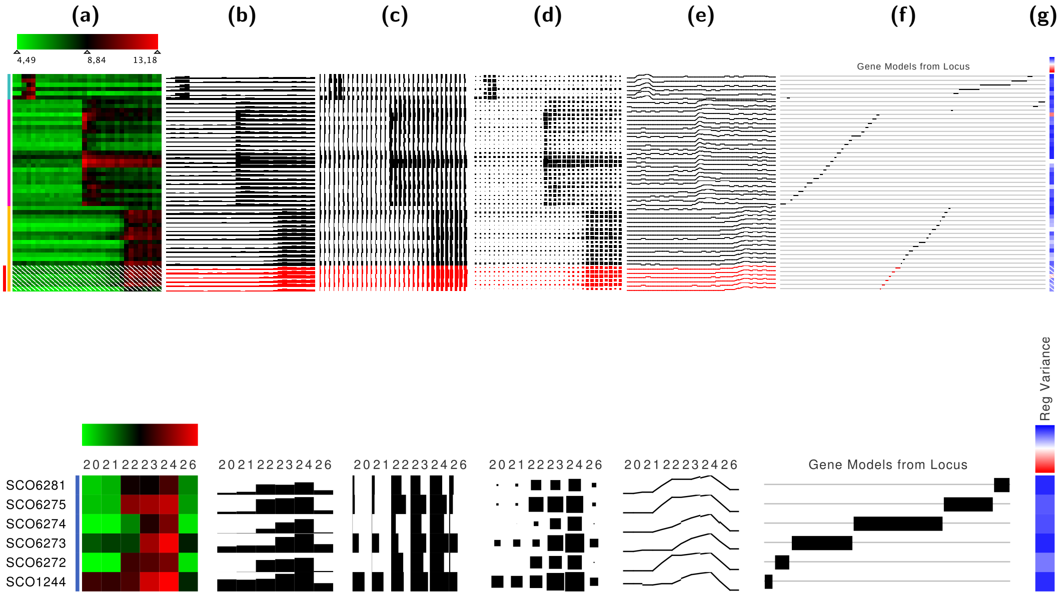


Figure 4.5.: Different heatmap rendering styles implemented in Mayday. Here an example for time-series expression data of three genomic clusters is shown (data from [111]). Top, zoomed-out display showing a conventional color-gradient heatmap (**a**, with large gradient legend on top), vertical expression bars (**b**), horizontal expression bars (**c**), expression boxes (**d**), and expression profiles (**e**) as well as two meta-information columns, one showing the relative location and exon structure of the respective genes (**f**), the second showing expression variance on a red-blue gradient (**g**, with small gradient legend on top). Selected genes are highlighted in red color or with diagonal lines, depending on the rendering style. At this zoom level, row and column names are automatically hidden. Bottom, zoomed-in display restricted to the first six rows and columns of the matrix. When more space is available, rows and column labels can be displayed. To display column headers also for very narrow columns (such as the meta-information column), header labels are automatically rotated. Many view elements have configurable options. For example, the gradient headers can be rendered with or without value indicators for minimum and maximum and the height of the vertical gradient can be adjusted.

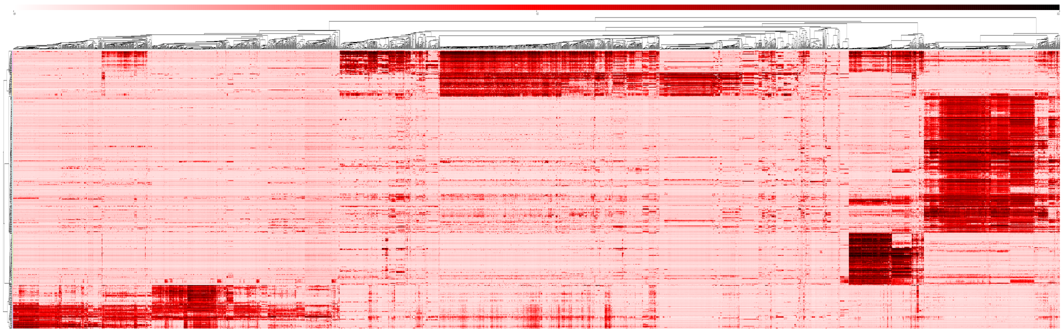


Figure 4.6.: A traditional heatmap plot visualizing the 435 most variant probesets of the E-TABM-185 dataset from ArrayExpress. The matrix displayed has 435 rows and 5,896 columns. Rows and columns are both clustered hierarchically using the UPGMA algorithm based on the Pearson correlation distance. Clustering trees are displayed to the left and atop the matrix. Expression values are mapped to an inverse heat colors gradient (white–red–black), shown at the top of the plot.

The basic invariant of the heatmap is that all visual elements on one row are related, i.e., they visualize one aspect of the same probe. Thus, a row header relates to all cells in the respective row and row header plugins are managed by the main heatmap class. Columns, on the other hand, can visualize quite different information, such as expression values (primary data) or meta information (secondary attributes). Often, several columns form a logical group based on common properties. For instance, a dataset with 20 experiments (columns in the expression matrix) is usually visualized in a heatmap with 20 columns, each of them visualizing the expression values of one of the experiments. The fact that columns often are grouped is reflected in the heatmap design: The central data structure manages a list of *column groups*, each group being created by a plugin instance. The column group plugin is responsible for creating the appropriate number of columns and managing its own column header plugins.

To find out which column header plugins can be used for a given column group, column groups can define properties. For instance, a clustering tree column header can only be used on column groups implementing the `HasExperimentTree` interface (e.g., on expression columns of a clustered dataset, but not on a single meta information column).

User interaction is handled by the main heatmap structure (row selection events and scaling), by the row resp. column header plugins (e.g., enlarging the area used for rendering a clustering tree), or by the columns (e.g., selection, tooltip display).

The heatmap can include meta information in three different ways: Additional columns can be included that show meta information mapped to colors (using a color gradient for numerical data, or a set of distinct colors for categorical annotations), or using a domain-specific visualization (e.g., for gene models as shown in figure 4.5). Numerical annotations can also be used as *relevance* information, and these are then visualized as color shading overlaying the coloring of an expression column, or by modifying the transparency values of the expression column's colors.

Finally, any meta information with a defined ordering can be used to sort the rows of the heatmap, as well as the columns in a group of expression columns.

The heatmap uses a combination of the approaches discussed in section 4.1.3 to speed up rendering. Firstly, clipping is employed to compute the heatmap cells, row and column headers that require updating. Secondly, each rendering plugin author can decide whether to use direct painting or a buffered approach. For conventional heatmap columns, rendering a cell is usually a very simple operation, consisting of the application of a uniform color fill. Thus, drawing all visible cells can be accomplished very quickly and buffering is not needed. Other elements, such as a clustering tree header, can benefit from buffering and use the facilities offered by the `AntialiasPlotPanel` class.

The efficiency of the new implementation reveals itself when analyzing large datasets. One such example is the ArrayExpress [115] dataset "E-TABM-185" comprising 5,896 hybridizations performed with Affymetrix HG-U133A microarrays. More than 370 biological conditions in human samples are accumulated in this dataset, each experiment covering 22,283 probesets (in the "processed" version of the dataset). Mayday can easily handle such large datasets given enough memory (in this case, 16GB of main memory were assigned). Gene variances were computed and the most variant genes selected (435 genes with variance > 9). The data was clustered both on rows (probes) and on columns (experiments) using the UPGMA algorithm and the Pearson correlation distance measure and the heatmap plot for the full matrix (not shown) as well as for the set of highly variant probes (figure 4.6) was created.

4.3. Expression in a genomic context: ChromeTracks

Gene expression happens in a genomic context. Transcripts are copied from the genome regulated by elements upstream (and sometimes downstream) of their transcription start site (TSS). Furthermore, in prokaryotes, several genes are often transcribed as one operon, resulting in a polycistronic transcript. Eukaryotes, on the other hand, have complicated post-transcriptional processing mechanisms so that mapping the expressed sequences back to the genome sequence can reveal the intron/exon structure of their genes.

Visualizing expression in its genomic context can often lead to important conclusions. For instance, genomic co-location of transcripts with similar expression profiles can reveal genomic clusters of genes under common regulatory control, or with a functional relationship (e.g., in so-called 'pathogenicity islands' [65]).

Based on the diploma thesis by Christian Zipplies [185], Mayday's genome browser ChromeTracks was developed to show any element of Mayday's data model in a genomic context. ChromeTracks is a *track*-based visualization: The x axis of the plot is associated with the chromosomal position (in bases) of the currently visualized chromosome. The y axis is used to distinguish different tracks, each of which displays one data aspect, such as the expression value in one of the experiments, meta information like p-values, or additional, external annotation (e.g., gene models, chromosome sequence information).

Table 4.1.: ChromeTracks track types

Track type	data	strands	#	visualization
— Probe data				
Profile	primary (1 exp.)	one	1	profile curve
Single Heatmap	primary (1 exp.) / meta	both	2	colored fixed-size boxes
Multi Heatmap	primary (all exp.)	both	2	colored fixed-size boxes
Stem	primary (1 exp.) / meta	one	3	colored variable-height boxes
— RNA-seq data (via SeaSight, see chapter 7)				
Mapped reads		both	1	colored read objects
Read coverage		both	1	coverage curve
— Chromosome annotation				
Chromosome sequence		both	–	full IUPAC alphabet [40]
Locus data, genetic coordinates (see 7.6.8)		both	–	
— External annotation				
Sequence data, e.g., from FastA files		both	–	full IUPAC alphabet [40]
Numerical data, from 'Wiggle' files		both	1	Wiggle curve

Track types are listed together with the data they use for rendering and the visualization method. Some track types can only display data for one strand (forward or backward), others can include both strands in a single track. #: Number of visual characteristics (color, size/location, transparency) used for data mapping.

ChromeTracks provides the core data structures needed for such a visualization and for constructing the track-based user interface, including event handling, coordinate transformations and optimized rendering strategies (see below). The actual track rendering is realized using Mayday's plugin management system: Each track type is implemented as a `TrackPlugin` with specific `Settings`. Data attributes are mapped to all or some of the three visual characteristics, color, size/y-position, and transparency. Several plugins are implemented for primary as well as meta data and to include additional annotations from internal (the DataSet) or external (the file system) sources. SeaSight (described in chapter 7) provides additional track rendering plugins for some of its own RNA-seq data types (see table 4.1 for a list of currently available track types).

Two special tracks are included in each visualization (see figure 4.7): Firstly, the ruler ('minor scale') indicates the current base position and can be used to jump to another location as well as to zoom in. Secondly, the global location indicator ('major scale') is a track which is not part of the x-axis coordinate system. Rather, it always shows a ruler representing the whole chromosome currently visualized. It can be used to jump to any location within the chromosome as well as to zoom in on any region. Users can switch to another species resp. chromosome from the view menu.

ChromeTracks offers many interaction possibilities for horizontal (base resolution) zooming, vertical (track height) zooming, panning and scrolling, probe selection and track configuration. Horizontal zooming and panning is available both through the major and the minor scales as well as using the mouse wheel. Tracks can be zoomed

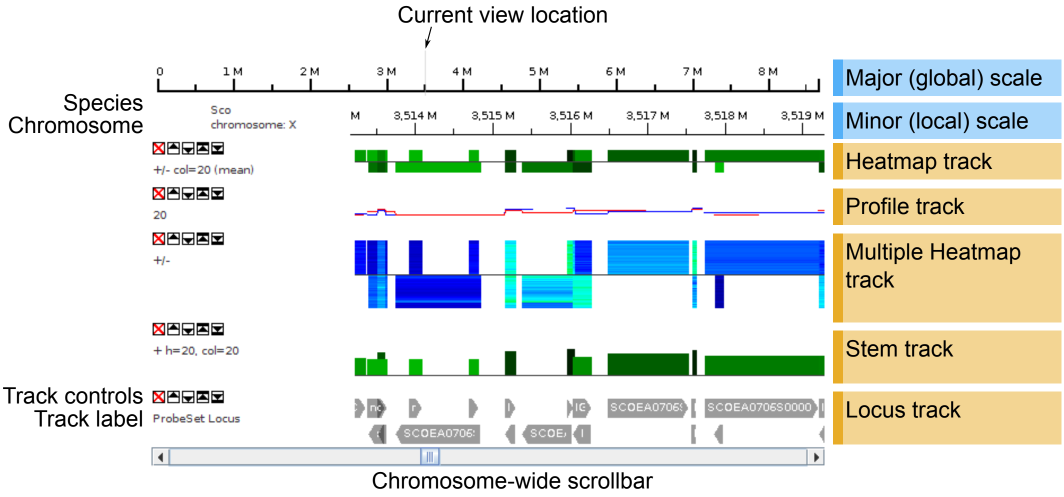


Figure 4.7.: ChromeTracks view. The two scales can be used for navigation (zooming and panning). Tracks can be rearranged by drag&drop or using the track controls. Each track is labelled either automatically, describing the mapping of data aspects to visual attributes (e.g., 'col=20' indicates that color is used to visualize probe data for the experiment labelled '20'), or by a user-defined label (e.g., 'ProbeSet Locus'). Several track types are shown as examples, displaying data of the *S. coelicolor* dataset described in 3.7. Further track types are listed in table 4.1.

(enlarged/shrunk) vertically using the mouse wheel. Probes can be selected either directly or by selecting a genomic range. Locations can be jumped to by specifying a base position, searching for an element or centering the currently selected probes in view. Tracks can be arranged by drag&drop and are automatically placed in a non-overlapping layout. Optionally, this behaviour can be disabled to allow tracks to overlap which can be useful to relate the information presented in different tracks (e.g., two wiggle tracks). Using the plugin system and the `Settings` framework's serialization capabilities, the current track layout and all tracks' settings can be stored to disk and restored during later analysis sessions.

Furthermore, each track can implement *tooltips* that display location-specific information whenever the mouse cursor is placed above the respective track. Such information can, for instance, include the current position (in base coordinates), elements associated with this position, and numerical values such as normalized read coverage.

To efficiently display data in the context of large eukaryotic chromosomes and allow for smooth scrolling and panning, a sophisticated tiled rendering system (see also 4.1.3) was implemented. Each track along the chromosome is split into tiles of a fixed width (in pixels) such that a certain interval of bases is represented by each tile, depending on the current zoom level. Based on the current viewport position (base offset in the chromosome), a track requests tiles from its *tile cache* and copies the pre-rendered images to the screen buffer. If a tile is currently not available, a request is placed in the track's *rendering queue* and the respective tile is rendered in a background task as soon as possible. This happens independently for each track, allowing for parallel rendering of all tracks. Whenever a tile is ready, it is placed into the cache, and an event is created notifying the track that there is new content ready for display.

Changes in the viewport position (resulting from scrolling or jumping to a new position) are communicated to the tracks' tile caches which automatically place all visible (but not yet rendered) tiles into the rendering queue. In addition, a certain number of tiles to the right and left of the current viewport are pre-rendered to prepare for (continued) local scrolling by the user. When the user scrolls the viewport by a small amount (e.g., to the left), existing buffers can be used to display the new area right away. The new center of the viewport is used to update the area that should be buffered and new buffers (here: to the left) are prepared while old ones (here: to the right) can be discarded if memory should be recovered.

In total, an area of 20,000 pixels width (corresponding to 20 not necessarily consecutive tiles, each of width 1000 pixels) is cached for each track. Event aggregation is used to prevent unnecessary updates when several aspects of the visualization are changed. For instance, during mouse-wheel zooming, several zoom levels are usually skipped before the user interaction stops and the view settles on the final zoom level. As zooming results in the invalidation of all cached tiles, event aggregation is an important efficiency improvement here.

The tiled approach was chosen for two reasons: Firstly, the genome browser deals with huge amounts of data which require, at times, quite complex operations for drawing (for instance when the layout of overlapping RNA-seq reads is computed). Thus, clipping is not sufficient as even rendering only the currently visible area can be very time-consuming when the visible genome region is large. Here, re-using cached tiles is the optimal strategy to quickly accommodate small movements along the chromosome without any expensive redrawing. Secondly, caching an image of the whole track is infeasible because users would have to wait a very long time before the track image was ready for display, even if they were only interested in a small region of the genome.

For illustration, consider a simulated example of 250 million reads (of length 32bp) distributed approximately uniformly over human chromosome 1 (249 mb). Initially, the tile cache is empty. After zooming to a resolution of one pixel per base and scrolling to any location within the genome, the first tile is rendered and displayed after 18 ms and the full view (1280 pixel width) is ready after 54 ms (as it requires at most three tiles). In comparison, rendering the whole chromosome at this resolution (which is equivalent to rendering 249,213 tiles) would require 76 minutes. In fact, in most real-world use cases it is simply impossible to cache an image of the whole track, as the image would be too large to keep in memory (if images of such large dimensions were even possible using Java Swing objects). Again consider human chromosome 1 at base resolution: The image for a complete track of medium height (50 pixels) would be about $249,000,000 \times 50$ pixels large, i.e., for each such track a *12.5 gigapixel* image would have to be cached. If the letters of the chromosomal sequence were to be displayed as well, the image size would further increase: If, say, 10 pixels of width were used for each letter, the resulting image would be 2.5 billion pixels wide and comprise a total of 125 gigapixels.

If a ChromeTracks visualization is exported to a bitmap or vector graphics file, instead of using the tiled rendering approach, the track plugins directly render to the image's `Graphics` object. For the vector file formats (SVG and PDF), this means that the view is exported as editable and scalable objects instead of as one possibly huge bitmap image.

4.4. Analysis of time-series transcriptome data

Originally, MAYDAY was designed to analyze data from relatively simple studies, often considering a number k of conditions $C_1, \dots, C_k$ where k was usually quite small, e.g., in the range of two to five. For each condition, several samples (replicates) would be hybridized to arrays, leading to experiments $(E_{11}, \dots, E_{1r_1}), \dots, (E_{k1}, \dots, E_{kr_k})$, where k_i is the number of replicates hybridized for condition i. Statistical testing for such study designs is straightforward and many established methods exist for the two-sample case (i.e., $k = 2$). The multi-sample case is covered by methods such as ANOVA [51, 52].

Decreasing costs for microarray experiments as well as growing interest in *Systems Biology* questions led to larger and more complex experiments, most notably time-series studies. Generally, experiments performed in the context of time-series studies involving multiple conditions can be described as $E_{ij\ell}$ where i is the condition, j refers to the time-point, and ℓ denotes the replicate. However, such studies often trade replicates for additional time-points. In fact, the high quality of microarrays and the near-perfect reproducibility of experiments performed under strict quality control can make creation of replicates for time-series studies unnecessary [11]. Time-series studies thus often contain experiments $(E_{11}, \dots, E_{1p_1}), \dots, (E_{k1}, \dots, E_{kp_k})$, where p_i is the number of time-points sampled for condition i. Each experiment E_{ij} is annotated with a time-point t_{ij}, specified in some unit relevant to the respective study, e.g., "hours since treatment".

Statistical analyses of (multiple) expression time-series studies are still subject to research. The two-sample statistics employed to find differentially expressed genes in non-time-series studies can not be applied here – not only for lack of replicates but also because it is quite inappropriate to define two groups to compare. There are often many more time-points than conditions, such that comparing all pairs of two time-points is not feasible. In most cases, this procedure would be unable to answer the researchers' question, which is: "Which genes *behave differently* (in a significant way) between different conditions tested, over time?". Such a question can not easily be translated into a statistical test. Additional complications arise from the multi-dimensionality of the comparisons. Time-points and conditions (see section 2.4.1 for a definition of these terms) already define two dimensions of interest, and studies can add further dimensions, e.g., by including different species or mutants. Conditions can also be defined in such a way that they can be expressed as multi-dimensional attributes, e.g., by combining (1) different growth media with (2) different temperatures in (3) anaerobic and aerobic environments. Finally, the time-points sampled for the different conditions may either not be the same (e.g., due to lack of manpower for obtaining samples synchronously), or they might appear

Table 4.2.: Time-Series Statistics Windowing Methods

Separator	Single window	Adjacent windows	Free windows
$\underline{123}45678$	$\underline{123}45678$	$\underline{123}45678$	$\underline{123}45678$
$\underline{1234}5678$	$1\underline{234}5678$	$1\underline{234}5678$	$\underline{123}45678$
$\underline{12345}678$	$12\underline{345}678$	$12\underline{345}678$	$\underline{123}45678$
	$123\underline{456}78$		$1\underline{234}5678$
	$1234\underline{567}8$		$1\underline{234}5678$
	$12345\underline{678}$		$12\underline{345}678$

The four windowing methods available in Mayday's time-series statistics plugin are explained using an example of 8 time-points (1-8) with the (minimal) size parameters set to 3 for each method. Each method creates pairs of groups by splitting the set of time-points in different ways. Groups are distinguished by underlining all time-points in the first group, unused time-points are printed in gray. The pairs of groups are used as the basis for applying pair-wise statistical tests.

to be identical (i.e., $t_{ik} = t_{jk}$) but for biological reasons they should not be compared directly (e.g., one sample might, for some reason, show a delay in the response to the treatment that is unrelated to the type of treatment itself).

As a first step towards determination of time-dependent differentially expressed genes, two approaches have been implemented in Mayday: The *time-series statistics plugin* allows for the computation of statistically significantly differentially expressed genes in a time-series (described in the next section). Tiala, a visual analytics method for the analysis of multiple time-series datasets is presented in section 4.4.2.

4.4.1. Statistical analysis of single time-series

If only a single time-series (without replicates) should be analysed, i.e., if experiments can be described as $E_1, \ldots E_p$, the question of what constitutes differential expression over time can usually be answered relatively easily. Still, depending on the details of the study and on the effects one wants to discover, different answers might fit best (see also figure 4.8). Researchers might be interested in genes that

- are permanently switched on (resp. off) at some (unknown) point in time
- are transiently up- resp. downregulated in some (unknown) time interval
- show a significant switching event at some (unknown) point in time, regardless of their behaviour over the rest of the time-series
- show significant expression differences between two (unknown) disjoint intervals anywhere in the time-series

The Mayday time-series statistics plugin is a first step towards offering a tool to address these questions in an automated fashion based on two-sample statistical tests. Essentially, it tackles the need for a definition of two groups to compare against each other by testing different pairs of groups (each group of size ≥ 3), defined according to one of four windowing systems shown below. For sake of exposition, the methods are introduced with an example in table 4.2.

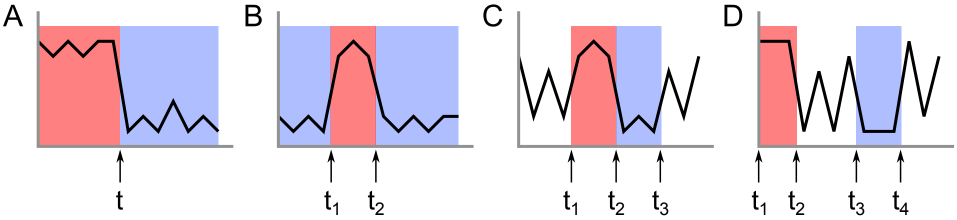


Figure 4.8.: Types of time-dependent events that can be detected with MAYDAY's time-series statistics plugin. Pairwise statistical tests are performed between the time-points in the red group and those in the blue group. A, persistent switching; B, transient change; C, local switching event; D, general differential expression. See the text for details.

- **Separator:** The experiments are split into two groups according to one time-point t which is moved along the time-series (figure 4.8A). The first group contains all experiments up to t, the second group contains all remaining experiments. This addresses the question "*Which genes are permanently switched on (resp. off) at some (unknown) point in time?*". The minimal size of each group can be set as a parameter.
- **Single window:** The experiments are split into two groups according to two time-points $t_1 < t_2$ which are moved along the time-series, keeping the number of experiments $t_2 - t_1$ constant (figure 4.8B). The first group contains all experiments in this window, the second group contains all remaining experiments. This addresses the question "*Which genes are transiently up- resp. downregulated in some (unknown) time interval?*". The number of experiments in the window can be defined as a parameter.
- **Adjacent windows:** The experiments are split according to three time-points $t_1 < t_2 < t_3$ into four groups: Experiments before t_1 are ignored, experiments between t_1 and t_2 form the first group, experiments between t_2 and t_3 form the second group, and experiments after t_3 are ignored (figure 4.8C). The biological question here would be "*Which genes show a significant switching event at some (unknown) point in time, regardless of their behaviour over the rest of the time-series?*". The number of experiments in the windows ($t_2 - t_1$ and $t_3 - t_2$) can be defined as parameters and are kept constant.
- **Free windows:** This is a generalization of the above method, where the two windows are not kept adjacent to each other, i.e., the set of experiments is split into groups according to four time-points $t_1 < t_2 < t_3 < t_4$ such that the first group contains all experiments between t_1 and t_2 and the second group contains all experiments between t_3 and t_4 (figure 4.8D). The very general question here is "*Which genes show significant expression differences between two (unknown) disjoint intervals anywhere in the time-series?*" Another way to find such genes (without a statistical significance assessment, however) would be to compute the per-gene expression variance.

As a result, each probe is assigned an aggregated 'p'-value which (even if it is not a p-value in the strict statistical sense, e.g., when using the mean as aggregation function instead of the minimum), can be used to rank genes with respect to how 'differentially' they behave. Since many tests are performed for each gene, and many genes are tested, corrections for multiple testing are important. The time-series statistics plugin uses two correction steps, called "Inner correction" and "Outer correction", respectively. The complete computation is performed in five steps:

1. **Pair creation:** Pairs of groups are created according to the selected windowing system (see above). Let k be the number of pairs.
2. **Statistical testing:** For each pair of groups, compute a selected statistical test for each gene. Let n be the number of genes.
3. **Inner correction:** For each gene, correct the k p-values using the selected p-value correction method. This corrects for multiple testing due to the number of pairs, k.
4. **Aggregation:** For each gene, aggregate the k (corrected) p-values using a selected summarization method, obtaining one aggregated p-value for each gene. The most obvious choice here is to report the *minimum* of the k values, as it describes the probability of the gene's expression profile showing 'differential' behaviour *in any pair.*
5. **Outer correction:** For each pair of groups, correct the p-values obtained in step 4 using the selected p-value correction method. This corrects for multiple testing due to the number of genes, n.

Based on MAYDAY's plugin system, the methods applied during this process can be selected from all available pairwise statistical tests (step 2; e.g., t test, SAM, Rank Product, ...), p-value correction methods (steps 3 and 4; e.g., None, Bonferroni, Šidák, Holm, FDR, ...) and summarization methods (step 5; e.g., geometric mean, median, harmonic mean, min, max), respectively.

For illustration, the time-series statistics plugin was applied to the *S. coelicolor* time-series data introduced in chapter 3.7. The 'adjacent windows' method was chosen with both window sizes set to three, to find genes which show a significant switching event at some (unknown) point in time, regardless of their behaviour over the rest of the time-series (as illustrated in figure 4.8C). The Rank Product method was used for statistical testing (with 100 permutations), Holm's p-value correction method was chosen for both inner and outer correction, and the minimum was used as aggregation function. As input, the set of 7839 protein-coding genes of *S. coelicolor* was used, of which 170 were reported to be statistically significant ($p < 0.05$). These were taken as input for a QT clustering (diameter 0.35, Pearson correlation distance, minimal cluster size four) which resulted in nicely defined clusters revealing a clear cascade of switching events (shown in figure 4.9).

These 170 genes could also be found using the 'separator' method or the 'free windows' method. However, the genes called significant by the 'free windows' method are a superset of the genes found by the 'adjacent windows' method, as this method reports additional genes with more 'blurred' switching events, which is not what was desired in this example. Furthermore, it requires considerably longer computing time due to the larger number of pairs to check. The 'separator' method runs much faster but it assigns statistical significance to all tested genes because the effect of

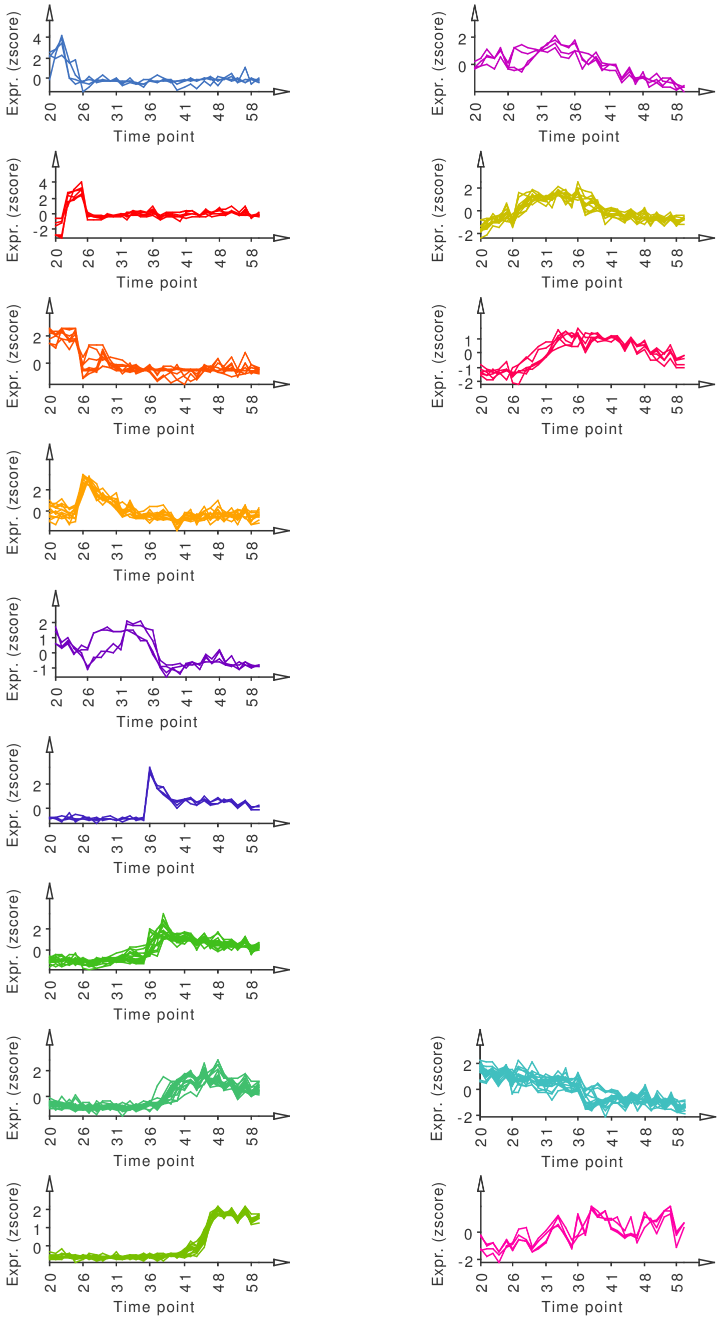


Figure 4.9.: Example application of the time-series statistics plugin. Expression profiles of 7839 protein-coding genes in *S. coelicolor* were tested using the 'adjacent windows' method (both window sizes set to three, see text for details) using the Rank Product (100 permutations) as statistical test and Holm's procedure as inner and outer p-value correction. Probes called significant at $p < 0.05$ were clustered using the QT clustering method (diameter 0.35, Pearson correlation distance, minimal cluster size set to four) and manually sorted to show the cascade of clear switching events over time (left). Less pronounced events are also detected (top right), as are general trends in the data (bottom right). These are also valid results in this application, as the corresponding profiles show at least one marked switch overlaying the general trend. Data from [111].

nutrient limitation is visible in the vast majority of genes as a gradual downregulation. Thus these methods are clearly not applicable to the given example data, but can be the methods of choice for other datasets where different biological phenomena are studied.

4.4.2. Visual analysis of multiple time-series

The Time-series Alignment Analysis plugin Tiala was developed to offer a visual analytics approach to finding significant differences between time-series studied under different conditions. The original Tiala implementation only allowed for pairwise comparisons of time-series. It was later extended by Günter Jäger to support multiple time-series comparisons, including the computation of multiple time-series alignments and 3D visualizations [79].

The basic idea of Tiala is to *align* two time-series such that each experiment E_{1i} of the first time-series is mapped to at most one experiment E_{2j} $(1 \leq i \leq p_1, 1 \leq j \leq p_2)$ of the second time-series (and vice-versa). This alignment is computed based on two important assumptions:

1. **Time-points are expressed as integer values.** Biological experiments are performed by human researchers, who can only work with a certain level of precision when sampling is concerned. Furthermore, while biological conditions might be changing quickly, they are not expected to change greatly in sub-second intervals. In fact, RNA polymerase ii has been shown to create transcripts at about 30 nucleotides per second [26]. Even studies trying to investigate the order in which transcripts are produced (e.g., to gain insight into regulatory cascades) do not require sub-second precision – and if they did, time-points could still be expressed as integer values, measured in milliseconds.
2. **Intervals between time-points are correct.** When sampling time-points in two separate conditions, there might be a deviation from the perfect sampling time, e.g., owing to too few staff members who can extract and process (purify, freeze) the samples. However, these deviations will be at least one order of magnitude smaller than the interval between two consecutive time-points. Thus, Tiala does not attempt to change the time-point designations to create a 'better' alignment. Trivially, this includes that the order of time-points is not changed during the alignment. Manual corrections of the time-point labels attached to the experiments are still possible, of course.

As a result of these two assumptions, the alignment consists of finding the optimal global shift between the two time-series, i.e., finding a δ such that $t_{1i} + \delta = t_{2j}$ for all matched experiments i, j from the two time-series. The idea is that finding the correct global shift to align two time-series will result in lower average distances between their expression profiles, as the majority of genes will not be differentially expressed between the time-series. To find this shift, Tiala offers several statistical and distance-based methods. Some experiments might have to be dropped from the alignment, either because, after shifting, they fall into a time interval not covered by the other time-series (i.e., they were measured before resp. after any experiment in the other time-series), or because they occurred at time-points that were not

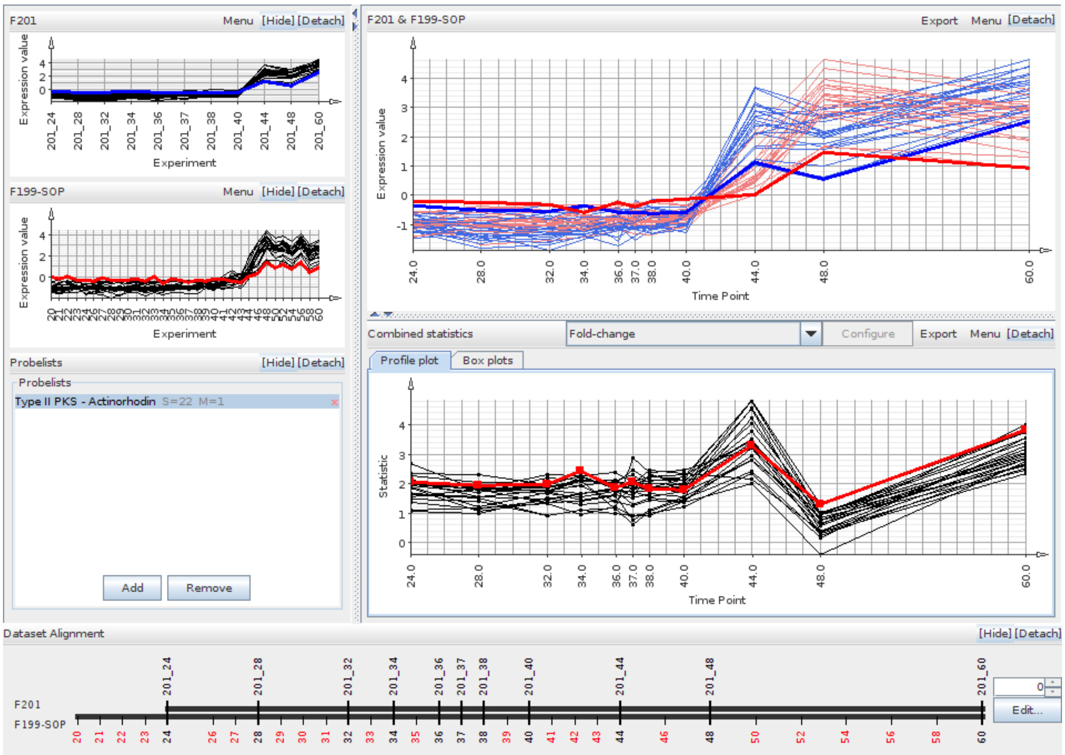


Figure 4.10.: TIALA user interface: Two time-series with different sampling intervals are aligned. Here the alignment contains no shift (bottom). Profiles are shown for the PKS cluster in *Streptomyces coelicolor* for the unaligned datasets (left) as well as after alignment (top right) together with fold-change profiles (center right). One gene is selected (red, blue profiles).

analyzed in the other series. An example for the latter would be time-series with different resolutions on the time axis due to different depth of sampling (e.g., one hour versus two hours resolution).

The resulting alignment is visualized firstly as two parallel axes showing the aligned and unaligned time-points, and secondly by assigning a distinct color to each of the two time-series and drawing the aligned profiles in a common profile plot using these colors. The user interface (see figure 4.10) includes the original, unaligned data sets as well as a plot showing the result of computing a user-selected statistical measure on the aligned profiles. All components of the view can be detached (shown in separate windows), to allow users to focus on certain aspects and make use of modern multi-screen workstations. Tight integration with MAYDAY's visualization framework furthermore enables users to visualize the aligned, unaligned and statistical data in all other visualizations available in MAYDAY, as well as to share selections with other visualizers, etc.

Günter Jäger extended the basic TIALA approach to analyses of more than two time-series [79]. The starting point is a multiple time-series alignment, where one sequence is arbitrarily chosen as *center* series to which all others are aligned in a pairwise manner and time-points that could be aligned in all time-series are chosen for visualization. The resulting set of time-points is independent of the choice of the center sequence. The user interface is similar to that for the two-dimensional case (see figure 4.11) but uses three-dimensional plots for the aligned profiles and multiple views for pairwise statistical comparisons.

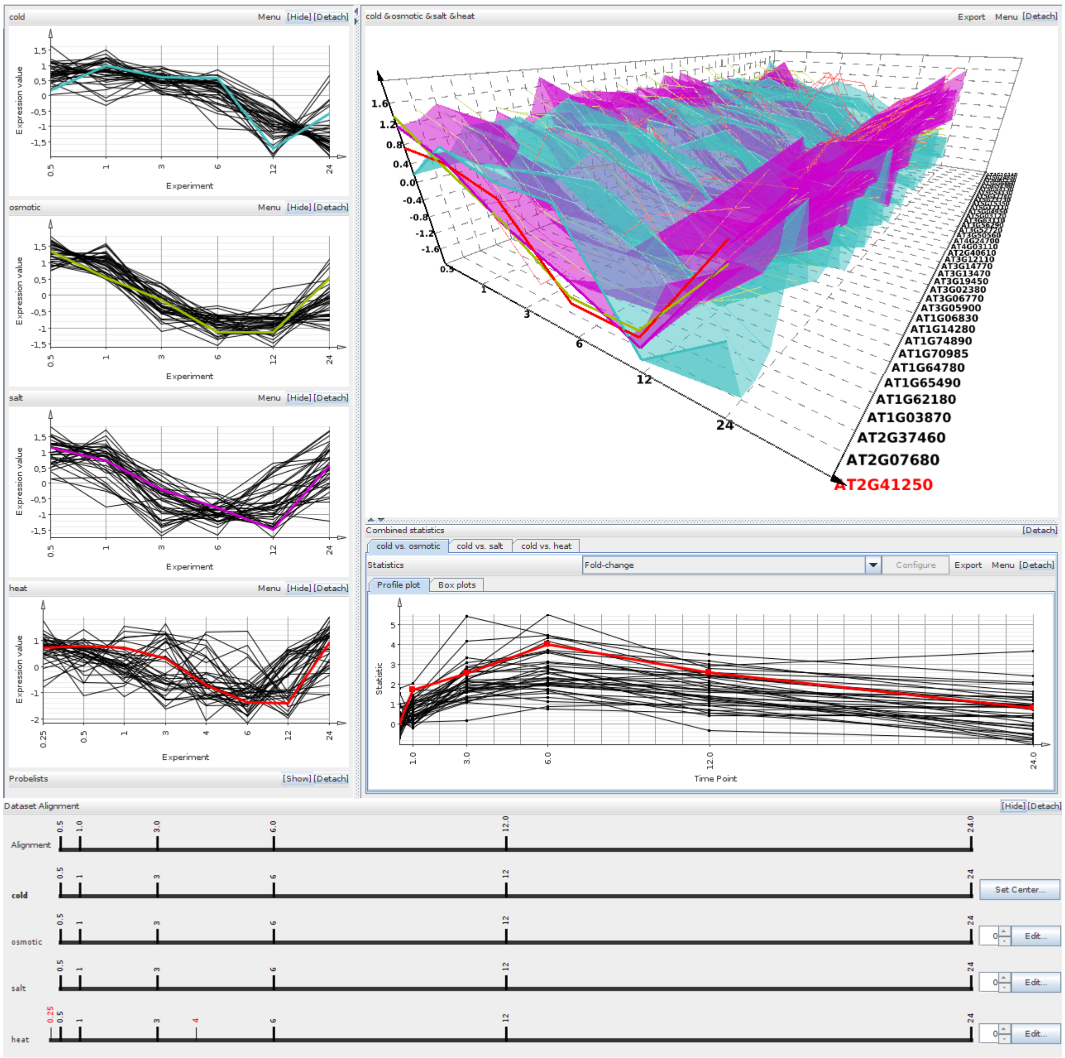


Figure 4.11.: TIALA user interface for multiple time-series comparisons. Here, four datasets are compared. The view layout is similar to the pairwise alignment case (figure 4.10), showing the multiple alignment (bottom), the profile plots for the unaligned datasets (left), and the statistics view (center right). The aligned profiles are shown in a three dimensional view (top right). The third dimension (z axis) is used to separate individual probes, distinct colors are used to represent each dataset and transparent expression surfaces are drawn to facilitate interpretation. Data from [152], figure modified from [79].

5. Integration of scripting languages

Bioinformatics analyses often require the application of a standard set of methods (sometimes laid down in *'standard operating procedures', SOPs*). Mayday offers a large range of methods and visualizations which can be combined in a very flexible manner to conduct analyses. Sometimes, however, analyses require a specific approach that is not covered by the available methods. Scripting languages allow bioinformaticians to quickly test ideas. Rather than implementing a new Mayday plugin for such a one-time occasion, an analyst might want to quickly test some ideas by *interactively* programming with the data. Another use case for interactive scripting arises during the development of a new method, where ideas are first tested in a 'quick and dirty' interactive session before a clean implementation as a Mayday plugin is created.

There exist several programming resp. scripting languages suitable for interactive data analysis, each with its own benefits and disadvantages. However, a user interface integrating such languages into Mayday always contains the same basic set of functionalities irrespective of the language used. The specific implementation of these might differ between languages, for instance because different languages use different *keywords*, but the basic concepts remain the same. The next section will introduce Mayday's generic framework for the integration of interactive programming. It is followed by three concrete examples of languages already integrated.

5.1. An interactive programming environment

The 'Mayday Universal Shell' (MUShell) package defines interfaces and abstract classes for a user interface for interactive programming along with basic implementations that can be used when specialized implementations are not needed. The package consists of the following parts:

- A **tokenizer** which splits any given string into so-called *tokens*, each of which is part of a certain *token class*. Several classes are predefined (see table 5.1) but this list can be extended if a specific language defines additional classes.
- A **syntax highlighter** that can be used to add color to a text string based on its decomposition into tokens, both for static (display-only) content as well as for editable GUI elements.
- **Auto-completion** that can support the user in programming either based on a dictionary of known keywords, or context-aware. The latter approach can, for example, include the names of currently visible variables in the dictionary, or restrict the set of completion suggestions to objects (resp. tokens) that are applicable at the current position in the source code (e.g., the names of member functions of an object).

- A **command queue and history** that stores commands entered but not yet evaluated as well as commands already processed. The history can be browsed, stored, loaded and executed again.
- So-called **code snippet elements**. In the most basic form, a code snippet is a small piece of code that can be inserted at the current position. More sophisticated implementations provide the user with a list of visible objects in the current scope (e.g., the current *environment* in `R`) or with a browsable list of objects and their members (as implemented for the JavaScript console, see 5.3).
- The **script engine interface** that links the input and output windows, the command queue and the command history with the actual script engine. Commands are sent to the engine for execution, resulting output is displayed to the user. The engine's current state ('busy' or 'idle') is queried to determine whether a command can be executed right away or needs to be put into the queue. The engine interface also supplies the auto-completer with information about available objects etc.
- A **user interface** bringing all these elements together in a combined view, as shown in figure 5.1.

Based on the MUShell package, plugins for interactive data analysis were implemented for three languages: The statistical computing language `R` [125] (section 5.2), the general-purpose scripting language JavaScript (section 5.3) and the database query language SQL (section 5.4).

5.2. Joining the powers of R and Mayday: RLink

5.2.1. R and Mayday

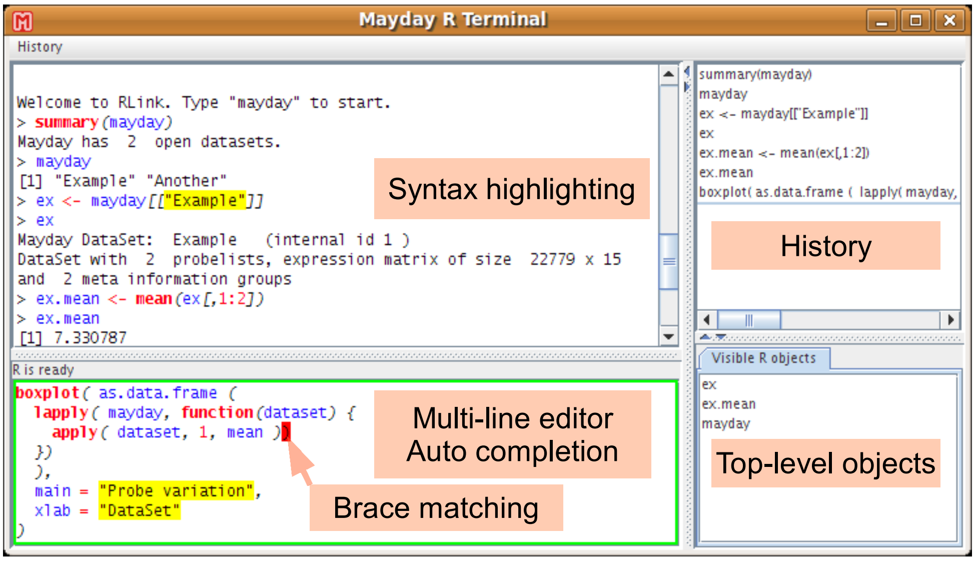
`R` is a programming language and an environment for statistical computing. The language has a large set of mathematical functions and many constructs for vector and matrix operations. The latter allow users to quickly specify complex computations, such as computing the element-wise logarithm of a large matrix and subtracting a (column) vector from each column of the matrix before finding all matrix rows which contain at least one element larger than some value x. Data can quickly be visualized (e.g., in histograms or scatter plots) and visualizations can be customized by a myriad of parameters.

`R` supports object-oriented programming (using so-called S3 and S4 classes) and has a sophisticated system for call parameter evaluation and variable scoping [168]. Most importantly, `R`'s basic functionality can be extended by installing *packages*, most of which are available through a central repository, the comprehensive `R` archive network *CRAN*. Such packages can contain `R` code as well as native `C` or `Fortran` code (using functions provided by the native `R` library) for optimized computations.

`R` has become a standard tool for bioinformaticians, and many packages specific for bioinformatics tasks have been developed which are collected by the *bioconductor* project [58]. New algorithms for bioinformatics problems are often developed and implemented as `R` packages before becoming available in other forms. This, together

Table 5.1.: Predefined MUSHELL token types

WHITESPACE	spaces, tabulators, line breaks
PUNCTUATION	comma, semicolon, etc., including matching braces
OPERATOR	the common one-character operators (e.g., +, −)
TYPE	data types (for typed languages)
OBJECT	names of visible objects (e.g., user-defined variables)
COMMAND	function calls (usually of the form `<Identifier>( )`
NUMBER	numeric literals in decimal or hexadecimal notation
STRING	string literals enclosed by (non-escaped) delimiters
COMMENT	including one-line and multi-line comments
ERROR_TOKEN	Something that is clearly wrong at its position, including non-matching braces, brackets, and parentheses
TEXT	anything not classified as another token type



```
Welcome to RLink. Type "mayday" to start.
> summary(mayday)
Mayday has  2  open datasets.
> mayday
[1] "Example" "Another"
> ex <- mayday[["Example"]]
> ex
Mayday DataSet:  Example   (internal id 1 )
DataSet with  2  probelists, expression matrix of size  22779 x 15
and  2 meta information groups
> ex.mean <- mean(ex[,1:2])
> ex.mean
[1] 7.330787
```

```
summary(mayday)
mayday
ex <- mayday[["Example"]]
ex
ex.mean <- mean(ex[,1:2])
ex.mean
boxplot( as.data.frame ( lapply( mayday,
```

```
boxplot( as.data.frame (
  lapply( mayday, function(dataset) {
    apply( dataset, 1, mean )
  })
  ),
  main = "Probe variation",
  xlab = "DataSet"
)
```

```
ex
ex.mean
mayday
```

Figure 5.1.: MUSHELL user interface for interactive programming showing the RLINK console. A multi-line editor with syntax highlighting (bottom left) can be used to enter commands which are placed in the command queue and evaluated. Results are shown in the output window (top left). Past commands are collected in the command history (top right). A list of top-level objects is shown in the lower right corner as an example of a '*code snippet*' element.

with R's powerful scripting language and the powerful interactive shell are the cause for R's popularity with bioinformaticians.

As a GUI application, Mayday has some advantages over R with respect to user-friendliness and interactivity of the visualizations. However, due to the much larger user base of R, complicated new analysis methods are usually not implemented within Mayday at first. Even if they are implemented eventually, they will not be available in Mayday as quickly as in R. Furthermore, while Mayday's plots offer many configurable properties, they are not as flexible as the programmable (though non-interactive) R plots. Thus it may be helpful for analysts if the strengths of both programs were combined.

5.2.2. The RInterpreter plugin

In 2004, Matthias Zschunke developed a plugin [186] to connect Mayday to R. This implementation was based on converting Mayday data structures to script files that, when processed by R, recreated the data structures as R objects. Then, R executed a user-defined script to process these objects and the resulting data (meta-information, new datasets, etc.) were again written to a file in a proprietary format. Finally, these files were parsed by the Mayday plugin and new Mayday data structures were created. This plugin created a first connection between Mayday and R, yet it had a few shortcomings:

- **No interactivity** – R functions had to be prepared in advance together with XML files describing the parameter values Mayday should pass to R. Interactive analysis, one of the core strengths of R, was not possible. Furthermore, debugging the scripts was not possible in an interactive fashion, as the complex R objects created by Mayday were not available for manual inspection in R.
- **Extensive serialization** – Every call to an R script resulted in the complete Mayday dataset being serialized to files and deserialized into R objects, even if only a single value was accessed by the user's script.
- **Hard to maintain** – To create intermediate files, the plugin required write access to the file system. Furthermore, the call to the R binary differs depending on the operating system. For instance, finding the R installation path on Microsoft Windows systems is only possible by accessing the system registry. This, together with changes between different R versions made the RInterpreter plugin hard to maintain.

5.2.3. RLink: Aims and foundations

To address the problems identified with the RInterpreter approach, a new connection between Mayday and R was developed. Its aims were to

- provide a **feature rich, interactive R shell** inside Mayday,
- give R users **live access** to Mayday's data structures while protecting them against illegal changes,
- share objects between Mayday and R in an **efficient** manner and avoid unnecessary copying of data, and to

- facilitate the use of Mayday objects in R by hiding the complexity of the Java–R transition and providing experienced R users with **familiar constructs**.

Mayday RLink, released in 2009 and presented at the R user conference “useR!2009”, was developed to complement and eventually replace the old RInterpreter plugin. Different approaches for connecting R to Mayday (as a Java-based application) were considered:

Firstly, a separate R instance could be started and data could be transferred by redirecting the input and output streams of R. This would allow users to interactively control R from a Mayday shell window, but live access to Mayday’s objects would be extremely difficult to implement, and impossible to implement efficiently.

Secondly, a networking connection could be established based on RServe [163]. Again, data exchange would be problematic and the burden of the implementation would be on the Java side. An interactive shell would be very challenging if not impossible to implement as RServe was designed to allow Java programs to call R functions. RLink requires the opposite direction, the aim is to call Mayday’s functions from within an R shell.

Java: *Extracting one element from a* `Map<String,Integer>`
Note that Java automatically ‘unboxes’ the `Integer` *object to an* `int` *value.*

```
int ret = myMap.get(”Key”);
```

rJava:
1. Creating a Java `String` *from the* `R` *character vector*

```
keyS <- .jnew( ”Ljava/lang/String;”, ”Key” );
```

2. Casting the Java `String` *to a Java* `Object`
to match the map’s `get` *function argument type.*

```
keyO <- .jcast(keyS, ”Ljava/lang/Object”);
```

3. Calling the map’s fully-specified `get` *function*
to retrieve the mapped-to `Object`

```
retO <- .jcall( myMap, ”Ljava/lang/Object;”, ”get”, keyO );
```

4. Casting the `Object` *to* `Integer`

```
retI <- .jcast( retO, ”Ljava/lang/Integer” );
```

5. Calling the `Integer`*’s fully-specified* `intValue` *function*
to extract the native `int` *value*

```
ret <- .jcall( retI, ”I”, ”intValue” );
```

RLink: *Using* `R`*’s named list semantics to represent the map*

```
ret <- map[[”Key”]]
```

Listing 5.1: Example of an RLink operator: Accessing an element of type `Integer` by its key (of type `String`) in a map. Top, original Java code; Middle, `rJava` code necessary to retrieve the element; Bottom, simplified code using native R semantics as provided by RLink.

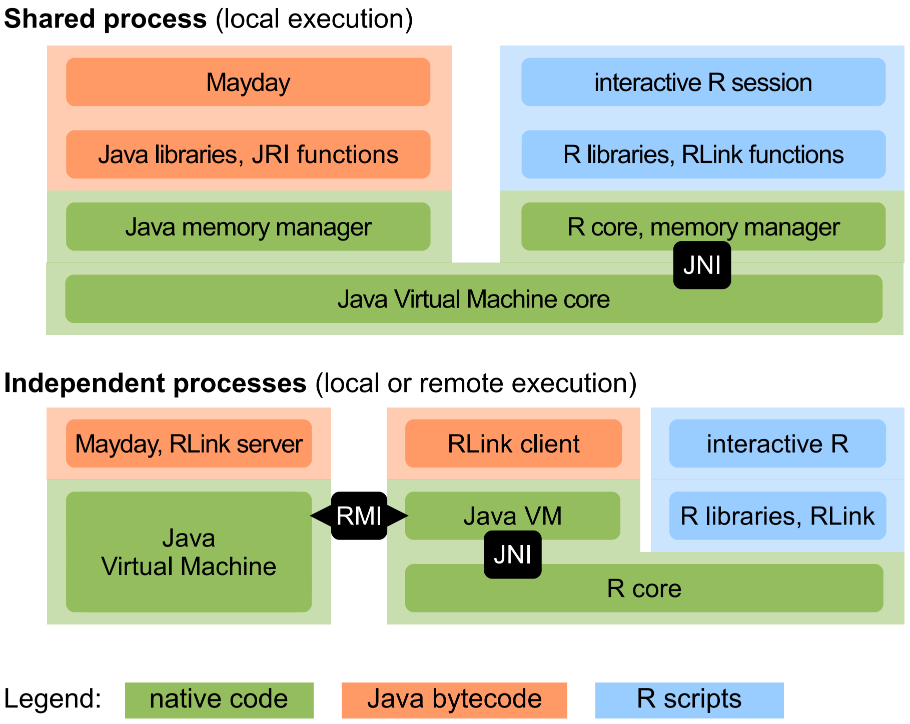


Figure 5.2.: RLink process architecture. Different architectures are used depending on whether the R session is directly embedded in Mayday's virtual machine, or whether a separate process is connected via Java's remote method invocation (RMI) protocol. As RMI is network-transparent, the second option also allows remote connections.

The third option, which was chosen for RLink, is to embed an R process into the Java Virtual Machine using the Java native interface (JNI) which enables Java programs to call functions implemented in other languages based on loading dynamically linked libraries ("shared objects"). On the Java side, the JRI package (now part of rJava [164]) implements the necessary linking code. Then, on the R side, the rJava package is used to call Java functions directly from R code. On top of rJava, RLink implements a layer of R functions that hide the complexity of calling functions across the language barrier (as shown in listing 5.1) and implement data access (see figure 5.2).

RLink consists of three major components:

1. A wrapper around R to serve as script engine (in two different implementations, see below).
2. The **user interface** based on the MUShell package, offering R specific syntax highlighting, and a code snippet element based on the current R *environment* which lists all variables and functions in the current scope.
3. A two-part communication layer: Safe access to Mayday's objects is implemented in Java. Operator overloading is implemented in R to hide rJava's complexity. The R part is also responsible for encapsulating Mayday's objects in R adapters. More details on object encapsulation are given in the next section.

5.2.4. Object encapsulation

To provide efficient and safe access to Mayday's data structures from within R in a manner that is easily usable for programmers familiar with R's syntax, is not straightforward. Firstly, Mayday and R do not share a common area of memory:

Mayday's objects are part of the Java virtual machine's object heap, are managed by the JVM's memory manager and subject to removal by the Java Garbage Collector (GC). When R is embedded in the JVM as described above, it shares the JVM's memory. R implements its own memory management and garbage collection facilities. As a result, the JVM's allocated memory is dominated by two (potentially) very large objects: The Java object heap and the R object heap. (The precise distribution of object heaps to allocated memory may differ, e.g., a typical JVM usually keeps several so-called 'generations' of object heaps.)

Both R and Java hide the 'true' location and nature of user's objects, i.e., users can neither determine the memory address that their object resides at nor the *layout* (the order of the object's bytes in memory). JNI/JRI solve this problem and allow objects to be visible on both sides. However, granting direct access to Mayday's objects from R scripts is dangerous as many of their functions are not meant to be called directly by users and could bring the dataset into an unexpected state.

In addition, if the complete object is exported from Mayday to R in this manner, the alternative mode of operation for network-transparent access (described in the next section) would require that each Mayday object type that could conceivably be used in R (including types referenced from the main data types) implement the RMI `Remote` interface which is clearly infeasible, considering the size of the Mayday project.

As a result of these considerations, RLink was implemented in such a way that only a single object (implementing `Remote`) is exported via JNI and all access is channelled through methods of that object. Only Mayday objects of a small set of types (`DataSet`, `ProbeList`, `Probe`, `MIGroup`, `MIType`, `MIManager`) are accessible from R. Upon first access from R, a Mayday object is associated with a unique numerical identifier which is encapsulated in an R object. This object implements an S3 class such that R operators can be overloaded with specific implementations suitable for the object's type.

Operator overloading is used to implement R semantics on the encapsulated objects. For example, the indexing operators (`[]`, `[[]]`) are used to retrieve and modify values contained in a `DataSet`'s expression matrix. Thus, a Mayday `DataSet` is encapsulated in R as a numeric value (the object's identifier) with attached `class` 'DataSet'. Essential R methods such as `sapply`, `lapply`, `print`, `rownames`, etc. are overloaded with implementations specific for `DataSets` and the indexing operators can be used to access `ProbeLists` (using list semantics), `Probes` (using vector semantics), as well as any element of the expression matrix (using matrix semantics).

The overloaded operators mask the true nature of the encapsulated object such that most R functions can be used directly on Mayday's object without the need to copy for example the whole expression matrix from Mayday to R. For cases where this is not sufficient, or where the programmer actually wants to work on a copy of the data,

Table 5.2.: RLink modes of operation

Mode:	Shared process	RMI connection
Number of R instances	1	no limit
Memory limit (32 bit)	3 GB for JVM and R	3 GB for each process
R process location	local	local or remote
Installation	more complex	easier
Connecting R and Mayday	simple	more complex
Speed of data transfer	very high	depending on connection

a special use of the matrix indexing operator `[]` without index (e.g., `m<-object[]`) has been added. For `DataSets`, for example, this copies the whole expression matrix into an R numerical matrix, an operation which can be interpreted as dereferencing the pointer which the object represents.

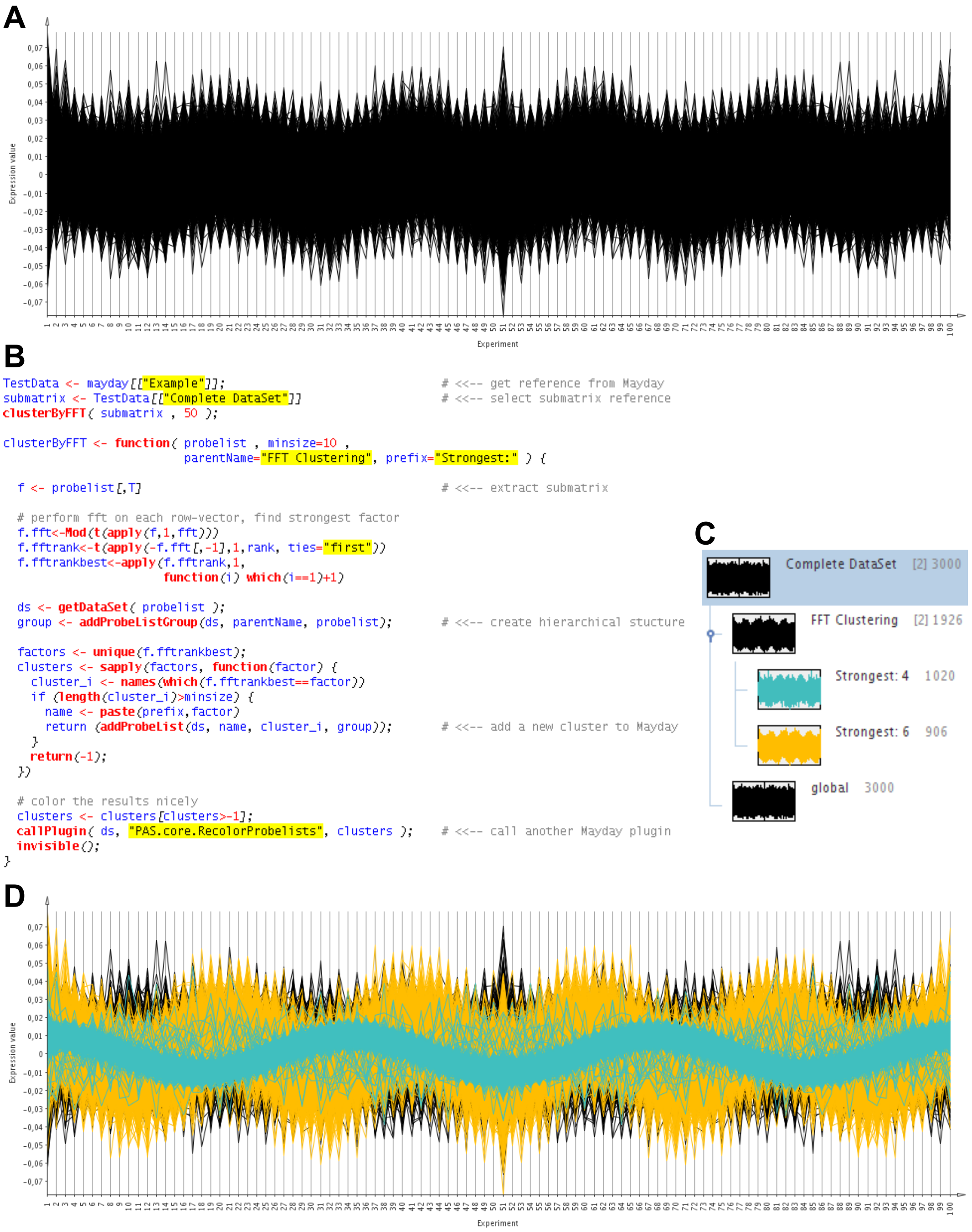
In addition to accessing Mayday's data structures, RLink also allows users to embed Mayday plugins into their R scripts, and to register R functions as plugins in Mayday (e.g., to implement a distance measure). This bi-directional integration can be used very effectively to solve problems that Mayday alone may not be able to solve easily. For an example, consider a data set of time-series expression data. A user would like to cluster genes with oscillating expression profiles according to the strongest frequency component, which is not possible in Mayday. Using the FFT transformation in R, this can be done with a few lines of R code (see figure 5.3).

5.2.5. Adding network transparency

The RLink implementation discussed so far has a major drawback, which results from the direct embedding of the R runtime into the Java Virtual Machine's process: R is not running as a separate process but as part of the JVM process. As such, both 'partners' share a common memory area. On 32 bit systems, each process can use a maximum of 4 GB of memory (in theory; in practice it is around 3 GB per process as some part of the address space is reserved for the operating system kernel). For large data sets, this can be a problem, especially when performing demanding computations in R. A second problem is that a bug in an R package can crash R and, by consequence, also crash the JVM.

By decoupling the two parts and running R in a separate process, both problems are solved. For RLink, this was achieved by adding an alternative script engine implementation, where the RLink R part is run in a separate R process, connecting to Mayday via Java's *Remote Method Invocation* (RMI) protocol. Thus, instead of running R inside Mayday's JVM, a tiny JVM used only for data transmission is embedded into the separate R process (figure 5.2).

This solution has three additional benefits: Firstly, RMI is network transparent by design. Thus, it does not matter where the separate R process is running. It could be on the same workstation, or on a high-performance computing system somewhere on



```r
TestData <- mayday[["Example"]];                              # <<-- get reference from Mayday
submatrix <- TestData[["Complete DataSet"]]                   # <<-- select submatrix reference
clusterByFFT( submatrix , 50 );

clusterByFFT <- function( probelist , minsize=10 ,
                          parentName="FFT Clustering", prefix="Strongest:" ) {

  f <- probelist[,T]                                          # <<-- extract submatrix

  # perform fft on each row-vector, find strongest factor
  f.fft<-Mod(t(apply(f,1,fft)))
  f.fftrank<-t(apply(-f.fft[,-1],1,rank, ties="first"))
  f.fftrankbest<-apply(f.fftrank,1,
                       function(i) which(i==1)+1)

  ds <- getDataSet( probelist );
  group <- addProbeListGroup(ds, parentName, probelist);      # <<-- create hierarchical stucture

  factors <- unique(f.fftrankbest);
  clusters <- sapply(factors, function(factor) {
    cluster_i <- names(which(f.fftrankbest==factor))
    if (length(cluster_i)>minsize) {
      name <- paste(prefix,factor)
      return (addProbeList(ds, name, cluster_i, group));      # <<-- add a new cluster to Mayday
    }
    return(-1);
  })

  # color the results nicely
  clusters <- clusters[clusters>-1];
  callPlugin( ds, "PAS.core.RecolorProbelists", clusters );   # <<-- call another Mayday plugin
  invisible();
}
```

Figure 5.3.: FFT clustering performed using RLINK. A, expression profile plot of 3000 simulated gene expression profiles over 100 experiments (time-points). 1000 profiles oscillate at a frequency of 0.04 (4 cycles over 100 experiments), 1000 at a frequency of 0.06 (6 cycles), 1000 oscillate randomly. Normally distributed noise was added to all profiles. B, RLINK function for clustering based on R's FFT transformation with syntax highlighting by MAYDAY's shell. C, resulting `ProbeLists` in MAYDAY. D, profile plot colored by cluster membership.

the network. Secondly, there is no limit to the number of `R` processes concurrently connecting to Mayday. This can be used to allow a simple form of collaboration (see also chapter 6), where several users connect to a single Mayday 'server' instance to work on the same data. Thirdly, as there is no direct (byte-code) linking between the `R` library and the Java Virtual Machine, this scenario is easier to set up from the end-user's point of view. A comparison of both modes of operation is given in table 5.2.

5.3. On-the-fly scripting: JavaScript console

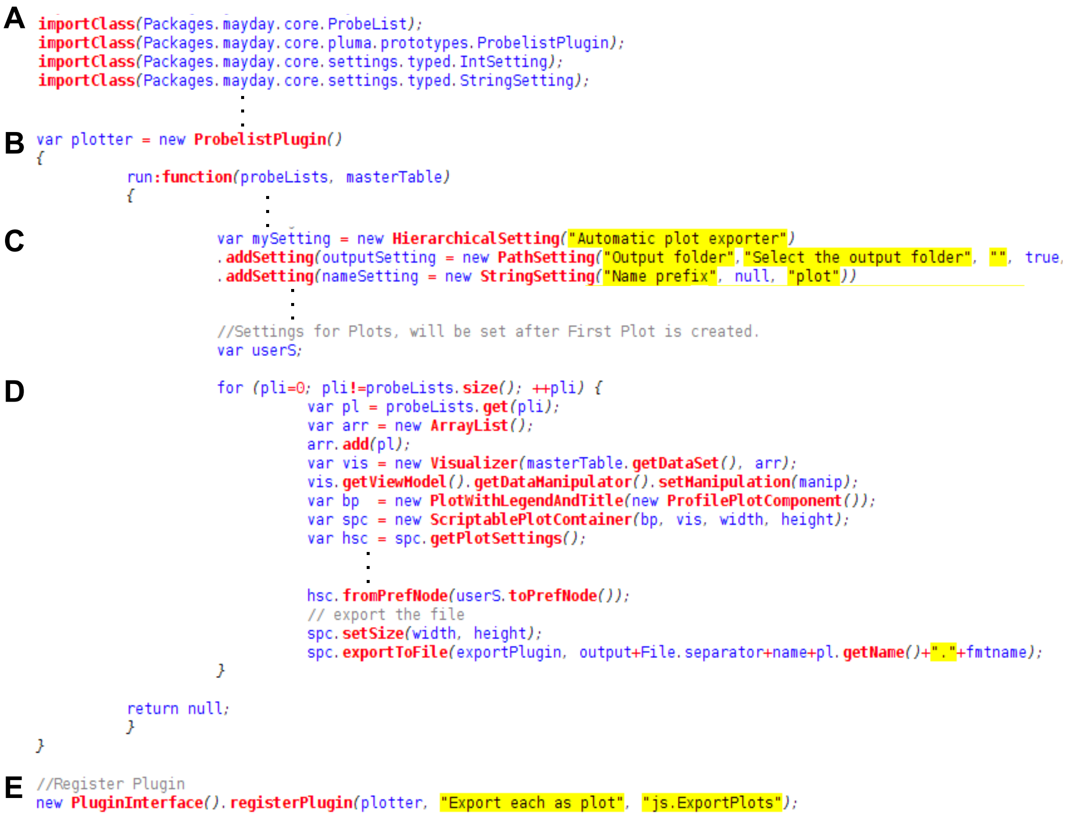
The RLink plugin can be used both for interactive analyses as well as for rapid prototyping of new analysis methods. However, RLink is not part of the default Mayday distribution due to the requirement that native code needs to be compiled on the user's workstation. This motivates the inclusion of another scripting language which can be distributed directly with Mayday. JavaScript is an obvious choice for such a language: On the one hand, JavaScript execution has been part of the Java specification since Java version 6 [83], requiring no additional or native libraries. On the other hand, programmers used to writing Java programs and plugins for Mayday are already familiar with the language as Java and JavaScript use very similar constructs.

The JavaScript console implemented during Tobias Ries' bachelor thesis [128] extends the MUShell framework. Its most important parts are a highly complex context-sensitive auto-completer as well as type-sensitive operator definitions. Further improvements are code snippet elements for quick access to Mayday's objects and to a list of visible objects (including their type and content), automatic code loading and a more comfortable editor window.

The auto-completer uses Java's `Reflection` API to inspect live objects *inside* the script engine, gain information about their type and create a list of available members of complex objects. Furthermore, sophisticated code analyses are performed to deduce the content types of Java containers (which, at run-time, are basically untyped) taking care not to change any object's state by calling functions which might have unintended side effects.

A flexible method for operator definitions has been implemented that allows users to (persistently) define operators in the form of JavaScript functions such that, for instance, the `[]` operator can be used on a `DataSet` object with a numerical parameter for one purpose, and with a string parameter for a different purpose. Such definitions are possible based on exact type matching as well as on inherited types (similar to the `assignableFrom` function of the `Class` reflection type). Even more far-reaching effects can be attained by using replacement rules based on regular expressions.

The JavaScript engine is coupled much more tightly to Mayday than the `R` core can ever be, mainly because the former is directly supported by Java while the latter had to be attached in a rather more complicated manner (see section 5.2.2). Programmers using JavaScript have direct access to all of Mayday's data structures, user interface elements, and plugins. On the other hand, however, the JavaScript

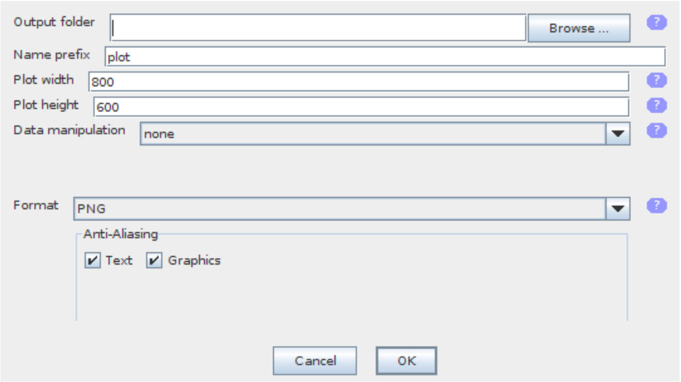


```
importClass(Packages.mayday.core.ProbeList);
importClass(Packages.mayday.core.pluma.prototypes.ProbelistPlugin);
importClass(Packages.mayday.core.settings.typed.IntSetting);
importClass(Packages.mayday.core.settings.typed.StringSetting);
                   ⋮
var plotter = new ProbelistPlugin()
{
          run:function(probeLists, masterTable)
          {                  ⋮
                    var mySetting = new HierarchicalSetting("Automatic plot exporter")
                    .addSetting(outputSetting = new PathSetting("Output folder","Select the output folder", "", true,
                    .addSetting(nameSetting = new StringSetting("Name prefix", null, "plot"))
                          ⋮
                    //Settings for Plots, will be set after First Plot is created.
                    var userS;

                    for (pli=0; pli!=probeLists.size(); ++pli) {
                              var pl = probeLists.get(pli);
                              var arr = new ArrayList();
                              arr.add(pl);
                              var vis = new Visualizer(masterTable.getDataSet(), arr);
                              vis.getViewModel().getDataManipulator().setManipulation(manip);
                              var bp  = new PlotWithLegendAndTitle(new ProfilePlotComponent());
                              var spc = new ScriptablePlotContainer(bp, vis, width, height);
                              var hsc = spc.getPlotSettings();
                                     ⋮
                              hsc.fromPrefNode(userS.toPrefNode());
                              // export the file
                              spc.setSize(width, height);
                              spc.exportToFile(exportPlugin, output+File.separator+name+pl.getName()+"."+fmtname);
                    }

          return null;
          }
}

//Register Plugin
new PluginInterface().registerPlugin(plotter, "Export each as plot", "js.ExportPlots");
```



Output folder

Name prefix plot

Plot width 800

Plot height 600

Data manipulation none

Format PNG

Anti-Aliasing

☑ Text ☑ Graphics

Figure 5.4.: JavaScript example plugin (excerpt). A typical JavaScript plugin for Mayday starts with a section where Java classes are imported (A), followed by the definition of a plugin object (B). When run, the plugin first creates `Settings` (C) to allow users to configure relevant parameters needed for processing the input `ProbeLists`, resulting in an automatically generated user interface (bottom). In this example, Mayday's visualization framework is used to create a profile plot for each `ProbeList` and export it using one of Mayday's image export plugins selected by the user via a `PluginTypeSetting` (D). Finally, the plugin is registered with Mayday's plugin manager (E). The complete code of the example shown here covers 84 lines of JavaScript code.

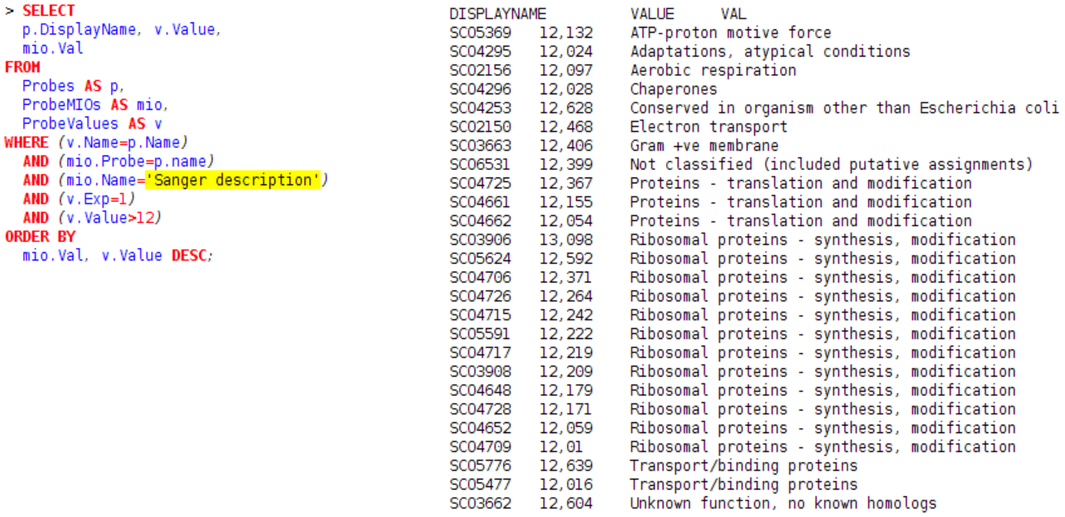
interpreter lacks the elegant mathematical syntax of `R` and its vast library of domain-specific packages. Thus, the typical application domain of JavaScript in Mayday are tasks that can be automated using existing Mayday components, possibly involve some user interaction, but do not require extensive computations. As an example for a JavaScript *plugin* written for Mayday, consider a small program (shown in figure 5.4) which takes the currently selected `ProbeLists`, creates a profile plot for each of them (using user-defined parameters for plotting) and exports these plots to image files (e.g., for later use in a publication). Such a plugin can be implemented in less than 100 lines of JavaScript code due to the strengths of Mayday's visualization and `Settings` frameworks as well as the plugin system.

5.4. Structured queries using SQL

With the dynamic `ProbeList` mechanism (described in section 3.5), Mayday offers a powerful filtering framework. Still, very complex filters might require additional implementation. Stephan Symons implemented an SQL console based on MUShell and the Apache Derby database engine.

Mayday's objects (`DataSets`, `MasterTables`, `ProbeLists`, and meta information groups) are wrapped in adapters that allow the SQL processor to execute queries on them as if they were tables. Queries can be entered in the well-defined SQL syntax, including such powerful operations as *joins*, ordering and sub-queries, and are optimized by the database engine before evaluation. Custom views and new tables can be created to facilitate analyses.

For illustration, consider the following research question: Given a dataset of expression data annotated with functional categories (e.g., the dataset described in section 3.7), find the categories that the genes with the highest expression values in the first experiment belong to and produce a table ordered by (1) the functional category and (2) the expression value, and include the locus tag (probe display name) in the table. To do this using Mayday's user interface, one has to start by creating a dynamic `ProbeList` of all genes with expression (in the first experiment) above a certain threshold, say 12. The meta information attached to the genes in that `ProbeList` can be displayed in a tabular view and sorted according to the functional annotation column. However, a second level of ordering can not be configured. For this, the table has to be exported into a tabular text file, imported into a spreadsheet application and sorted there. Using SQL, the question can be answered using the query shown in figure 5.5.



```
> SELECT
  p.DisplayName, v.Value,
  mio.Val
FROM
  Probes AS p,
  ProbeMIOs AS mio,
  ProbeValues AS v
WHERE (v.Name=p.Name)
  AND (mio.Probe=p.name)
  AND (mio.Name='Sanger description')
  AND (v.Exp=1)
  AND (v.Value>12)
ORDER BY
  mio.Val, v.Value DESC;
```

```
DISPLAYNAME          VALUE     VAL
SC05369    12,132    ATP-proton motive force
SC04295    12,024    Adaptations, atypical conditions
SC02156    12,097    Aerobic respiration
SC04296    12,028    Chaperones
SC04253    12,628    Conserved in organism other than Escherichia coli
SC02150    12,468    Electron transport
SC03663    12,406    Gram +ve membrane
SC06531    12,399    Not classified (included putative assignments)
SC04725    12,367    Proteins - translation and modification
SC04661    12,155    Proteins - translation and modification
SC04662    12,054    Proteins - translation and modification
SC03906    13,098    Ribosomal proteins - synthesis, modification
SC05624    12,592    Ribosomal proteins - synthesis, modification
SC04706    12,371    Ribosomal proteins - synthesis, modification
SC04726    12,264    Ribosomal proteins - synthesis, modification
SC04715    12,242    Ribosomal proteins - synthesis, modification
SC05591    12,222    Ribosomal proteins - synthesis, modification
SC04717    12,219    Ribosomal proteins - synthesis, modification
SC03908    12,209    Ribosomal proteins - synthesis, modification
SC04648    12,179    Ribosomal proteins - synthesis, modification
SC04728    12,171    Ribosomal proteins - synthesis, modification
SC04652    12,059    Ribosomal proteins - synthesis, modification
SC04709    12,01     Ribosomal proteins - synthesis, modification
SC05776    12,639    Transport/binding proteins
SC05477    12,016    Transport/binding proteins
SC03662    12,604    Unknown function, no known homologs
```

Figure 5.5.: SQL example query (left) extracting a list of gene locus tags, expression values in the first experiment and functional annotation ordered first by the annotation and then, descendingly, by the expression value, and restricted to such genes whose expression value is larger than 12. Further SQL operations, such as *aggregation* and *grouping*, as well as *joins* across additional tables can be used to refine the query. The resulting table (right) shows a dominance of ribosomal proteins among the highly expressed genes in a time-series expression dataset in *S. coelicolor* [111].

6. Collaborative analysis

In bioinformatics, one of the main hindrances to efficient analyses are problems of data exchange, owing on the one hand to the huge volume of data produced by modern high-throughput technologies, and, more importantly, to the diversity of data formats. This problem not only manifests itself when researchers from different groups want to exchange data and analyse it collaboratively, but also when a single researcher is working with the data, as different analysis steps (especially during higher-level analysis) require the application of different programs, each with its own demands regarding data input format.

In this chapter, both the problem of data exchange between programs as well as that of collaborative data analysis between different researchers is addressed. Regarding the first problem, integrating as many tools as possible into one comprehensive software package (the Mayday approach) can be helpful as it reduces the number of occurrences of the problem that need to be dealt with. The integration with R further extends the circle of tools available to a researcher using Mayday, but it may still not be enough.

6.1. The Gaggle

The *Gaggle* [143] provides a fundament for further integration. The basic idea is to replace the current system of file system based manual conversion between data formats and analysis programs with an automated approach using inter-application communication (using the RMI protocol, as done by Mayday RLink, see section 5.2) based on a small set of defined interchange objects.

These object types were defined with a view on Systems Biology projects where users mainly deal with tabular numeric data (e.g., expression matrices), lists of object identifiers (e.g., gene names, accession numbers), clustering results, and different kinds of networks (such as protein-protein interaction networks, gene regulation networks, etc.). Gaggle defines five object types to cover most data exchange tasks (see table 6.1 for details):

- **NameList:** A list of object identifiers (strings).
- **Matrix:** A numerical matrix with row and column names.
- **Cluster:** A set of row and column names, used to define a sub-matrix resulting for example from bi-clustering of an expression matrix.
- **Network:** A set of nodes with attributes and connections (of different types, e.g., positive and negative regulation).
- **Tuple:** The most generic data structure, containing a set of (possibly nested) named values, and a name.

Table 6.1.: Gaggle data type grammar

GaggleData	→ Name , Species , Content , Tuple
Content	→ (NameList | Matrix | Cluster | Network | Tuple)
Tuple	→ Name , Single*
Single	→ Name , (Value | Tuple)
Value	→ (`Integer` | `Long` | `Float` | `Double` | `Boolean` | `String` | GaggleData)

Each GaggleData object contains a specific Content depending on its type (NameList, Matrix, Cluster, Network, or Tuple) and an additional Tuple for metadata. The Tuple type allows for nesting of objects.

Each object contains an additional *tuple* instance which can be used to transport meta information regarding the object. Applications supporting the Gaggle interfaces, called *Geese* (sg. *Goose*), send objects of the aforementioned types to the *Gaggle Boss*, a small RMI application which acts as a server that all *Geese* connect to. The Boss broadcasts the object to all attached applications (except for the sender itself).

To integrate an existing application with Gaggle, it needs to support the RMI protocol and define import as well as export conversion methods that convert between the internal object representation of the application and the Gaggle object types. In addition, 'appropriate' handling of incoming data (e.g., display, annotation of existing data, creation of a new dataset, changes to existing visualizations) needs to be implemented.

Currently, more than twenty applications can connect with Gaggle (listed on `http://gaggle.systemsbiology.net/docs/geese/`), among them the Gaggle Genome Browser [8], the MeV tool for microarray analysis [133], and a plugin for the Firefox web browser allowing for database searches [9]. The power of Gaggle is directly linked to the number of programs implementing the Gaggle interfaces.

Unfortunately, the Gaggle system has several fundamental shortcomings:

1. The **dependence on Java RMI** makes integration of programs written in other languages relatively difficult.
2. Gaggle offers only a **narrow set of object types** geared towards matrix-based analyses (matrices, named elements, sub-matrices, etc.). Transcriptomic analyses, for example, rely very much on genomic mapping and transcription is often analyzed in the context of a genomic neighborhood (see also section 4.3). This has become even more important with the introduction of high-throughput sequencing methods. Many data formats for genomic coordinates and annotations are currently in use, differing in the level of detail of the annotation, as well as in the coordinate specification (zero-based vs. one-based base position, end-inclusive vs. end-exclusive intervals) which often hinders data exchange. However, there is no Gaggle object type to represent genomic coordinates of any kind. As the Gaggle specification does not include versioning or a similar system for introducing changes later on, adding a new data type now would break compatibility with existing Geese and changes of the existing (set of) object types are unlikely to ever happen.

3. Gaggle's object types are only defined in terms of the basic data type they contain (numeric data, string data, etc.). There is **no semantic definition** of their content. On the one hand, this makes the approach extremely flexible, as 'anything' can be encapsulated in Gaggle objects (especially in the 'tuple' type). On the other hand, it is up to the user to make sure that data sent from one application can be interpreted by the receiving applications. It is virtually impossible to write an application that receives a Gaggle 'tuple' object and does 'something sensible' with it, without knowing exactly where the object originated. Even in the case of the most simple Gaggle object, which only stores a list of names, the receiving application can only interpret these names within its own scope. For example, if one application sent a list of RefSeq [123] accessions which are received by an application that only 'understands' UniProt [6] identifiers, meaningful data exchange cannot take place. Obviously, this is a general problem in the field of bioinformatics, where several naming systems exist in parallel and an automated mapping solution seems to be in the distant future (even though such systems where already proposed, such as the *Protein Identifier Cross Reference* service [41]).

4. The **complete lack of security** is problematic. If a Gaggle Boss instance is run on a machine that is open to the network (i.e., not behind a restrictive firewall), external users can connect to the running Boss and receive a copy of all data transmitted between the attached applications. Furthermore, the Gaggle Boss also accepts and broadcasts data from Geese that *are not even connected* to the Boss, allowing external users to 'insert' data into a running Gaggle session. This results from the distinction between a 'Gaggle connection' and an RMI connection: A 'Gaggle connection' can contain a large number of RMI connections and is defined by connecting, followed by (several) data exchange events and finally disconnecting. A single RMI connection, in comparison, is equivalent to one remote method call, originating at a Geese application and executed in the context of the Boss instance. From the point of view of the Gaggle user, the new data will seem to appear completely out of the blue.

   This problem is not restricted to cases of deliberate interference with a user's session. For instance, if several users share a computing server and each of them wants to have a Gaggle session on that server, only one Boss instance will function as the RMI host for *both users'* Geese to connect to and data will be exchanged between all connected applications, leading to confusion increasing with the number of simultaneous users.

   When using Gaggle for remote exchange (see section 6.3), the security issue becomes even more problematic, as data is sent unencrypted and unauthenticated and attackers could potentially listen in on the data exchange or even participate by sending data to the respective Gaggle Boss instance.

Finally, Gaggle only addresses the problem of *local* data exchange between several applications running on the same machine. The more general case of several applications running on different, spatially separated machines is not addressed (even though the underlying RMI technology is network transparent). In section 6.3, the *GaggleBridge* program is introduced, which extends the Gaggle functionality to also

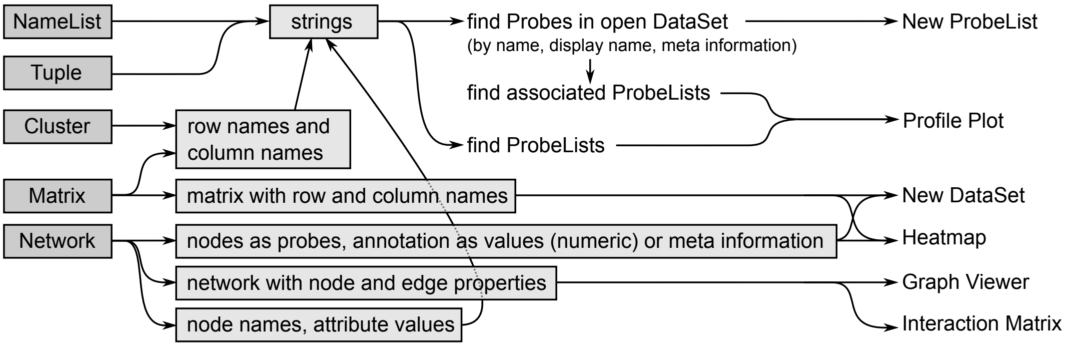

NameList
Tuple
Cluster
Matrix
Network
strings
row names and column names
matrix with row and column names
nodes as probes, annotation as values (numeric) or meta information
network with node and edge properties
node names, attribute values
find Probes in open DataSet
(by name, display name, meta information)
find associated ProbeLists
find ProbeLists
New ProbeList
Profile Plot
New DataSet
Heatmap
Graph Viewer
Interaction Matrix

Figure 6.1.: Gaggle data import. Users can decide how to integrate received Gaggle objects (left) into their Mayday session. Depending on the data type, different options are possible, either resulting in a new `DataSet` or `ProbeList`, or producing an instant visualization.

cover the area of *remote* data exchange. This enables researchers to collaboratively analyse data irrespective of the geographic distance between them (an idea already touched upon in section 5.2, where collaboration was limited to analyses performed in Mayday and `R`). Furthermore, GaggleBridge proposes a method to add encryption and a simple form of authentication to the data exchange.

6.2. Integration of Mayday with the Gaggle

Notwithstanding the shortcomings of Gaggle that were discussed in the previous section, an integration of Mayday with the other *Geese* can simplify complex analyses. In her diploma thesis, Claudia Broelemann [31] implemented the Gaggle interfaces within Mayday both for sending and for receiving data. Her plugin addresses the problem of missing information on the semantics of received data by offering users a choice of options how to continue with the incoming data (see figure 6.1):

- **Names** (from name lists, matrix row/column names, cluster row/column names, network nodes and from tuple string content) can be used to create new `ProbeLists` in existing data sets, either by matching them to `Probe` names or to meta information associated with probes. Alternatively, matching `Probes` can be visualized directly in the context of the `ProbeList` containing them.
- **Numerical Matrices** can be used to create new data sets, which can optionally be visualized as heatmaps. The source can be incoming matrix data, as well as a network object, where any numerical node annotations are interpreted as experiment values.
- **Networks** can be visualized in Mayday's Graph Viewer. Node and edge properties are also imported.
- **Meta-information** can be imported from network node annotations. A hierarchical structure of meta information groups is automatically inferred from the annotation names (column labels).

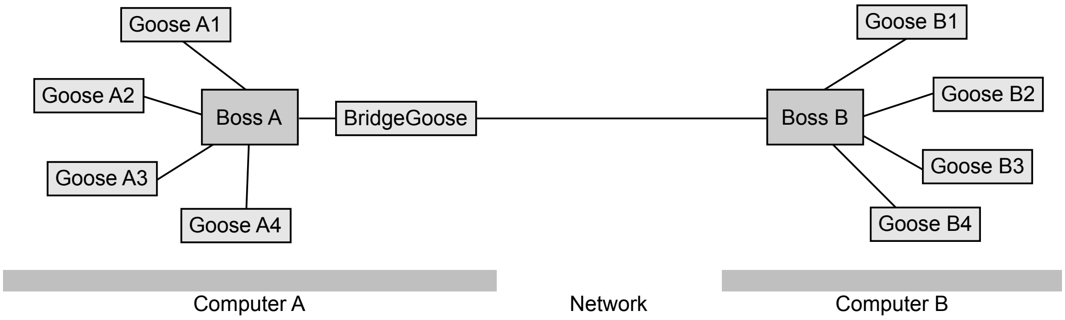


Figure 6.2.: GaggleBridge: Two (or more) Gaggle Boss instances are linked together by a special type of Goose, the BridgeGoose.

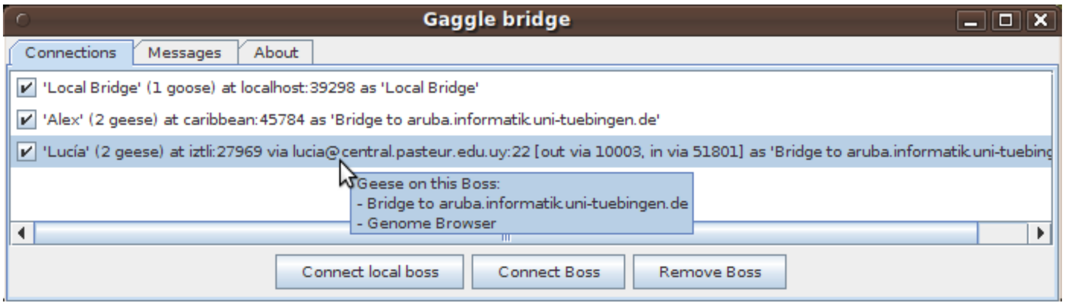
As a sender, MAYDAY can transmit the names and display names of selected `Probe`s or the content of selected meta information objects as NameList, Cluster or Tuple objects. `ProbeList`s and the corresponding submatrices of the expression matrix can be transmitted as Gaggle Matrix object. Networks visualized with the MAYDAY Graph Viewer plugin can also be transmitted via Gaggle.

6.3. Extending Gaggle for collaborative multi-user analyses

The Gaggle approach to inter-application data exchange can be extended to a platform for collaborative analyses: Using RMI, the Gaggle data exchange is network transparent. Basically, a Goose can connect to a Gaggle Boss instance on any other machine, thus linking for example a locally running genome browser with a remotely running microarray analysis software. Here, 'remote' can stretch as far as another continent or as close as two computers in the same room. As a second computer also introduces a second human user, extending the scope of inter-application exchange beyond the local machine transforms Gaggle into a software for collaborative analyses.

Both users can work on the same data in different Geese, or they could be running the same Goose application and see the same view of the same data simultaneously. Exchange between the users' Geese only happens upon user request, allowing each user to browse and work with the data until some elements of interest have been found (e.g., a list of interesting genes) which can then be transmitted to the other use with the click of a button (allowing for network latency when large data is exchanged). The two users can thus collaborate on the analysis of the data set without the need for emailing or up- and downloading data sets. This setup would typically include a telephone or video chat connection for verbal communication.

However, with the original Gaggle Boss program, this is not easily possible. Two reasons can be distinguished: Firstly, most Geese applications automatically connect to a locally running Boss instance and do not allow users to specify alternative, remote Boss addresses. Secondly, network configurations, especially at research institutions, often include restrictive firewalls and NATs (network address translation)



☑ 'Local Bridge' (1 goose) at localhost:39298 as 'Local Bridge'

☑ 'Alex' (2 geese) at caribbean:45784 as 'Bridge to aruba.informatik.uni-tuebingen.de'

☑ 'Lucía' (2 geese) at iztli:27969 via lucia@central.pasteur.edu.uy:22 [out via 10003, in via 51801] as 'Bridge to aruba.informatik.uni-tuebing

Geese on this Boss:
- Bridge to aruba.informatik.uni-tuebingen.de
- Genome Browser

Figure 6.3.: GaggleBridge user interface showing connections to three Boss instances, one to the local Boss, one direct network connection and one connection tunneled through SSH. Information about connection endpoints is included, and the tooltip window shows a list of Geese connected to the respective Boss instance. Part of a figure published in [15].

which prevent direct RMI connections from outside users. The *GaggleBridge* [15] program implements a solution to both problems.

Instead of changing all Geese to allow users to specify other, non-local Boss addresses, GaggleBridge adds another Goose (the 'BridgeGoose') which relays data between two Boss instances (see figure 6.2). If a Goose G_{A1} on computer A sends data to its local Boss B_A, the Boss broadcasts the data to all other local Geese $G_{Ai}(1 < i \leq n)$. One of these, say G_{An}, is a BridgeGoose. This means that G_{An} is connected to B_A locally and to another boss B_B on computer B via a remote connection (as G_{Bm}, say). Upon receiving G_{A1}'s data from B_A, G_{An} transmits the data to B_B. For B_B, the data is coming from one of its connected geese, i.e., B_B makes no distinction between its local Geese $G_{Bj}(1 \leq j < m)$ and the BridgeGoose $G_{Bm} \equiv G_{An}$. The data is broadcast to all Geese G_{Bj} (excepting the 'sender' G_{Bm}).

This approach has several advantages: No existing Goose applications need to be changed, and neither does the Gaggle Boss. As the Gaggle Boss allows users to choose, for each broadcast, which Geese should receive the data, users can selective enable resp. disable the BridgeGoose, thereby either limiting data exchange to all local applications or extending it to a remote user's session. Also, a BridgeGoose can connect to any number of Boss instances, and any number of BridgeGoose instances can be used in parallel, such that the number of users that are connected is virtually unlimited.

The problem of establishing a network connection through firewalls is addressed by the introduction of SSH tunnelling: The BridgeGoose can be configured to establish a secure shell (SSH) connection to a given machine and to tunnel all RMI traffic through the SSH connection. As most research facilities allow SSH connections to pass their firewalls, this circumvents the firewall problem. In addition, it adds encryption, data compression and even a form of authentication (based on the SSH username/password authentication scheme).

GaggleBridge's user interface makes establishing and managing multiple connections as simple as entering the remote machine address and optionally specifying the SSH

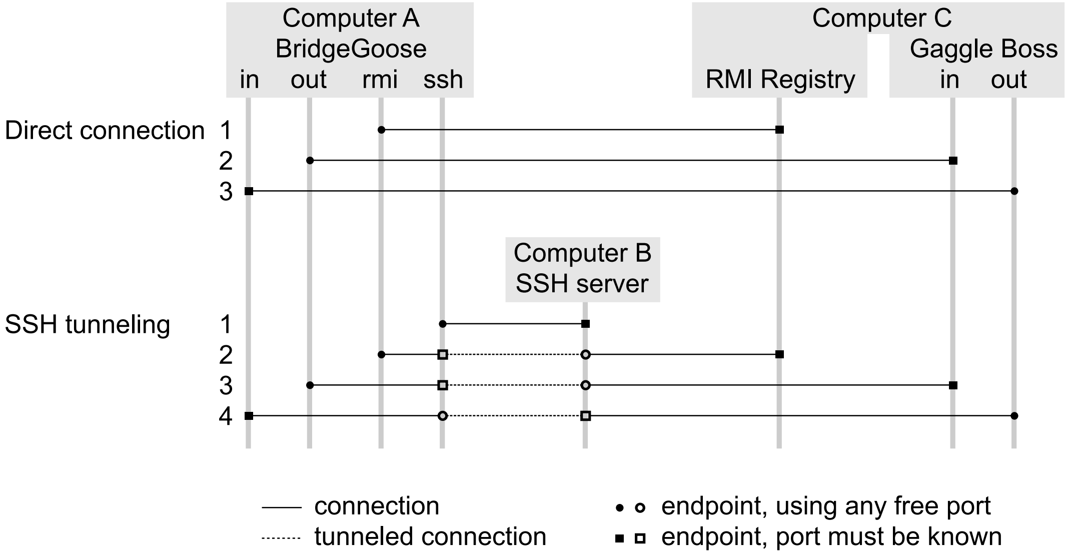


Figure 6.4.: Establishing a GaggleBridge link.
A direct connection (top) is established in three steps: The BridgeGoose on `A` connects to the RMI Registry on `C` (1) to request the Boss address. It creates an outgoing connection to the Boss instance (2) and registers itself. The Boss creates a connection back to the BridgeGoose (3) and confirms the registration. `A` and `C` can be the same machine.
An SSH tunneled connection (bottom) is more complex: The BridgeGoose establishes an SSH connection to the SSH server on `B` (1) and creates a tunnel to the RMI registry. Through this first tunnel it connects to the RMI registry (2) and asks for the Boss address (relative to `B`). It creates a second, outgoing tunnel to the Boss and the local RMI socket is modified to point to the local tunnel endpoint. The BridgeGoose is exported to RMI address space, resulting in a local incoming connection endpoint towards which a third, incoming tunnel is created from the SSH server. RMI socket information is modified to replace the local endpoint by the SSH tunnel's entrance on `B`. The Goose connects to the Boss instance through the second tunnel (3) and registers itself. The Boss creates a connection back to the BridgeGoose through the third tunnel (4) and confirms the registration. Any pair of `A`, `B`, and `C`, or all three can refer to the same machine.

server and login credentials. Tunnels are automatically created and GaggleBridge monitors the tunnels to check if they need to be reestablished (e.g., after a network problem) and to query remote Boss instances about the number and names of Geese connected to them (figure 6.3).

Establishing the SSH connection is more complicated from a technical point of view, as several RMI connection endpoints need to be modified in the right order (using `Reflection`) to redirect traffic through the tunnels (see figure 6.4).

7. SeaSight: Integration of Sequencing and Microarray Data

Mayday is a powerful and feature-rich platform for microarray-based transcriptomics analyses. With the recent introduction of new high-throughput sequencing technologies [20, 47, 122], transcriptomics studies based on sequencing of transcripts (termed “RNA-seq” [173]) have become a feasible alternative to microarrays. In the course of this work, Mayday was extended to allow its application to data from RNA-seq experiments as well as to incorporate a framework for raw data import (for data from different underlying technologies) and normalization. The following sections describe Mayday's capabilities without the SeaSight extension, related software, the desired result of implementing the extension, and considerations for the design. General features of the implementation are described as well as several highly-optimized data structures supporting efficient computations. Finally, an example analysis is presented.

7.1. Mayday without the SeaSight extension

Mayday was originally conceived as a visualization tool for normalized microarray data [56] offering several visualization plots and clustering methods. As expected for a first version, Mayday's features were not sufficient to perform all steps necessary for transcriptomic analyses. Data normalization and preprocessing was excluded from the original scope of the program and was usually accomplished using `R`, which was also employed for statistical testing. Data exchange between `R` and Mayday was simplified by Matthias Zschunke's R Interpreter plugin (see section 5.2.2).

Thus, the first version of Mayday was a tool placed at the end of the preprocessing pipeline, used for analysis and visualization of data normalized by other programs. (Normalized) data could be imported from tab-separated text files. Raw data could not be imported directly, however for some formats (Affymetrix CEL, Illumina), import modules were implemented over time as student projects [162, 82]. These modules made use of `R` and appropriate parser packages to read and normalize the data before loading it into Mayday.

7.2. Related Software

The new possibilities and challenges resulting from the RNA-seq technology have led to the development of many new software tools, in the academic as well as the commercial fields. In this section, only methods developed and made available as

Table 7.1.: Related Software

Name	published	PF	GUI	CP	features
ArrayExpressHTS	Jan 2011 [63]	R	no	no	read mapping, filtering, expression quantification
ASC	Nov 2010 [181]	R	no	no	statistical testing (single method)
baySeq	Aug 2010 [66]	R	no	no	statistical testing (single method)
DESeq	Oct 2010 [4]	R	no	no	statistical testing (single method)
edgeR	Oct 2009 [130]	R	no	no	statistical testing (single method)
ERANGE	May 2008 [109]	Python	no	no	expression quantification
GenPlay	May 2011 [94]	Java	yes	no	genome browser view of array & seq data
Galaxy	May 2010 [24]	Python	yes	no	transcript assembly, quantification
rnaSeqMap	May 2011 [96]	R	no	no	data input, transcript detection, memory management, 'building blocks'
KNIME+modules	Aug 2011 [80]	Java	yes	no	data input, quantification
RseqFlow	July 2011 [172]	Python	yes	yes	mapping, expression quantification statistical testing
rQuant	Oct 2009 [25]	Matlab	no	no	expression quantification
TopHat	Feb 2009 [160]	C++	no	no	transcript detection, quantification
SeaSight (+Mayday)	Jan 2011 [14] Sep 2010†	Java	yes	yes	data input, quantification, normalization, statistical testing, visualization

Software programs in the fields of RNA-seq analysis and joint analysis of RNA-seq and microarray data. PF, programming platform; GUI, graphical user interface; CP, does the program offer the complete pipeline from raw data to visualized statistical testing results; †, date of submission

open-source or free software by academic researchers will be discussed. Most are implementations of individual methods as R packages building on existing projects. Recently, more comprehensive packages have been published which combine several aspects of the data analysis pipeline, such as normalization and statistical testing, for instance. Such packages provide a foundation on which programmers can implement their own methods and pipelines. As with most R packages, these are geared towards "power users" and require detailed knowledge of R, the respective package and the underlying statistical methods. Some also provide a (semi-)automated workflow suitable for most analyses. Finally, stand-alone software packages offer methods to perform (at least part of) the analysis outside of R. The remainder of this section will introduce several such tools together with comments on their scope (an overview is given in table 7.1). Note that many of these tools were published after our submission of SeaSight to PLoS one.

7.2.1. Single method implementations

Implementations of single steps in the usual RNA-seq analysis pipeline largely fall into two groups, *expression quantification* methods and *statistical testing* methods.

Expression quantification of RNA-seq data (based on mapped reads and genome annotations) is a complex task. Reads that map non-uniquely to the respective genome (so-called *multi-reads*) must be handled in a meaningful way, and, more importantly, signals resulting from several transcript variants being expressed at different levels at the same time must be deconvoluted to obtain the real signal for each transcript. ERANGE [109] is a set of python scripts that can be used to compute expression (RPKM) values for a given set of transcripts taking multi-reads into account. TopHat [160] is a stand-alone program that computes the real expression values for alternatively-spliced transcripts (so-called FPKM values). rQuant [25] uses model fitting to compute transcript expressions based on the observation that the expression strength along each transcript does not follow a uniform distribution. It also reports RPKM values.

Once expression values have been obtained, statistical tests can be employed to find significant differences between several (classes of) samples. Such tests are often applied in the `R` computing environment and all eminent methods in this category are implemented as `R` packages. One of the first published methods, edgeR [130] models the distribution of reads per transcript by a negative binomial distribution, estimating the parameters from the data. This idea was expanded by the authors of DESeq [4] who implemented a method to estimate the (negative binomial) distributions for each gene in each sample even in the absence of replicates, which is an important feature as long as sequencing is still too expensive for most researchers to produce the required number of replicates for more robust statistical methods. The Analysis of Sequence Counts (ASC) [181] package also offers a method that works without replicates, “borrowing information across sequences” to estimate the distribution of sample variation. Differential expression is then detected using an empirical Bayes method. A similar statistical approach is taken by the baySeq [66] package, albeit requiring the presence of several replicates for each condition.

Interestingly, the statistical methods for differential expression (DE) detection do not work with RPKM expression values as input, since these already are the result of some basic normalization (namely, by exon length and sequencing depth), but rather require the input data to be a numerical (integer) matrix of *observed read counts* per transcript per sample. This fact indicates a (hopefully transient) division of the community into two groups: On the one hand, researchers develop sophisticated transcript expression quantification algorithms based on statistical models and algorithmic optimization, the results of which are used as input for relatively simple methods for calling differential expression. On the other hand, very simple quantification methods (i.e., counting) are used to provide input data for sophisticated statistical methods for DE detection. Whether either of these approaches will dominate in the future is unclear, but it is likely that a combination of both will be used. Yet since the statistical methods rely on the assumption of a discrete distribution of counts (e.g., the negative binomial distribution) while the quantification methods produce expression estimates on a continuous scale, it is hard to see how the gap between the two can be bridged.

7.2.2. Pipelines & Frameworks

RseqFlow [172] is a web-based pipeline which encompasses all steps from read mapping, via expression quantification (RPKM) to detection of differential expression (using the DESeq method). The extensible workflow management system can spread the workload over several machines and installation is simplified by the fact that the authors offer a virtual machine for download.

Another web-based solution is provided by the Galaxy platform [60]. Wrapping around external tools for mapping (e.g., Bowtie [95]), data conversion (SAMtools [98]), splice junction prediction (e.g., TopHat [160]) and transcript assembly (Cufflinks [161]) as well as simple differential expression detection methods (Cuffcompare and Cuffdiff), it offers users an easy to use tool for basic expression analysis. The project's focus, however, is not on transcriptomic studies, and higher-level analyses such as clustering or machine learning methods are not offered at present.

The inappropriately named rnaSeqMap [96] package does not offer any mapping capabilities. Rather, the package contains building blocks which, according to the authors, "may be used to construct processing pipelines that iterate over fragments of chromosome, genes or intergenic spaces". As such, the package serves as a bridge between data backends, offering connections to mySQL databases, reads stored in BAM [98] and plain-text tabular files, and statistical analysis packages such as DESeq and edgeR as well as custom analysis scripts. The two key features introduced with this package are an implementation of the Aumann-Lindell algorithm for finding 'irreducible regions' (e.g., transcribed exons) from RNA-seq data, as well as memory management methods programmers can employ to adapt their own analysis methods to large data. These methods allow for the 'slicing' of the genome into 'manageable parts' in terms of memory. While this certainly improves the situation for programmers who need to handle large data in `R`, it begs the question why the actual 'slicing' (i.e., the determination of the correct parameters for the trade-off between speed and memory use) is left as an exercise to the user of the package. Techniques such as memory mapping (see section 7.6.3) could be used to *transparently* 'slice' the data without placing an additional burden on programmers.

ArrayExpressHTS [63] is an analysis pipeline implemented as an `R` package. Starting with read sequences in FastQ format, the pipeline calls an external read mapping program (Bowtie [95], TopHat [160] or BWA [97]), and converts the resulting SAM files to BAM format using the SAMtools program [98]. The aligned reads are then imported into `R` and expression is quantified, resulting in `ExpressionSet` objects which can be passed on to statistical methods such as DESeq or edgeR for differential expression detection. The package offers methods to run the time-consuming mapping tasks on a computing cluster.

The Konstanz Information Miner KNIME [22] is a generic platform for building processing workflows from small modules. Recently, new modules were introduced for tasks specific to next-generation sequencing data [80]. These include input and output modules for the FastQ, and SAM/BAM formats, a module for removing adapter sequences from reads, and, most importantly, a module for computing transcript counts from mapped reads. While these new modules open the door for the development of KNIME workflows for RNA-seq data, all further processing, as well as statistical and visual and analysis steps are left to external programs. The authors suggest the use of `R` for these tasks.

7.2.3. Stand-alone applications

GenPlay [94] is a stand-alone tool able to import data from different sequencing experiment types (RNA-seq, ChIP-seq) as well as array-based experiments for visualization in a genome browser view. It offers some very basic computations (such as computing the mean, minimum and maximum over windows of fixed or variable sizes). Data normalization as well as more sophisticated statistical computations are not offered. Furthermore, in contrast to SeaSight and the Mayday Genome Browser (section 4.3) which offers track types for displaying aligned read data and derived coverage values ('wiggle tracks'), GenPlay does not use sophisticated memory management and relies on loading *the complete dataset* into main memory. The authors state a requirement of 24 GB of main memory to work with multiple tracks at base pair level and suggest the use of a dedicated workstation. On computers with fewer resources, users are forced to either accept extremely slow interactivity due to excessive use of on-disk swapping of memory, or can not perform their analyses at all.

7.3. The motivation for creating SeaSight

As shown in the previous section, several tools have emerged for dealing with RNA-seq data. They cover one of the three fields of expression quantification, statistical testing, or visualization. None of the programs described offers methods for all three steps in the usual data analysis pipeline and most are implemented as R packages requiring extensive knowledge of that environment before they can be employed to produce meaningful results.

While individual implementations of new methods are necessary to bring the field of RNA-seq analysis forward, a software offering several of these methods in a common user interface is highly desirable if the new methods are to be widely accepted. Furthermore, such a software can use an internal data format that makes translating data between different formats (such as R object types) unnecessary. The new sequencing technologies result in new kinds of input data for expression studies. Research on this kind of data is currently hindered by the need to manually bring the raw data into a form suitable for detailed study. Furthermore, combining data from different sources such as microarrays and RNA-seq is a difficult problem due to the different nature of the data.

The aim of SeaSight was to provide a common framework for transcriptomics data pre-processing. By integrating different methods for expression quantification and normalization together with importers for many file formats for RNA-seq data as well as raw microarray data and annotation formats, it enables users to perform *combined* analyses of data from different sources. Mayday offers a large number of tools to analyze and visualize normalized expression data. These methods are basically technology-agnostic and do not rely on the data coming from a traditional microarray experiment.

Some methods, such as the t test, are based on the assumption that data is normally distributed. Raw read data does not follow a normal distribution, but after appropriate normalization, *logarithmic normalized read counts* can essentially be treated like logarithmic normalized single-channel microarray data (see section 7.8 for an example analysis). Regarding the 'divide' between the 'sophisticated normalization plus standard statistics' and the 'simple counting plus sophisticated statistics' approaches mentioned in section 7.2, SeaSight is positioned on the former side, converting raw read data into 'microarray-like' expression data. Thus, data imported and processed using SeaSight can be analyzed with the full breadth of methods already implemented in Mayday.

7.4. Design

SeaSight's initial design was created by first identifying *challenges* posed by the nature of the input data and the processing tasks that should be possible in the new framework. A set of very basic *assumptions* was formulated about what the typical application of SeaSight to new data would involve. Based on these two, *design requirements* were stated. The next three subsections will briefly deal with these three categories, followed by an overview of the steps required to process data from the raw input files into a form that can be used within Mayday.

7.4.1. Challenges

Raw data from transcriptomics experiments are highly diverse, coming from different technologies and experimental platforms. They vary in terms of the file formats used, the overall data volume produced for one sample, the possible sources of experimental bias, the distribution of signal values, and also regarding the set of transcripts queried. This results in the following challenges:

- **Data integration:** Data can be obtained from different types of microarrays, it can be in the form of aligned reads from RNA-seq experiments, or it can be data normalized by some external pipeline. These data types need to be imported and integrated into a common format.
- **Data storage and handling:** High-throughput technologies produce large amounts of raw data. Especially data from RNA-seq experiments requires optimized data structures that allow for efficient handling of the data during quantification and normalization.
- **Data normalization:** Different technologies have different shortcomings. These include experimental noise and systematic bias. As a consequence, different methods are needed to correct these problems, and there are usually a host of competing methods available.
- **Data summarization:** RNA-seq experiments allow for the creation of base-specific expression profiles. However, each sequencing run (resp. sample) will likely contain data for a different subset of genomic locations due to sampling, while no information is available for all other locations. Also, not all experimental procedures produce strand-specific information. The locations queried by microarray experiments depend on the set of probes represented on the arrays

used. When different technologies, or different array platforms (or different versions of the same platform) are combined in one study, a common set of interrogated genomic locations has to be produced as the basis for comparisons between samples.

7.4.2. Assumptions

A typical application of SeaSight starts with importing raw data for each sample and ends with the production of a Mayday `DataSet`. Based on imported raw sample data, 'experiments' are defined, one for each sample. Normalization methods are applied on a per-experiment, per-technology, and finally per-dataset basis. A set of common genomic locations is produced and expression is quantified for each location in each experiment. The resulting expression matrix is the basis for transcriptomics analyses using Mayday's analysis and visualization methods. As a basis for the design of SeaSight's data structures and processing workflows, seven assumptions can be formulated for this process:

1. A dataset is built from several experiments, each of which contains data obtained using a specific technology.
2. Raw experimental data can be very large, especially for RNA-seq experiments. For most workstations, the raw data size of one experiment is already larger than available main memory.
3. Experiments are usually performed in groups, where each experiment in the group is performed using the same technology. Other levels of grouping can be created based on other relevant characteristics, e.g., which lab performed a given experiment.
4. Normalization involves performing successive steps of transformation on each experiment. Such steps can be technology-specific (e.g., microarray background correction) or generic (quantile normalization to remove inter-sample distribution differences). Some steps will be applied to one experiment at a time (e.g., intra-experiment normalization) while others operate on a group of experiments (e.g., inter-experiment normalization). These groupings of experiments into sets usually differ between different steps in the normalization pipeline.
5. For each transformation method, an acceptable state for the input data is defined, limiting the applicability of the method to experiments fulfilling this requirement. By applying the transformation, the experiments' properties are changed, resulting in a new state for each experiment. This state is a combination of data type, available data streams (e.g., two-channel microarray foreground and background values), genomic locations, and additional annotations.
6. Transformation steps can be very time-consuming, due to complex computations or the large data volume that needs to be processed. Users might want to set up a processing pipeline without waiting for each transformation to finish before selecting the next method. If the outcome of processing the raw data is not satisfactory, a user might want to replace individual steps of the

pipeline with alternative methods. Such alternative normalization runs might be performed within the same session, or at a later time.

7. The end result of SeaSight processing a dataset is a state in which all experiments are *comparable*. Firstly, this means that the sets of genomic loci queried in the different experiments overlap (i.e., that a common set of loci has been constructed). Secondly, it also means that the expression values fall into the same range (and possible also have the same resolution). Experiments are comparable if their values are *semantically identical*, that is if a specific expression value in one experiment has the same meaning if encountered in another experiment.

7.4.3. Design requirements

From the challenges identified and assumptions formulated above, certain design requirements follow:

1. Experiments need to be modeled in a very flexible way allowing for the inclusion of data from very different sources (assumption 1). This involves provisions for different data structures with a set of common methods and capabilities, as well as parsers for a range of input formats, mostly for the different microarray data formats. A description of experiment types and available parsers is given in section 7.5.2.
2. To handle the expected data volumes efficiently and allow for quick application of transformations even on standard workstations, optimized data structures are required (assumption 2). The data structures developed for SeaSight are explained in section 7.6.
3. Transformations are applied to (sets of) experiments in a sequential manner, with different input sets for each transformation (assumption 3 and 4). Thus a mapping of transformations to (sets of) experiments is required to construct the input data for each transformation, and a mapping of experiments to (ordered lists of) transformations is needed to plan the execution of the normalization 'pipeline'. How these mappings are modeled is described in section 7.4.4, the data structure responsible for these mappings is described in section 7.5.1. The input state of one transformation is determined by the output state of the preceding transformation, or set of transformations in case several experiments are used as input. The properties of transformations and a list of currently implemented transformations are presented in section 7.5.3.
4. SeaSight must have a means to describe the properties of an experiment, covering different aspects of the data (assumption 5). These must be structured such that transformations can specify acceptable input and predicted output states. Experiment properties are described in section 7.5.2.
5. The SeaSight user interface must allow users to quickly set up and configure a transformation 'pipeline' without forcing them to wait for processing between adding transformations (assumption 6). An indication of the expected result of applying the pipeline to the input data should be provided. It should be possible to insert, remove and replace transformations and re-run the pipeline.

Such changes might be desired a long time after the initial application of the pipeline (e.g., several months later), thus serialization of the complete pipeline, its settings and the input data is needed. The user interface is introduced in section 7.5.5, serialization is described in section 7.5.1.

6. As experimental values from different experiments (platforms/technologies) can only be compared based on their association with genomic locations, SeaSight must include data structures for genomic coordinates, as well as parsers for several data formats used to store and exchange such information. Transformations applicable to genetic coordinates should also be included, together with methods to join different sets of coordinates into a common set (assumption 7). These methods are described in section 7.4.6 and their implementation is presented in section 7.5.4. Finally, validation methods should be offered that can help users in avoiding problems due to incompatible coordinate definitions (e.g., chromosome name mismatches).

7.4.4. Modelling the transformation steps

The mapping of transformation steps to unordered sets of experiments and of experiments to ordered lists of transformation steps can be represented by an $n \times m$ matrix T (the *transformation matrix*). Each row represents one of the n experiments and the order of rows is of no significance. Each column represents one *'step'*, and the order of columns represents the order in which 'steps' are executed.

The number of columns, m, is at most equal to the number τ of transformations, but can be smaller ($m \leq \tau$). This is possible whenever different transformations can be applied in parallel. Two transformations t_1 and t_2 can be executed in parallel, if

1. they act on disjoint sets of experiments, and
2. there exists no dependency between them, i.e., the input set of t_1 is not dependent on the output of t_2 or vice-versa.

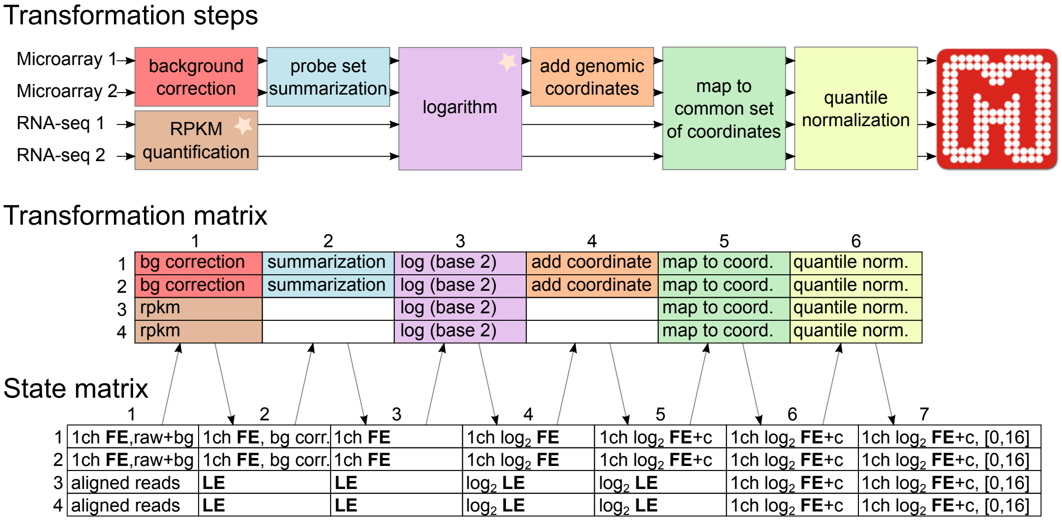
Thus, m is the total number of successive steps needed to apply all transformations taking into account all opportunities to execute transformations in parallel.

Let the term *transformation instance* denote a specific application of a transformation, that is the triple of transformation, the input set of experiments and (optional) parameter values. Then each cell of the matrix T contains zero or one transformation instance and each instance can occupy one or more cells. The cells assigned to a specific transformation instance all occur in the same column but need not be in consecutive rows. Finally, cells in T can be left empty, in which case the data in the respective experiments is not altered in that step. The transformation matrix can be used for efficient execution of the transformation pipeline with a relatively simple algorithm shown in listing 7.1.

To allow users to setup the transformation matrix without actually executing the individual steps, the state of each experiment needs to be modeled at the beginning (raw state) as well as after each applied transformation (intermediate states) up to the final state. This is done with a second matrix, S (the *state matrix*) of size $n \times (m + 1)$. Here, the ith column contains the state of each experiment before the ith transformation step. The first column contains the raw state, columns 2 to m

```
1. Create a variable $data_e$ for each experiment $e$.
2. Assign the raw data object of each experiment $e$ to $data_e$.
3. For each column $T_{\cdot j}$ of $T$:
    1. For each transformation instance $t_k$ in $T_{\cdot j}$:
        1. Collect the set $E_{t_k}$ of experiments that $t_k$ uses as input
           This is equivalent to finding all $i$ with $T_{ij} = t_k$.
        2. Execute $t_k(E_{t_k}, \text{parameters})$ in a new thread.
        3. Update $data_e$ with the respective output of $t_k$ for each $e \in E_{t_k}$.
    2. Wait for all threads to finish.
4. Create the final expression matrix based on the expression vectors in $data$.
```

Listing 7.1: SeaSight pipeline execution based on the transformation matrix



Transformation matrix

	1	2	3	4	5	6
1	bg correction	summarization	log (base 2)	add coordinate	map to coord.	quantile norm.
2	bg correction	summarization	log (base 2)	add coordinate	map to coord.	quantile norm.
3	rpkm		log (base 2)		map to coord.	quantile norm.
4	rpkm		log (base 2)		map to coord.	quantile norm.

State matrix

	1	2	3	4	5	6	7
1	1ch **FE**,raw+bg	1ch **FE**, bg corr.	1ch **FE**	1ch log$_2$ **FE**	1ch log$_2$ **FE**+c	1ch log$_2$ **FE**+c	1ch log$_2$ **FE**+c, [0,16]
2	1ch **FE**,raw+bg	1ch **FE**, bg corr.	1ch **FE**	1ch log$_2$ **FE**	1ch log$_2$ **FE**+c	1ch log$_2$ **FE**+c	1ch log$_2$ **FE**+c, [0,16]
3	aligned reads	**LE**	**LE**	log$_2$ **LE**	log$_2$ **LE**	1ch log$_2$ **FE**+c	1ch log$_2$ **FE**+c, [0,16]
4	aligned reads	**LE**	**LE**	log$_2$ **LE**	log$_2$ **LE**	1ch log$_2$ **FE**+c	1ch log$_2$ **FE**+c, [0,16]

Figure 7.1.: Example transformation and state matrices: The transformation steps to convert raw data into a Mayday dataset are represented in the transformation matrix while the state matrix holds the (predicted) state of each experiment before (resp. after) applying the transformations. Colors are used to distinguish transformation instances. Transformations with lazy evaluation (see section 7.7.2) are marked with a star. FE, feature expression data (expression values are accessible by feature name); LE, locus expression data (expression values can be computed based on genomic regions of interest); 1ch, single channel data; bg, background intensities; c, genomic coordinates. In the final state, all four experiments contain comparable data (as defined in section 7.4.2).

contain the input states for the transformations in the respective columns of T and the last column $(m+1)$ contains the state after application of all transformations (see figure 7.1 for an example of T and S).

7.4.5. Constructing the transformation and state matrices

During matrix setup, when transformations are selected and added to groups of experiments by the user, the state matrix is employed to decide which transformations are applicable to a given input set of experiments: Each transformation method implements a test method which, given an experiment state, decides whether the transformation can be applied to this experiment. This is based on considerations such as whether the application is *technically possible*, that is whether input data is in the right format (e.g., preventing the application of microarray background correction methods to sequencing data), as well as whether the application is *useful* (e.g., applying a log-transformation to negative data).

Users can add transformations into any column of the matrix. Using the selected set of input experiments and their state at the desired location, SeaSight creates a list of applicable transformations for the user to choose from. If a transformation method offers configurable parameters (via the `Settings` framework), it is initialized based on the input experiments and their states, and a configuration dialog is presented to the user.

If the transformation is inserted at the 'end' of the current processing pipeline, it is simply added to the transformation matrix and the state matrix is updated with the new final state for each transformed experiment (as reported by the transformation method). If, however, the transformation is added somewhere else, i.e., added before the first transformation, inserted between a pair of transformations, or added by replacing another transformation, the process of adding it to the matrices is more complex and may require user interaction. The details of this process are thus presented with SeaSight's user interface in section 7.5.5.

7.4.6. Locus-based data integration

When data is obtained using different technologies or different platforms, the problem of data integration not only concerns the different distributions of the resulting expression values, but also the mapping of values between different experiments: Microarray platforms differ in the transcripts they query, the names used for these transcripts (such as vendor-specific names, RefSeq accession numbers, gene names, etc.) and the genomic origin of the probe sequences, and each sequencing run produces reads (potentially) mapping to different locations.

The one information that is available for all transcriptomics technologies are the genomic locations of the features (coordinates of probe sequences, mapping position of reads). Thus data integration can be achieved by mapping all information to genomic coordinates and then constructing the expression matrix based on these locus-mapped expression values (see figure 7.2). For microarray data, the set of genomic locations is obtained from annotation files linking probe (set) names to

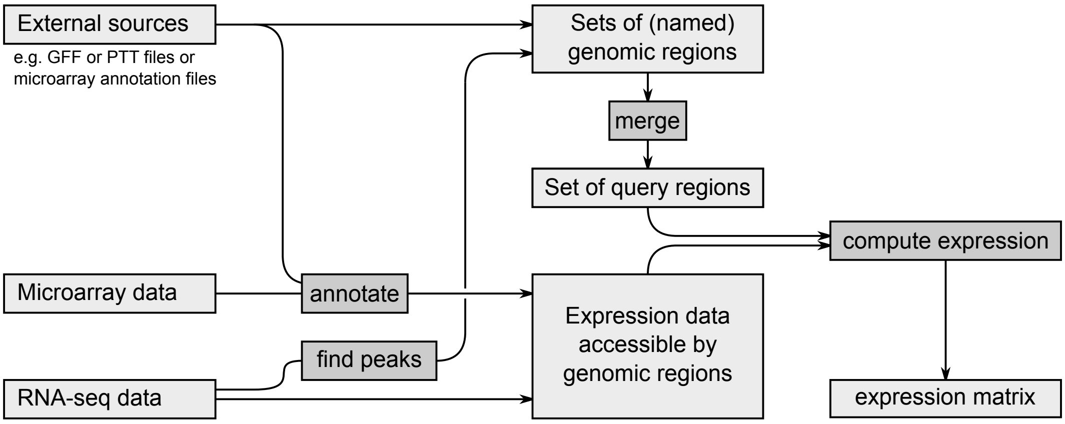

External sources
e.g. GFF or PTT files or microarray annotation files
Sets of (named) genomic regions
merge
Set of query regions
compute expression
Microarray data
annotate
Expression data accessible by genomic regions
find peaks
RNA-seq data
expression matrix

Figure 7.2.: Locus-based data integration. Experimental data is mapped to genomic locations either intrinsically (for RNA-seq data) or by annotating feature-based expression data with genomic regions from external sources. A set of *query regions* is either taken from an external source, or obtained by finding peaks in aligned read data, or created by merging several such input sets of genomic regions. The final expression matrix results from querying each experiment for expression values for each of the genomic regions in the query set.

genomic locations. For RNA-seq data, this information is attached to each read during the mapping step. One has to take care to make sure that both annotations are based on the *same version* of the respective reference genome, i.e., microarray probe expression data annotated with locations referring to the human genome in, say, NCBI's version 18 can only meaningfully be combined with RNA-seq reads mapped to the same genome version.

Different experiments with locus-based expression values can then be combined into an expression matrix based on a set of genomic *regions of interest* (e.g., the set of known genes for the given organism): For each locus, each experiment is queried to provide an expression value based on the locus coordinates. The exact method differs based on the experiment type. RNA-seq data, for instance, provides transcript count data at base resolution, which has to be appropriately summarized into an expression value for each locus, as described on page 101). Constructing the expression matrix can be seen as a three step process:

1. Select genomic regions of interest ('*query regions*'). These can come from one source (e.g., the set of known genes for a given organism) or from several sources, for example from the sets of regions queried by different microarray platforms which were used to obtain the raw data.
2. If more than one source is selected, combine the sets of regions into a single set. Different methods for this combination can be devised, some of which are discussed below.
3. Based on the (combined) set of query regions, merge the available data into an expression matrix.

Each of these three steps will be presented in more detail below.

Sources of genomic regions

Users may want to select the set of query regions from one or more sources, depending on the respective study. Such sources can be

- **Microarray probe annotations** which assign a genomic coordinate to each (named) probe (set) represented on the array.
- **Algorithms** which create a set of query regions, for instance by splitting the genome into a set of regions of equal size (a '*tiling*' approach). A more sophisticated approach would be to use a data-driven algorithm such as a gene finder (called 'peak finder' in ChIP-seq literature) which tries to predict transcript regions from the *coverage* of reads at each genomic position. In this case, names have to be constructed for the features, e.g., by describing their genomic coordinates in human-readable form.
- **External annotation files** containing a list of named genomic regions in one of several formats (e.g., GenBank feature files, protein table files, tabular text files).

Within SeaSight, genomic coordinates can be imported from external annotations files in a wide variety of formats (see section 7.5.4 for a list) to be used as a source for the set of query regions. Alternatively, imported coordinates can be mapped to microarray features (or any other feature-based expression values, e.g., such values imported from tabular text files of data normalized externally).

For RNA-seq experiments, a simple peak-finding algorithm is offered to predict a set of putative transcripts from the read coverage. However, due to the complexity of the gene-finding problem, especially in eukaryotes with their exon-intron gene structure, the use of dedicated tools (such as 'G-Mo.R-Se' [45]) for this step is encouraged.

The input for the second (coordinate set combination) step can come from external files, from coordinates derived or computed from imported experiments, or they can be the result of applying *coordinate transformations* on one of the above. Such coordinate transformations (presented in section 7.5.4) may be used for two very different reasons. Firstly, annotations often differ in the way numeric coordinates are represented (either using zero or one to represent the first base of the genome and either including or excluding the last base), as well as in the textual representation of species and chromosome names. For example, the largest human chromosome could be referred to as 'Chromosome 1', 'chr1', just '1', or by an accession number such as `NC_000001.10` (Assembly GRCh37.p5). Thus, to make the coordinates from different experiments (or from experiments and query regions) compatible, transformations might be needed. To spot such problems, SeaSight offers a 'consistency check' method which can detect possible problems with incompatible coordinate specifications and issue a descriptive warning (see also section 7.5.4). The decision whether to continue without changes or apply a coordinate transformation is left to the user.

The second reason for using coordinate transformations is related to the actual data used, instead of only to the specifics of the annotation: For example, one might be interested in relating gene expression values with the level of chromatin modification in the genes' promoter regions (as determined by chromatin immunoprecipitation, ChIP). In this case, the genomic coordinates associated with the ChIP data could be transformed by shifting them downstream by a constant term such that they overlap with the transcript region assessed by the gene's microarray probes. Thus the resulting expression matrix would directly link gene expression in one column with promoter chromatin modification in another column.

Another example would be changing the strand information (forward resp. backward strand) in the genomic coordinates associated with an experiment studying antisense RNA expression to allow its direct comparison with gene expression on the sense strand.

Combining sets of genomic regions

The decision how to combine several sets of genomic coordinates into one set of *query regions* depends on the data studied and on the researcher's interest. Within Mayday SeaSight, several methods are available (see figure 7.3) which cover a broad range of possible applications.

The most simple approach is the *union* method, which constructs the set of query regions by taking the set union of all input sets of genomic coordinates. This means that the combined set will contain all coordinates of the input sets, excepting exact duplicates (same species, chromosome, strand, start and end coordinates, and exon structure). This approach can be useful when the genomic regions come from two versions of a microarray design, where both platforms have a large overlap in the regions queried. For most other applications, however, the union approach is too simplistic and more complex schemes based on the genomic regions' nucleotide positions (start, stop, exon boundaries) have to be applied. SeaSight offers four such methods, which differ mostly in how fine-grained the resulting regions cover the input regions, resp. how strongly they aggregate input regions into larger query regions:

- **All fragments** are computed by building a single ordered list containing the start and end coordinates of all covered regions. Based on this list, all regions that lie between pairs of coordinates (irrespective of whether these are start or end coordinates) and are covered by at least one input region are defined as query regions. This method results in a large number of query regions but achieves the highest possible resolution of features over the genome.
- **Overlap merging** is a method which creates query regions by starting with a region defined in one input set and extending that region to the left and right until all overlapping regions in all input sets are covered. The covered regions are removed from the input and the next starting region is chosen. This can be useful for instance if the input regions are from different microarray platforms which cover slightly different subregions of an organism's genes. The minimal required overlap can be defined as a parameter of this approach.
- **Greedy merging** is a relaxed version of the overlap merging method. In addition to merging all overlapping regions, the candidate query region is extended to the left and right to also include non-overlapping input regions in close proximity as defined by a maximal distance parameter. The application

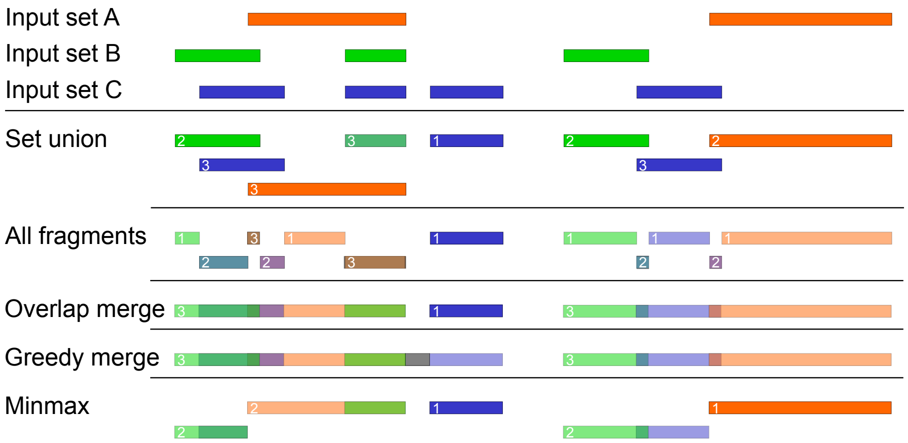


Figure 7.3.: Combination methods for sets of genomic regions. Three input sets of genomic regions (A,B,C, top) are combined according to five different schemes (see text for detailed descriptions). Colors are used to visualize the merging of regions. Numbers indicate the number of input coordinate sets covered by the merged regions. If the input sets are from experiment data (e.g., microarray data annotations), this is equivalent to the number of non-empty columns in the expression matrix row representing the respective region.

of this method can be helpful for datasets where the above-mentioned overlap merging is too strict to actually merge, for example, microarray targets from different platforms.

- The **minmax approach** processes regions from all input sets in decreasing order of size. Starting with the largest input region, all other regions overlapping it are merged until a (user-defined) maximal size is reached. Overlapping regions that would increase the merged region's size beyond the limit are clipped (the overlapping part is removed). They can later form the starting point of a new query region. Remaining regions below a minimal size are removed.

A final filtering step can be added to remove regions which are not represented in enough experiments, according to a user-defined threshold. Thus, rows which contain a large number of `NA`s (i.e., are undefined for a large number of experiments) can be removed from the expression matrix. This is especially useful for approaches which generate a large number of candidate query regions, such as the 'all fragments' method. Removing regions below a certain minimal size limit is advisable to avoid introducing artifacts in the computed expression values, such as extremely high RPKM values resulting from normalizing the number of reads (R) by a very small region size ($K < 1$, given in kilobases).

Summarizing expression values

When the expression matrix, the final output of running SeaSight, is constructed, each experiment in the transformation matrix corresponds to one column of the expression matrix, and each region in the set of query regions corresponds to one row. Thus, each cell in the expression matrix results from computing an expression value for a pair (experiment, region). The method to compute this expression value differs depending on the type of data associated with the experiment:

For microarray experiments, the expression can be computed from the probes overlapping the query region. If several probes are covered by the region, their values are combined using an appropriate statistic, such as the median or mean, possibly assigning different weights to the probes depending on the percentage of overlap they share with the query region.

For RNA-seq data, where each query region contains a number of reads which also might be zero, different quantification methods have been proposed and are implemented in SeaSight, ranging from simply counting the number of overlapping reads (optionally weighing them according to the length of their overlap), to more sophisticated schemes such as the RPKM measure which takes into account that the number of reads mapping into a genomic region is not only proportional to the number of RNA transcripts produced from that region but also increases with the length of the region and with the total number of reads produced in the sequencing run.

Finally, the rows of the expression matrix are annotated with the respective genomic region in the form of meta information objects of the `LocusMIO` type and the row (probe) names are set according to the names of the query regions (if available) or a human-readable description of the region's coordinates.

Using locus-based data integration is only one of the possible routes SeaSight allows users to take. If only feature-based expression data are to be imported, the integration can be performed based on a mapping between the feature names (e.g., probe identifiers) of the different experiments. It is also possible to construct the expression matrix (i.e., create feature-based expression data from locus-based data) as an *intermediate* step in the transformation matrix and apply further (feature-based) normalization methods, such as inter-experiment quantile normalization.

7.4.7. Summary

Building a Mayday dataset from raw data produced by different platforms consists of (a subset of) the following steps:

1. Parse raw data files and create experiments
2. Apply platform- and experiment-specific transformations (e.g., bias correction)
3. Integrate data
    - based on feature names:
        - map feature names between different experiments
        - create a set of common features
    - based on genomic locus:
        - find peaks from RNA-seq experiments
        - load genomic regions from external files
        - apply coordinate transformations
        - annotate feature-based experiments with coordinates
        - construct a set of query regions
4. Summarize expression levels for the final set of features resp. regions
5. Apply further transformations (e.g., inter-experiment normalization)

7.5. Implementation

The previous section discussed the ideas behind the design of Mayday SeaSight and introduced the fundamental data structure, the *transformation matrix* which links *experiments* and *transformations*. The implementation of these abstract concepts is presented in the following, together with an overview of the experiment data types, file parsers, transformations, and coordinate transformation methods currently implemented. The rather high-level view of the implementation presented here is complemented by section 7.6 which gives detailed descriptions of the underlying efficient data structures.

7.5.1. The matrix

The central data structure in SeaSight is the transformation matrix modelled by the `TransMatrix` class. Its main purpose is to manage the relationship between the experiments and the transformations and to offer methods for manipulating the matrix by inserting or removing experiments, or by inserting, removing, or replacing transformations. Instead of storing two matrices explicitly (the transformation and the state matrix), the `TransMatrix` class stores three mappings which, in combination, contain the same information:

- **Experiments→Transformations:** This is required, among others, to compute the current state of each experiment.
- **Transformations→Experiments:** Based on this information, the input set for each transformation can be constructed.
- **Transformations→Execution index:** The execution index is equivalent to the column index of the transformation instance in the transformation matrix.

In addition, the `TransMatrix` keeps track of annotation objects which arise during runtime but are not associated with any specific experiment, either because they describe the complete `DataSet`, or because they are identical for a whole group of experiments (e.g., a mapping of feature names to genomic coordinates) and need only be stored once.

The creation of a `DataSet` at the end of the transformation pipeline is not an action performed by the `TransMatrix` class after pipeline execution. Instead, some transformation plugins implement the special empty interface `DatasetCreatingTransformation`. When such a transformation is applied to a set of experiments, it creates a new `DataSet` and moves it into Mayday's core data structure, the `DataSetManager`. Before executing the pipeline, the `TransMatrix` class checks whether all experiments contain at least one `DatasetCreatingTransformation` and warns the user in case some experiments might have been 'forgotten' during matrix set-up.

Modelling `DataSet` creation as 'just another' transformation has several benefits: Firstly, several different plugins can implement this functionality in different ways (which is already done to support dataset creation directly from `LocusExpression` data without going through a mapping transformation). Secondly, it allows users to create several `DataSets` from one transformation matrix using the central feature of applying transformations to a subset of all experiments. A use case would be to normalize a large number of experiments together and then split them into several DataSets for easier handling within Mayday. Thirdly, as the dataset creating transformations create a DataSet as a side effect only without modifying the data in the transformation matrix (i.e., without changing $data_e$ for the affected experiments), users can create 'snapshots' of their data at different steps in the transformation pipeline. This also introduces the possibility for creating DataSets with overlapping sets of experiments as input.

Besides controlling matrix set-up and execution, the TransMatrix class is responsible for *serialization*, i.e., storing the whole transformation matrix to a file (and loading it again later on). This involves storing all transformations as well as their properties, their position in the transformation matrix, the raw data for each experiment, any additional coordinate data included in the matrix (e.g., resulting from locus transformations, coordinate merging or peak finding), and additional mappings such as feature–feature mappings, or probeset–probe mappings. The whole serialization process makes heavy use of Mayday's plugin manager and the serialization capabilities of the Settings framework (which was, in fact, initially created exactly for this purpose).

The SeaSight file format is based on the very popular compressed ZIP file format. It contains several file entries, one for the transformation matrix, one for each transformation instance, each experiment, each set of genomic coordinates, and one for each mapping. During the serialization process, each experiment is asked to serialize its data into its respective file entry. For this, the subclasses of the Experiment type are linked to serialization classes which can be generic, e.g., for tabular feature expression data, or highly specific, only working for that particular kind of data. It is up to the serialization class to decide whether to serialize the experiment data directly, or whether to simply write a reference to an external file (e.g., an Affymetrix CEL file) into the experiment data stream and ask the TransMatrix to embed that external file in the SeaSight project file. As a result, the final SeaSight file contains all data necessary to recreate the transformation matrix at any later point while still being relatively compact.

7.5.2. Experiment properties

Experiments in SeaSight are modelled as a combination of elements that are defined upon importing the raw data and do not change when transformations are added/executed ('static properties'), properties that are affected by the addition of transformations to the transformation matrix ('set-up properties'), and properties that change during execution of the pipeline ('runtime properties'), as shown in figure 7.4. The static properties include the experiment name, information about the data source, the initial (raw) state of the data and the raw data itself.

The set-up properties include annotations that are added during the matrix construction, as well as the 'current' state, i.e., the state of the experiment in the last column of the state matrix. Intermediate states are not stored explicitly, but can be quickly created on demand. At runtime, the current data (called $data_e$ in section 7.4.4) is

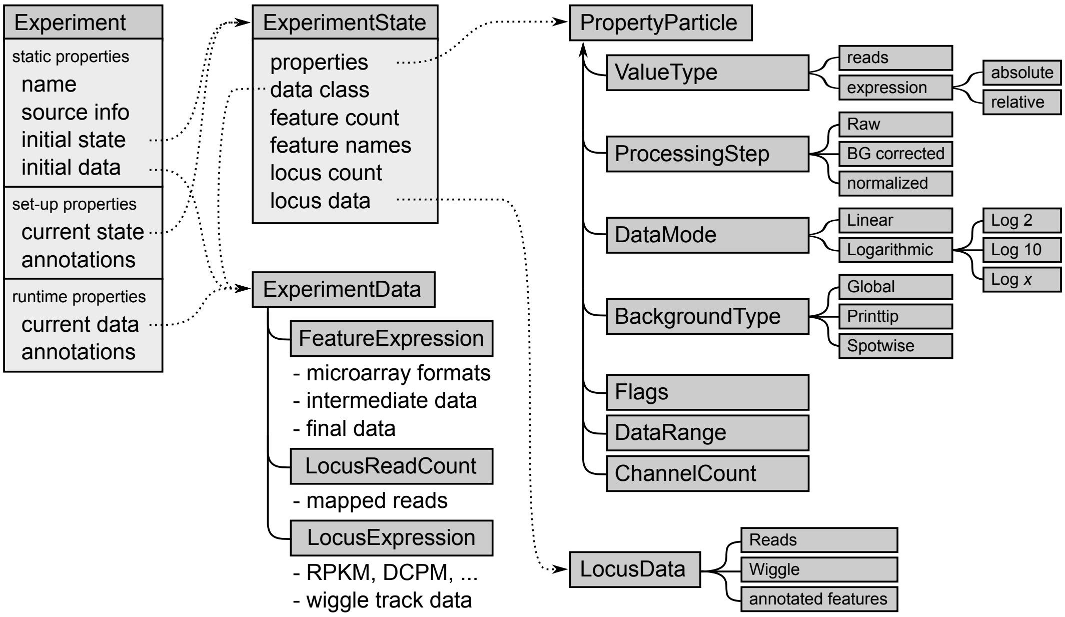


Figure 7.4.: SeaSight experiments: Experiments are characterised by static properties including an initial state and the primary experiment data. During pipeline set-up, the state of the experiment changes according to the transformations that are applied to it. When the pipeline is executed, the data content of the experiment is changed successively. The actual experiment data is separated from the experiment's state to allow for quick checking of the applicability of a given transformation during set-up. The experiment state includes a set of properties which can be mutually exclusive (e.g., data is either linear or logarithmic).

also stored, together with annotations that are added by some transformations and will become part of the Mayday `DataSet` at the end of processing.

As experiments need to be able to store vastly different kinds of data, the actual data is not part of the `Experiment` object itself, but is stored in an instance of `ExperimentData`, of which, currently, three different implementations exist:

- **FeatureExpression** data contains a set of uniquely named features, each of which is associated with a numeric value, as well as (optional) information about the layout of these features on the microarray surface, which is required by some normalization methods. This is the format most similar to Mayday's expression matrix representation, but in contrast to an expression matrix column, an object of the `FeatureExpression` type can hold several *data channels*. For example, raw data from a two-channel microarray experiment contains four data channels, namely foreground and background values for each of the two channels.
- **LocusReadCount** data can produce a set of overlapping objects for any genomic region, e.g., mapped reads.
- **LocusExpression** data can produce a numeric value for any genomic region, e.g., RPKM values.

Table 7.2.: SeaSight experiment parsers

FeatureExpression	LocusReadCount	LocusExpression
Tabular text files	Tabular text files (read, position)	Wiggle files
Mayday `DataSets`	SAM files of mapped reads	
Affymetrix CEL files	BAM files of mapped reads	
Agilent feature files		
GenePix files		
ImaGene files		
ScanArray files		

Most parsers are implemented for raw microarray data, as there exist many different file formats (with often even different versions) most of which can not be parsed using Mayday's very flexible tabular text file parser. Mapped read data is most commonly either stored as SAM/BAM files or in a tabular file format defined by the read mapping software used.

The initial data of an experiment is created by a file parser class which reads one or more raw data files and produces the appropriate `ExperimentData` object (see table 7.2 for a list of supported file formats). During runtime, the content (and type) of the 'current data' field can change with each successive application of a transformation to the experiment. To allow transformations to decide whether they are *applicable* to a certain input experiment, the data type (Java `Class`) of the `ExperimentData` object is part of the experiment's state which is described by the `ExperimentState` class. This class also stores information about the number and names of features (if present) as well as the number of locus objects (e.g., if features are annotated with genomic coordinates). Such locus data can be used for the generation of the set of query regions (section 7.4.6). In many cases, locus data objects are only lazily created upon access (and then cached, see section 7.7.1) as they can be expensive in terms of construction time and memory requirements.

The most important aspect of the `ExperimentState` are the experiment properties. Together with the type of the `ExperimentData`, experiment properties are the main basis on which transformations define their applicability and users can assess the predicted result of running their transformation pipeline. They are stored as a set of `PropertyParticle` objects describing different aspects of the data, such as the data mode (linear, logarithmic), or the type of microarray background information present. `PropertyParticle`s can define mutual exclusion rules to ensure that the experiment properties are sensible. For example, all `PropertyParticle` inheriting from the `DataMode` class are mutually exclusive, as data can be either linear *or* logarithmic, and if it is logarithmic, it can either be base two or base ten, but never both.

7.5.3. Transformations

Transformations in SeaSight are realized as plugins implementing the `Transformation` interface (or, for the most part, inheriting from the `AbstractTransformationPlugin` class). Transformation authors need to implement functions to determine

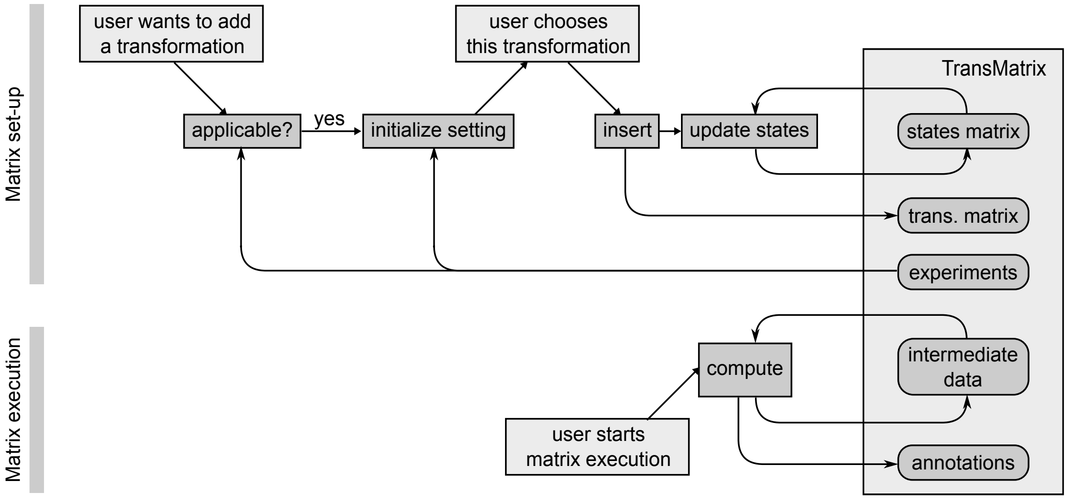


Figure 7.5.: Interactions between transformations and the `TransMatrix`. During matrix set-up, transformations use information regarding the current state of experiments (i.e., their input) to decide whether they are applicable to the data, as well as to initialize configurable settings. When an applicable transformation is added to the transformation matrix, the experiment states are updated accordingly. During matrix execution, transformations access intermediate experiment data (processed by preceding transformations) and update it with the result of their own computations.

the applicability of their transformation based on a set of input experiments (and their states), to generate a new `ExperimentState` (see section 7.5.2) based on input state(s), and (optionally) to initialize configurable settings based on the input set of experiments (see figure 7.5). The actual computation of the transformation, performed during matrix computation, can obtain input data (in the form of `ExperimentData` objects) from the `TransMatrix`, and modify or replace this intermediate data. General annotations which should not be linked to one specific `Experiment` can also be deposited with the `TransMatrix`.

Due to this relatively simple interface and the implementation of a lot of functionality in the `TransMatrix`, `AbstractTransformationPlugin` and `AbstractExperiment` classes, transformation authors can focus on the implementation of their actual method and only need to implement three SeaSight-specific functions, as well as the registration method for Mayday's plugin manager for their method to become part of SeaSight. A list of currently implemented transformation methods is presented in table 7.3.

7.5.4. Genomic coordinates

Mayday can read genomic coordinates from a variety of file formats (see table 7.4), including the most commonly used GFF (generic feature) and GenBank formats. In addition, a parser for tabular text files can be used to import genomic coordinates from almost any tabular format. Based on Mayday's very generic parser for tabular text files, which can deal with different column separators, files with or

Table 7.3.: SeaSight transformations

Microarray data: background correction	*Sequencing data*: Add pseudo read counts
– spotwise subtraction	*Sequencing data*: expression quantification
– normal+exponential model	– read count
– RMA correction	– arithmetic mean coverage
	– geometric mean coverage
	– reads per million mapped, RPM
	– reads per kilobase of exon model, RPK
	– reads per kilobase per million, RPKM
	– depth of coverage per million, DCPM
	– square root arcsin coverage
	variance stabilizing, $f(x) = \sqrt{n}\,\arcsin\sqrt{x/n}$
Multi-channel feature data: misc.	*Single-channel feature data*: summarization
– channel extraction	– arithmetic/harmonic mean, median
– channel swapping	– minimum, maximum
– channel renaming	– *median polish
– Red/Green → MA transformation	
Multi-channel feature data: normalization	*Single-channel feature data*: normalization
– dye-swap	– *scale mean to 0 and/or std.dev. to 1
– loess (locally-linear regression)	– *scale according to a given percentile
– loess using print-tip information	– *quantile scaling
– *reference channel quantile scaling	
Feature expression data: map values	*Locus expression data*: map values
– logarithmically	– logarithmically
– exponentially	– exponentially
– logarithmically to range $[0, 16]$	
Feature expression data: transform loci	*Locus expression data*: transform loci
Feature expression data: features	
– change feature names	
– *restrict to common set	
Data conversion:	Pass-through side-effect methods
– Locus→Feature expression	– Write feature expression vector to file
(using query regions)	– Compute mate-pair distance distribution
– Feature→Locus expression	
(add locus annotation to features)	`DataSet` creation
– Feature→Read count	– from feature expression data
(simulate reads for expressed features)	– from locus expression data

There are more than 40 transformations available in SeaSight, listed here according to the type of input data they accept. Transformations marked with a star (*) require several experiments as input, all others can be applied to one or more experiments.

Table 7.4.: Supported file formats for genomic coordinates

Tabular text files (generic parser)
Tabular text files (MAYDAY LocusMIO type)
GenBank format (single- and multi-locus)
EMBL format (similar to GenBank format)
PTT (protein table) format
GFF (generic feature) format, version 2 and 3, including exon models

without headers, and different quoting characters, this parser allows coordinates to be specified in any column order, missing information (e.g., chromosome name) can be specified by the user, and different encodings, for instance for the strand (such as +/−, F/R, D/P, W/C, or S/A) are automatically detected.

Coordinate transformations (see section 7.4.6) can be applied to imported coordinates, to coordinates attached to feature-based expression data (e.g., annotated microarray data), as well as to coordinate-based expression data (e.g., mapped reads). SEASIGHT's locus transformation method implements methods which can be used alone or in combination to

- **Replace species and/or chromosome names** which is helpful when integrating data from different sources.
- **Change the strand location** by defining a mapping between the four possible strand locations 'forward', 'backward', 'both', and 'unspecified'.
- **Move coordinates** by a fixed number of bases, thereby moving all their exons.
- **Move upstream and/or downstream coordinates**, enlarging or shrinking the coordinate's region.
- **Set the length** of the coordinate by either placing the upstream coordinate relative to the downstream coordinate or vice-versa.
- **Convert exon models to primitive coordinates** either by creating one coordinate for each exon or by creating a large coordinate spanning the whole region from the first to the last exon. See section 7.6.8 for details on the differences between complex and primitive coordinates.

Methods for merging multiple sets of genomic coordinates into a set of query regions as discussed in 7.4.6 are implemented as efficiently as possible. The union, all fragments, overlap merging and greedy merging methods run in $O(n \log n)$ in the number of input coordinates, the logarithmic component being necessary for sorting the coordinates in the input sets. The runtime of the minmax method is $O(n \log nk)$ where k is the (average) number of overlapping loci which are considered for merging.

To ensure that the set of query regions and the experiments' coordinates are compatible and can be used to create an expression matrix, SEASIGHT includes a 'Consistency check' method. The motivation lies in the fact that there is no generally accepted scheme for species and chromosome identifiers in bioinformatics. As a result, different datasets and annotation files use different identifiers for the same chromosomes, resulting in the need for manually mapping identifiers. To be able to

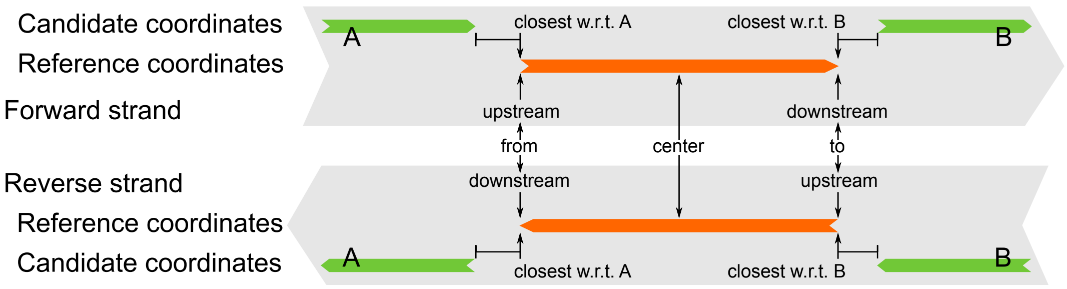


Figure 7.6.: Coordinate anchor points. The 'upstream' and 'downstream' points are mapped to the strand-independent 'from' and 'to' positions depending on the coordinate's strand location, while the 'center' point is independent of strandedness. If pairs of coordinates are considered, e.g., for distance calculations, there is also a 'closest' anchor point which is either the upstream or the downstream coordinate, whichever is closer to the second coordinate's selected anchor point.

correctly map identifiers, the first step is to recognize that there is, in fact, a problem with a given set of data, preferably before a time-consuming normalization pipeline is run in vain. The consistency check method implemented in SeaSight inspects the input and query coordinate sets and issues warnings for three classes of problems:

1. **Query set not covered:** If the query set contains species or chromosome names which are not covered by the experiments, the resulting expression matrix will contain some rows consisting only of `NA` values.
2. **Experiment set not covered:** If input experiments contain data for species or chromosome names which are not contained in the query set, this data will not be used. For example, if the query set and some experiments use the species name 'Homo sapiens' while other experiments use the species identifier 'hsa', the resulting expression matrix will have some columns completely made up of `NA`s.
3. **Duplicated chromosome names:** Some data sets contain multiple identifiers for the same chromosome, for instance due to unfinished assemblies used in read mapping (e.g., chromosome '2' vs. chromosome '2_random'). More importantly, when merging multiple sets of coordinates into one query set, there could be similarly-named chromosomes which should have been merged (e.g., 'chr2' and 'chromosome 2' denote the same chromosome).

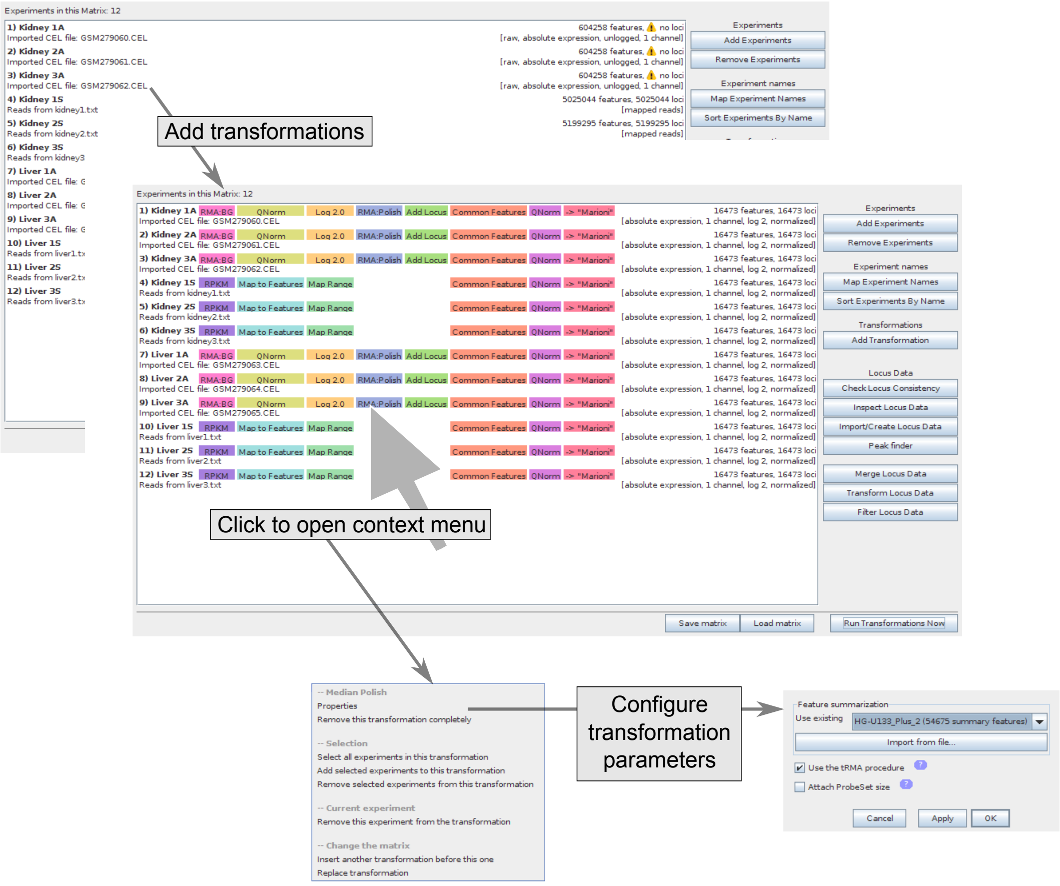
Finally, SeaSight also offers a method to filter one set of coordinates based on another set. Using $X = \{x_i\}$ to represent the set being filtered and $Y = \{y_j\}$ the set to filter against, the method *keeps* each $x_i \in X$ whose distance $\min_j d(x_i, y_j)$ is smaller than a user-defined threshold. The distance calculations can be done in a strand-specific (only using y_j which are on the same strand as x_i) or strand-agnostic fashion and the distance thresholds can be defined separately depending on whether y_j is upstream or downstream of x_i. The *anchor points* used for distance calculations can be selected independently for coordinates in X and Y (e.g., comparing the upstream coordinates of X with the downstream coordinates of Y). Available choices are 'upstream', 'downstream', 'center' and 'closest' (see figure 7.6).

7.5.5. The SeaSight user interface

SeaSight's user interface is a rather straightforward representation of the transformation matrix (see figure 7.7). For each experiment (row), the name and a description of the source of raw data is given, followed by the list of transformations which work on the experiment. Transformations are represented by boxes with a short label describing the transformation. The boxes' background color is used to convey the information which boxes (all in the same column) belong to the same transformation *instance*. Thus, the visualized properties are the mapping of experiments to transformations (transformation box y coordinate), the mapping of transformations to execution steps (transformation box x coordinate), the mapping of transformation instances to sets of input experiments (transformation box color), and the ordering of experiments in the final output dataset (experiment row order).

The order of experiments can be changed by drag&drop or by sorting the experiments according to their names. Experiment properties such as the experiment name, for instance, can be configured. External (tabular) files can be used to rename all experiments automatically. Transformations can be appended after the last transformation in the matrix, before the first transformation or between any pair of transformations (either by inserting or by replacing an existing transformation) using a menu available by clicking on any transformation box. Further actions in this menu allow users to change the input set of the respective transformation instance (e.g., by removing the current experiment from the input set), as well as to modify the parameters of the transformation method.

If a transformation is inserted, removed, or replaced, which is not the last transformation in each concerned experiment, or if such a transformation's properties are changed, other transformations to the right of it might not be applicable any more. To prevent users from unwanted side-effects, SeaSight evaluates the consequences of the change before applying it: First, a copy of the original transformation matrix is created and all further changes are applied to this copy. All transformations 'to the right of' the changed transformation are removed from the matrix, as are all further transformations depending on them (which may also be in other rows of the matrix). Then, the desired change is performed (by inserting, removing, replacing or reconfiguring a transformation), resulting in new output states for the affected experiments. The previously removed transformations are added back to the matrix one by one, always checking whether they are still applicable to the changed experiments. If any such test fails, the respective transformation is put into a list of unsatisfied transformations. If all transformations could be added without problem, the changes to the cloned transformation matrix are applied to the original matrix. Otherwise, the user is presented with a list of transformations that *would be deleted* as a result of the intended change, and given the choice to either go through with the changes, or cancel the process and reconsider.



Experiments in this Matrix: 12

1) Kidney 1A
Imported CEL file: GSM279060.CEL
604258 features, ⚠ no loci
[raw, absolute expression, unlogged, 1 channel]

2) Kidney 2A
Imported CEL file: GSM279061.CEL
604258 features, ⚠ no loci
[raw, absolute expression, unlogged, 1 channel]

3) Kidney 3A
Imported CEL file: GSM279062.CEL
604258 features, ⚠ no loci
[raw, absolute expression, unlogged, 1 channel]

4) Kidney 1S
Reads from kidney1.txt
5025044 features, 5025044 loci
[mapped reads]

5) Kidney 2S
Reads from kidney2.txt
5199295 features, 5199295 loci
[mapped reads]

6) Kidney 3S
Reads from kidney3

7) Liver 1A
Imported CEL file: G

8) Liver 2A
Imported CEL file: G

9) Liver 3A
Imported CEL file: G

10) Liver 1S
Reads from liver1.t

11) Liver 2S
Reads from liver2.t

12) Liver 3S
Reads from liver3.t

Experiments in this Matrix: 12

1) Kidney 1A RMA:BG QNorm Log 2.0 RMA:Polish Add Locus Common Features QNorm -> "Marioni"
Imported CEL file: GSM279060.CEL
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

2) Kidney 2A RMA:BG QNorm Log 2.0 RMA:Polish Add Locus Common Features QNorm -> "Marioni"
Imported CEL file: GSM279061.CEL
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

3) Kidney 3A RMA:BG QNorm Log 2.0 RMA:Polish Add Locus Common Features QNorm -> "Marioni"
Imported CEL file: GSM279062.CEL
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

4) Kidney 1S RPKM Map to Features Map Range Common Features QNorm -> "Marioni"
Reads from kidney1.txt
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

5) Kidney 2S RPKM Map to Features Map Range Common Features QNorm -> "Marioni"
Reads from kidney2.txt
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

6) Kidney 3S RPKM Map to Features Map Range Common Features QNorm -> "Marioni"
Reads from kidney3.txt
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

7) Liver 1A RMA:BG QNorm Log 2.0 RMA:Polish Add Locus Common Features QNorm -> "Marioni"
Imported CEL file: GSM279063.CEL
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

8) Liver 2A RMA:BG QNorm Log 2.0 RMA:Polish Add Locus Common Features QNorm -> "Marioni"
Imported CEL file: GSM279064.CEL
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

9) Liver 3A RMA:BG QNorm Log 2.0 RMA:Polish Add Locus Common Features QNorm -> "Marioni"
Imported CEL file: GSM279065.CEL
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

10) Liver 1S RPKM Map to Features Map Range Common Features QNorm -> "Marioni"
Reads from liver1.txt
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

11) Liver 2S RPKM Map to Features Map Range Common Features QNorm -> "Marioni"
Reads from liver2.txt
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

12) Liver 3S RPKM Map to Features Map Range Common Features QNorm -> "Marioni"
Reads from liver3.txt
16473 features, 16473 loci
[absolute expression, 1 channel, log 2, normalized]

Feature summarization
Use existing: HG-U133_Plus_2 (54675 summary features)
☑ Use the tRMA procedure
☐ Attach ProbeSet size

Figure 7.7.: SeaSight user interface: The transformation matrix is visualized as experiments (rows) to which transformations (columns) are applied. On the left, each experiment is named and information about the raw data is shown. Color is used to show the input set of experiments for each transformation (i.e., each transformation *instance* is assigned a distinct color). Experiments can be selected by clicking on their name and they can be freely reordered using drag&drop. On the right, the final state (corresponding to the last column of the state matrix) of each experiment is described. Buttons (far right) can be used to add and remove experiments, rename them using an external file of name mappings, sort experiments by name, add a new transformation to selected experiments, as well as perform operations on locus data. Buttons at the bottom of the window allow users to save resp. load the matrix, and to start the execution of the transformation pipeline. Clicking on any transformation opens a context menu (bottom) from which the transformation can be removed, the input set of experiments modified, another transformation added in front of or in place of the current transformation, and parameters of the current transformation changed. As an example, the configuration dialog of the 'Median Polish' transformation is shown (bottom right). Example data from [105].

To the right of the last transformation box, the final state of each experiment is described, showing experiment properties (see section 7.5.2) as well as the number of features resp. genomic loci covered. Further actions, such as coordinate transformations, coordinate set merging methods and the consistency check function (all described in section 7.5.4) are accessible from a list presented at the right of the main SeaSight window. Serialization (loading and storing) as well as the actual execution of the matrix can be triggered from the buttons at the bottom of the window.

7.6. Efficient Data Structures

7.6.1. Requirements

Mayday SeaSight should be broadly applicable to the different data types mentioned above and its data structures should allow programmers to relatively easily implement their transformations in an efficient manner concerning time and memory consumption. Time requirements depend on the time complexity of operations defined on the underlying data structures as well as on implementation details such as caching (see 7.7.1). They are usually in a balance with memory requirements, e.g., fast access is trivially achieved by raising memory complexity. The large size of expression data from high-throughput experiments precludes this naïve approach and requires optimized data structures complemented by implementation details such as memory mapping (see 7.6.3), caching (7.7.1) and memory locality.

These abstract requirements translate into the following concrete demands, specifically concerning objects mapped to genomic coordinates:

Time

1. constant time insertion of an object at a given (complex) coordinate
2. constant time access to the any element covering a given location
3. constant time access to the any element spanning a given location
4. constant time access to all information concerning a given element
   (such as the start and end coordinates, strand etc.)
5. constant time access to all mapping positions of a given element

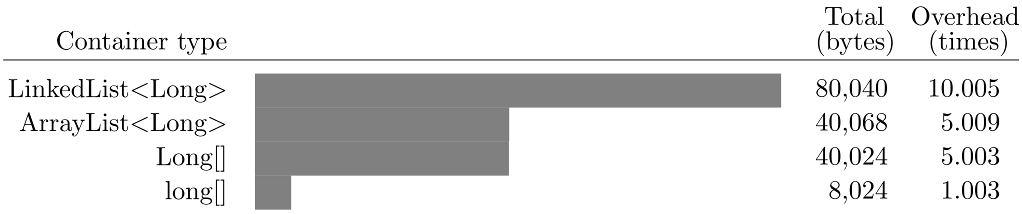
Memory

6. space linear in the number of bases covered,
   independent of the size of the genome
7. space linear in the number of elements covering a given base

How these requirements translate into data structures will be explained in the following sections.

7.6.2. Primitive types

The Java language knows two basic types of objects: primitive and class objects. Primitive (or *native*) objects represent data types native to the hardware, such as



Container type		Total (bytes)	Overhead (times)
LinkedList<Long>		80,040	10.005
ArrayList<Long>		40,068	5.009
Long[]		40,024	5.003
long[]		8,024	1.003

Figure 7.8.: Memory requirements of different containers. As an example, the total number of bytes necessary to store 1000 numeric values of type `long` in different Java containers are shown.

numeric types of different length, characters and boolean values. Class objects can represent any kind of complex, structured data type. While many different containers for class objects exist (linked lists, vectors, maps, sets), the only collection able to hold values of primitive type is an array. Arrays have several drawbacks as they lack many of the desirable properties of the container types (such as O(1) insertions in linked lists, or finding an object in O(1) in hash sets), and they are of fixed size. To allow programmers to use the basic numeric, character and boolean types in generic type specifications (such as template definitions for containers), Java defines one class type for each primitive type.

These corresponding types, however, introduce an enormous penalty regarding memory (often referred to as *object tax*), as can be seen in figure 7.8. As an example, consider a collection of 1000 long values. The native `long` type consumes 8 bytes of storage. The corresponding object type, `Long`, adds another 24 bytes (three 64-bit pointers, assuming a 64-bit operating system), a three-fold overhead. Since arrays of objects are in fact arrays of references to objects, a non-native `Long`[] array of 1000 `Long` values is an array of 1000 references (each 8 bytes long) to 1000 objects, each requiring 24+8 bytes, which results in a total of 40 bytes per `long` value, i.e., 40,000 bytes for all values, plus another 24 bytes object overhead for the array itself. If smaller types such as `int` or `byte` were used, the overhead would be even larger since all Java objects are aligned in memory to addresses divisible by eight bytes.

The more convenient `ArrayList`<`Long`> type is basically only a wrapper around a `Long`[] array, adding another 44 bytes (an 8-byte reference to the array, the 4-byte size of the array, two 4-byte positions of the last and first element in the array, as well as the 24-byte object overhead for the list itself). Finally, a `LinkedList`<`Long`> which allows for $O(1)$ insertion and removal of objects has the largest penalty as it stores each individual `Long` value in a separate entry object, from which the double-linked list is constructed. Thus, each entry contains a reference to the value (8 bytes), a reference to the previous as well as next entry (2×8 bytes), i.e., 48 bytes per entry *in addition to* the 32 bytes consumed by the referenced `Long` object, totalling 80 bytes per value, a tenfold overhead.

With the large volumes of data that SeaSight is required to handle, especially in the context of RNA-seq experiments, native objects must be used as often as possible. As a consequence, it was necessary to develop efficient data structures capable of storing data of primitive types, accessing them in an efficient manner, and allowing the use of *memory-mapping* (see Section 7.6.3). These structures are presented in the next sections, following a more detailed requirements analysis.

7.6.3. Memory-mapped structures

Memory mapping is the process of representing (part of) a file stored in a filesystem inside the memory space of a running process [177, 158]. Thus, if the process accesses a certain address space in its virtual memory, it is in reality accessing data on the filesystem. This approach lies at the heart of modern operating systems as it allows the OS's virtual memory manager to allocate more virtual memory to each process than there is real physical memory available, which is called *overcommitting.* If physical memory is exhausted (or used by other applications), individual *pages* of virtual memory are written to disk ("*paged out*") to be read back ("*paged in*") later when they are accessed again. Such pages are either stored in so-called "swap" files (the accurate term is *paging files*) or dedicated partitions.

Apart from the automatic memory management done behind the scenes by the operating system's virtual memory manager (VMM), programmers can also employ this method to map any file into their program's memory. Existing files can be mapped read-only, write-only or for read-write mode. If a non-existent file is mapped, it will be created by the VMM (depending on the process having sufficient rights for the given location).

During the development of SeaSight, new *memory-mapped* data structures were implemented for Mayday. This includes structures to store long arrays of the native Java types, such as `double`, `byte`, `char`, `int`, `long` (varying in length from 1 to 8 bytes depending on the data to be stored), and `boolean` (packed into bytes). Some very specific structures have also been implemented using memory mapping, e.g., a replacement for `HashMap<String,Long>`.

The advantages and disadvantages of memory-mapping in general and with respect to Java applications are discussed in more detail below.

Advantages

Memory-mapping offers several advantages, some of which are of particular interest to programmers developing Java applications:

- **Efficiency:** The VMM is an essential part of the operating system and, due to its importance for every program, highly optimized. Furthermore, it possesses detailed knowledge of the hardware specifics of the given computer and can select the optimal page size for read and write requests. Upon reading, whole pages are loaded into memory, with the VMM heuristically predicting which pages are needed next (such as in forward or backward sweeps over a file). When changes need to be written back to disk, they can be deferred for some time and are then agglomerated into larger write operations if possible.
- **Transparency:** Memory-mapped data can, in principle (see the 'disadvantages' section for more details), be handled like any other data in a processes' memory. No special code is necessary for working with the data, no explicit

loading and storing or calling of file system code is required. For example, the methods implemented in Mayday SeaSight can switch from memory-mapped buffers to pure in-memory buffers on the fly, e.g., if no more disk space is available for mapping, and do so without any special code for each case.

- **Persistence:** If data is mapped in read-write mode, changes to the data in memory will be written back to the mapped file by the VMM. Thus, memory-mapped files are a convenient method to store data persistently and have it instantly available in the next invocation of a program, without the need for defining file formats, writing parsers, or data conversion. Furthermore, if only a few bytes in a huge file need to be changed, the programmer does not have to take care of writing them back at the correct locations, a simple change in memory is sufficient.

- **Memory locality:** While virtual memory allows users to run many applications simultaneously by extending the amount of memory "available", excessive *paging* leads to very poor performance. If it is known in advance that certain data are always processed together, programmers can make use of that knowledge and put these data close together in memory. Thus, working on the data will require only one paging operation, not several. Java objects are placed into the Java heap at locations unknown to the programmer (Java has nothing equivalent to C/C++ pointers). As a result, normal VM paging will likely not find related objects on the same page. If they are instead placed in close proximity in a memory-mapped area, they will be paged in resp. out together (except in cases where they fall on different sides of a page boundary, or are too large to occupy only one page). While working on memory-mapped data is, understandably, slower than directly working in physical memory (also see the next section) for small datasets, the optimized paging possible due to programmers exploiting memory locality reverses this situation if the data are too large to be handled in main memory (see figure 7.9 for an example). If large collections of identically structured objects are to be stored, programmers can place certain aspects of these objects (e.g., the genomic start positions of genes) together in one mapped data block. In this way, algorithms that only process that particular aspect of the objects will only require a smaller amount of data to be paged into memory than if all aspects of each object would be stored together (e.g., start and end position, strand, etc.).

- **Low priority:** Physical memory used for memory-mapping usually has a lower priority than memory used for normal VM paging. This allows time-consuming operations on large data mapped into memory to work in the background while users can still work on their computers without experiencing unresponsive programs that are waiting for their pages to be brought back from disk. If a foreground application needs some part of its memory to be paged in, the VM will simply *drop* a memory-mapped page of the background task from memory (if necessary writing changes back to disk first) and allocate the recovered space to the application.

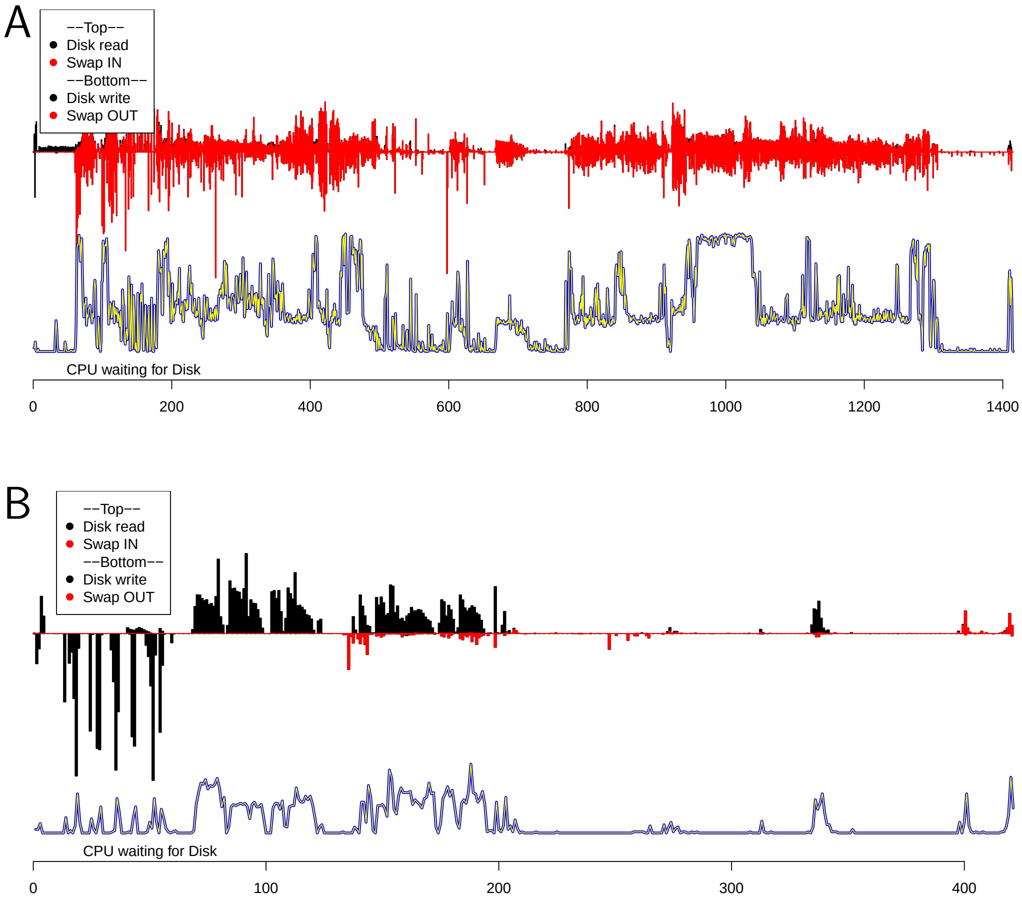


Figure 7.9.: Virtual Memory statistics obtained while importing an RNA-seq dataset using SeaSight. Data was acquired using the Linux `vmstat` command. Disk access is shown in black color, excluding accesses due to virtual memory paging, which are shown in red color. Lines rising from the baseline indicate disk read accesses, lines extending below the baseline represent disk write access. The blue/yellow graph indicates the percentage of time the CPU was waiting for data to become available. A, importing the data without memory-mapping, i.e., holding all data in virtual memory (requiring the Java VM heap space to be set to a value above the limits of physical memory available on the machine); B, using memory mapping and a small Java VM heap size. Note the different total time as indicated by the scale below each plot.

- **Extending Java limits:** Java applications are running inside a virtual machine (VM) which is started with a user-defined, *fixed* maximum amount of virtual memory ("heap space") available to the application. Programs that have to be able to handle data of vastly different sizes must either be run with very high default heap space values to make sure they can handle even the largest files, or programmers have to implement complicated on-demand data retrieval and storage functionality (effectively duplicating the work done by the developers of the virtual memory manager, albeit without knowledge of hardware-specific optimal page sizes etc.). Memory-mapped files increase the virtual memory footprint of the Java VM, but consume only a small, constant amount of Java heap space. Thus they allow to extend the virtual memory available to an application at runtime, in the same way as programs written in languages such as C/C++ can.
- **Extending the paging file:** Each memory-mapped file can be stored in another location on the file system, or even on another file system altogether. Thus, memory-mapped data does not use space in the paging file or dedicated swap partition. Users may, for instance, choose to place a memory-mapped file on a large harddrive to run analyses on very large data, while maintaining only a small paging file (resp. partition) for everyday use.

Disadvantages

Unfortunately, the advantages of *memory-mapping* are somewhat offset by a number of disadvantages, especially when they are used within Java programs:

- **Unstructured storage:** The data that is mapped into memory consists only of a block of bytes. To add structure (such as Java objects referencing other objects), these have to be constructed from the block of bytes either directly after mapping, or lazily upon access. However, constructor invocations are expensive operations, and if objects are created from the mapped data, the benefits of using memory mapping disappear, as the created objects consume Java heap space. Other languages, such as C/C++, allow to set pointers into the mapped data and *cast* them to any object type, but Java does not offer this possibility. Instead, objects need to be created. Using *flyweight* objects, which only provide convenient access to the underlying mapped data without copying the values to the heap can alleviate the problem, but they require more work during programming.
- **Speed:** Due to the need for conversion resp. object creation, memory-mapping can be slower than directly working with Java objects on the heap. However, as soon as the limits of physical memory become relevant, the situation is reversed (see above).
- **Garbage collection:** Being unstructured blocks of data outside of the Java heap, memory-mapped areas can not be processed as a whole by the Java garbage collector. This shifts the burden of memory management towards the application developer. However, as memory mapping is usually employed to store large amounts of static data which is not expected to change over the

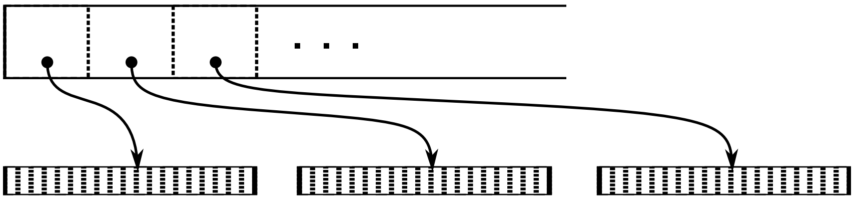


Figure 7.10.: A `LinkedArray` object maintains a list of blocks which are either arrays of a primitive type or `ByteBuffer` objects (to allow memory mapping). Blocks are of a fixed size and can be added to increase the number of elements stored. The total number of elements that can be stored is between 2^{31} (block size 1) and 2^{62} (block size 2^{31}).

lifetime of the program, explicit memory management (e.g., finding unused areas and compacting them) can be disregarded in most cases. Some structures in SeaSight implement compaction functions (as specified by SeaSight's `CompactableStructure` interface), such as the `long` arrays which can switch from an eight-byte `long` representation to shorter representations at runtime (if the range of the stored values permits it).

- **Constant size:** The size of memory-mapped data blocks has to be specified at the point of mapping the file and cannot be changed later (at least not in Java programs). Thus, structures that might need to grow must be implemented carefully, e.g., by mapping more blocks at a later stage (as is done in SeaSight).
- **Debugging:** Since the mapped data is outside the Java heap space and for most of the time also outside of physical memory, debugging applications that use memory mapping is made much more complicated. The contents of the mapped files can not be inspected in the debugger and programmers can only use tools such as hex editors to look into the files that are mapped into memory. Finding the position of a certain value requires knowledge about the size of objects stored, as well as the index of the object of interest. Even then, the bytes at the respective position need to be combined into meaningful information, which can be relatively easy (e.g., when multi-byte numeric values are stored) or very hard (e.g., when the bytes represent a complex structured object).

7.6.4. Large and flexible arrays as the basis

Apart from being fixed in size, a major drawback of primitive arrays in Java is that their maximum size is limited to about 2 billion elements by the maximal value of the (signed) integer holding the array's size. To overcome these two limitations, a more flexible implementation of a native-type array constitutes the foundation for most of SeaSight's data structures. The idea is to create a list of arrays of a predefined size (see figure 7.10). The total size of the data structure can be increased in steps of the block size, random access is possible in constant time and the maximal number of elements is 2^{62} (if using the maximal block size of 2^{31}). These "linked arrays" can be used to store large amounts of data without specifying the exact amount in advance, which is often not feasible, e.g., when importing an unknown number of mapped reads from a large text file. Each native data type requires its own implementation of this structure (all based on an abstract base class that handles block allocation). SeaSight mostly uses the `LinkedLongArray`, `LinkedDoubleArray`, and `LinkedCharArray` classes, the latter of which can handle 2-byte unicode characters or 1-byte ASCII characters, allowing SeaSight to conserve memory for non-unicode strings. A container for boolean values (`LinkedBooleanArray`) is based on the sparse array implementation (see below) and packs 64 boolean values into one long value.

With the introduction of *memory-mapping* (see Section 7.6.3), the internal representation of the data has been changed from primitive arrays to `ByteBuffer` objects, either representing regular in-memory buffers or memory-mapped buffers. In both cases, the number of bytes per element (e.g., 8 for a `long` value) is automatically reduced to the minimum required to represent the values stored in the linked array (e.g., down to 3 bytes per long if the maximal value stored is 2^{24}). The list of files associated with the mapped blocks is also stored to make sure that the files are removed from the file system when the `LinkedArray` itself is garbage collected. However, a small first *block* of each linked array is still stored as a native-type array, for two reasons: Most importantly, it makes sure that very small lists (e.g., small temporary objects) are handled without the memory-mapping overhead. As a beneficial side-effect it also simplifies debugging as at least parts of the objects' content are available for inspection by the programmer.

7.6.5. Efficiently handling millions of lists

The `LinkedArray` classes efficiently handle the storage of large amounts of values. However, while they can be used to store few lists of millions of elements, they are not suitable for storing millions of lists of only a few elements each. The main problem is the overhead associated with the lists themselves (e.g., the 24 bytes object overhead for each list, plus the reference to the internal storage object etc.). Furthermore, the runtime of Java's garbage collector is adversely affected by the need to check millions of objects for live pointers.

These considerations led to the development of the `MultiArray` class. Based on two `LinkedArray` objects, it can manage an arbitrary number of short (unordered) lists of `long` values, placing list headers into the first `LinkedArray` and list contents into the second one (see figure 7.11). New short lists can be added to the object at any time, and the individual short lists can grow (by a specified block size). For the sake of efficiency, lists can neither be removed nor can objects be deleted from them. Furthermore, the only access operation is via a special `Iterator` which traverses the list starting at the most recently inserted element and continues backward. Choosing the correct block size is essential to balance the overhead due to the block 'pointers' with the overhead due to large half-filled blocks.

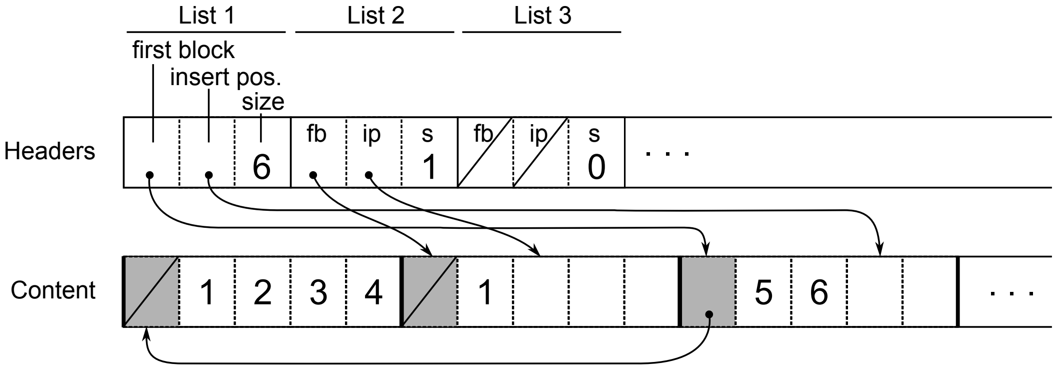


Figure 7.11.: A `MultiArray` object uses two `LinkedLongArrays` to store a large number of short, unordered lists. Lists are built from blocks of a fixed size (here: 4) and chained together back-to-front via 'pointers' (block offsets). New content blocks are allocated to the lists in the order of their creation, leading to an interleaved structure in the 'content' array. The values 1–6 have been added to list '1' to show the insertion order, list '2' contains only one element, list '3' is empty.

7.6.6. Associating data with genomic positions

Data associated with genomic positions is often distributed unevenly, i.e., elements tend to cluster together at certain genomic positions while large stretches of the genome are not associated with any data. Thus it would be very wasteful to store the elements associated with each position in an array of lists of elements, one list per position. While this would allow $O(1)$ access to the elements at a given position, most lists would remain empty and the array alone would be huge. If, on the other hand, pairs of <position,list> were stored, for instance in a hash map or similar structure, access would no longer be $O(1)$. More importantly, Java hash maps consist of many objects incurring a large memory penalty (and garbage collection overhead). Finally, standard maps are very bad in terms of memory locality and performance degrades rapidly as soon as paging is involved.

In Mayday SeaSight, reads associated with genomic positions are one of the most essential data types. The `MappingStore` structure was developed for this kind of data and will be explained in the following sections. It relies heavily on the basic SeaSight data structures (e.g., the multi array and linked array structures described above and the overlap array structure described in the following section) as well as on the structures implemented for genomic coordinates (described in section 7.6.8) and will be fully explained in section 7.6.9.

7.6.7. Sparse Arrays & Overlap Arrays

If only few positions in a large array contain values, data is said to be *sparse*, and the array is *sparsely populated.* SeaSight contains a very memory-efficient implementation of a sparse array which is the basis for data structures that allow to access objects by their genomic position.

The SparseArray class creates a tree structure that can grow as needed, adding new levels to the hierarchy if elements are added beyond the current size of the array. Each level in the hierarchy is associated with a *multiplier* value: At the leaf level, each node can store 1000 objects associated with a genomic range of 1000 bases. The next level nodes store references to 1000 blocks, each of size 1000. On the third level, each block covers a range of 10^9 bases (1000×1000). Accessing elements within this array is possible in $O(\log_{1000} n)$ and its memory requirement is at most $O(\log_{1000} n \times k)$ where k is the number of non-empty positions.

This idea is extended by the OverlapArrayLong class (and associated node classes) which more explicitly models the tree structure and adds two important refinements: Firstly, the size of each node is reduced to 100 elements and instead of storing 100 objects in each leaf node, each of the 100 positions is associated with two lists (managed by two MultiArray instances for all nodes in the OverlapArray), used as follows. Each leaf node *covers* 6400 consecutive genomic positions. If an object is added to a position p, this position is converted to an index $i \in [0, 100[$ as well as an offset $o \in [0, 64[$ such that

$$i = \left\lfloor \frac{p}{64} \right\rfloor \quad \text{and} \quad o \equiv p \bmod 64.$$

A new long value is added to the 'overlap' list at position i with the o^{th} bit set to 1, and a second long value is added to the 'content' list, representing the object just added.

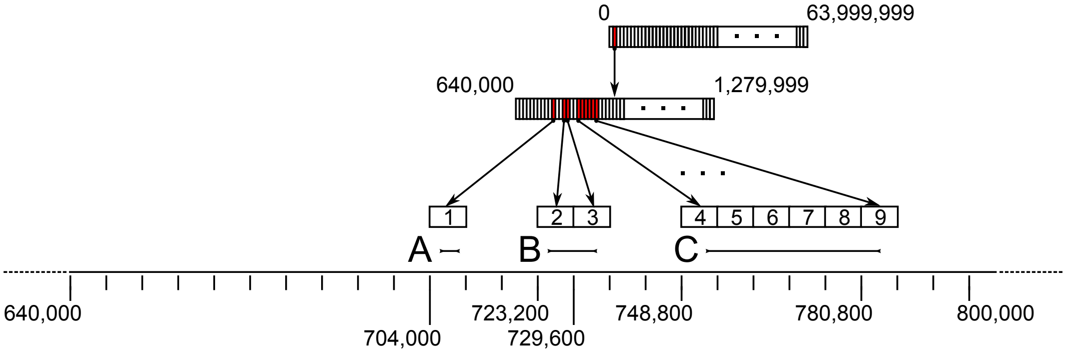
The second refinement in the OverlapArrayLong class is that each node, including internal nodes, also maintains a 'span' list of elements that *completely cover* the genomic range associated with the node. This allows elements that cover large intervals to be stored and retrieved very efficiently.

Thus, if an object spans more than one position (e.g., a gene annotation covering the region $[p_1, p_2]$), the range is converted to a pair of positions (i_1, o_1) and (i_2, o_2). Three cases can be distinguished for leaf nodes (see figure 7.12):

- If $i_2 = i_1$, a long value is constructed with the bits o_1 to o_2 set to 1, and this value is added to the 'overlap' list at position i_1.
- If $i_2 = i_1 + 1$, a long value with the bits o_1 to 63 set to 1 is added in position i_1 and another long value with bits 0 to o_2 set to 1 is added in position i_2 to the 'overlap' list while the same object identifier is added to the 'content' list in both positions.
- If $i_2 > i_1 + 1$, positions i_1 and i_2 are treated as above, and in intermediate positions, the object identifier is simply added to the 'span' list.

As with storing elements in leaf nodes, the left- and rightmost internal node that still contains the element requires special handling (delegating storage to its children in a recursive fashion) while the intermediate nodes simply store the object identifier in their own 'span' list.

The large genome range associated with each node (starting with 6400 bases at the leaf level) is an important factor for keeping the number of node objects stored in Java heap space to a minimum as well as the number of steps necessary to access objects associated with a particular base.



Gene Coordinates & Computed index and offset values

Gene			start			end		
ID	from	to	node	index	offset	node	index	offset
A	704,150	704,160	1	2	22	1	2	32
B	729,590	729,610	2	99	54	3	0	10
C	755,185	780,815	4	99	49	9	0	15

Node contents

```
Node 1, overlap[ 2] = 00000000 00000000 00000111 11111111 00000000 00000000 00000000 00000000
        content[ 2] = {A}

Node 2, overlap[99] = 00000000 00000000 00000000 00000000 00000000 00000000 00000011 11111111
        content[99] = {B}
Node 3, overlap[ 0] = 11111111 11000000 00000000 00000000 00000000 00000000 00000000 00000000
        content[ 0] = {B}

Node 4, overlap[99] = 00000000 00000000 00000000 00000000 00000000 00000000 01111111 11111111
        content[99] = {C}
Node 5, span = {C}
Node 6, span = {C}
Node 7, span = {C}
Node 8, span = {C}
Node 9, overlap[ 0] = 11111111 11111110 00000000 00000000 00000000 00000000 00000000 00000000
        content[ 0] = {C}
```

Figure 7.12.: Example of the `OverlapArrayLong` structure. Three elements (A–C) are associated with positions along a chromosome. Element A falls into a single leaf node, B covers two consecutive nodes and C covers altogether six consecutive nodes. No other elements are stored, thus the nine leaf nodes are the only elements stored in the internal node which in turn is the only node stored at the root node. The ranges covered by internal nodes are written next to the nodes' left and right boundary, the visible range of the chromosome and the ranges of the leaf nodes are indicated at the ruler. The positions of the elements are given in the table with their node, index and offset values computed as described in the text. The (non-empty) values of the nine nodes are shown below in the node contents table.

7.6.8. Genomic Coordinates

Data structures for genomic coordinates were first added to Mayday by Matthias Zschunke [187] in 2006. A genomic locus was defined by a species/chromosome pair, a start and end position on the chromosome, and a strand identifier (forward, backward, both, unspecified).

During the development of SeaSight, two major additions were made to Mayday's handling of genomic coordinates, while preserving the old interfaces and maintaining compatibility with plugins and stored data from previous versions:

1. **Complex coordinates:** While the existing coordinate format was well-suited for studies of chromatin modifications, computing transcript strengths for eukaryotic genes requires a definition of genomic loci able to model multi-exon genes. The coordinate description model used in gff and GenBank files was adopted to describe such coordinates, consisting of the specification of genomic intervals and two operations, namely *join* and *complement*, These create a tree structure describing the genomic locus. Internally, Mayday provides this tree structure as well as a flat structure consisting of a list of *'atoms'*, i.e., (start,end,strand)-tuples. Several convenient methods exist to convert these two formats into each other.
2. **Coordinate containers:** SeaSight needs to be able to access genomic elements by their location (species, chromosome, strand, location). The original genomic coordinates contained references to their chromosome (including the species), but the implementation did not provide access to elements located on a given chromosome (or at a given location). For the type of query necessary for SeaSight, e.g., *find all elements covering chromosome c, positions i to j*, iteratively searching through large lists of (possible unordered) genome-associated elements and comparing their chromosome and position values to c, i and j, respectively, is not feasible. This is complicated further by the complex coordinate structure mentioned above, as well as by the different types of queries that are possible. Take for example the the distinction between elements *covering* a genomic position (i.e., the respective base is part of the element) and elements *spanning* a given location (i.e., the respective base is not part of the element but the element covers bases upstream as well as downstream of that base). To address these issues and prevent code duplication, memory-efficient containers were implemented that provide efficient (in most cases $O(1)$) access to elements located in genomic regions, as well as further methods required for SeaSight. These will be described in more detail below.

Containers for genomic coordinates

Building on the efficient structures introduced above, containers for genomic coordinates were implemented, centering around the `AbstractLocusChromosome` class[1]. Two considerations were taken into account.

[1] public abstract class AbstractLocusChromosome < CoordinateType extends AbstractGeneticCoordinate > extends SimpleChromosome implements CompactableStructure

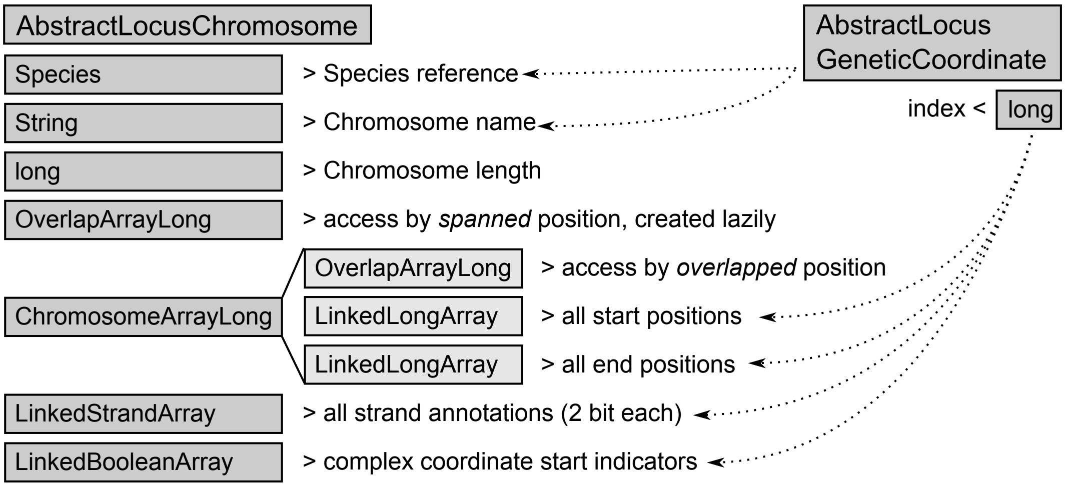


Figure 7.13.: Internal representation of the `AbstractLocusChromosome` class and elements accessed by the `AbstractLocusGeneticCoordinate` flyweight class. The different elements of genetic locations are stored in separate specialized containers, `OverlapArrayLong` instances are used for efficient access by *overlapped* or *spanned* coordinate or coordinate ranges.

Firstly, since the number of chromosomes is typically quite small with potentially millions of elements associated with each chromosome, the chromosome is not referenced by each element but is stored *implicitly*: start, stop and strand information for each element are stored in the respective chromosome object and the element itself is not represented by a Java object most of the time. During access, a small object is created that provides access to all aspects of the genomic coordinate, transparently communicating with the chromosome object, a technique known as the *flyweight* pattern. Keeping all elements of one chromosome in one container is beneficial for most algorithms as data is usually processed chromosome-wise such that memory-locality becomes a major factor for algorithm runtime.

Secondly, start, stop and strand information are stored in three large (memory-mapped) arrays for each chromosome. Thus, instead of storing the elements associated with the forward and backward strand in two separate containers, they are all stored together. This requires some filtering when strand-specific queries are processed. However, if separate containers were used, elements associated with 'both' strands or elements having 'unspecified' strand location would need to be stored either in two further containers (also resulting in additional query processing) or would need to be duplicated in the forward and backward strand and annotated in some fashion to denote their special status.

To store complex coordinates in these containers, they are first split into their 'atoms' which are stored as individual elements in the start, stop and strand arrays. A further array of `boolean` values is used to store whether a given 'atom' is the first one of a complex coordinate, or the continuation of a previously started complex coordinate. Reconstruction of the complex coordinate model is done transparently by the flyweight access object.

The structure of `AbstractLocusChromosome` and the classes it uses internally for storage are shown in figure 7.13. The flyweight class encapsulating access to the coordinates is `AbstractLocusGeneticCoordinate`[2]. Different chromosome types storing different chromosome-associated data are derived from these two classes. They all inherit the methods implemented in the abstract base class, which provide the following functionality:

AbstractGeneticCoordinate

- **Chromosome, start and end:** In addition to this basic information, strand-specific functions are implemented as `getUpstreamCoordinate` and `getDownstreamCoordinate`.
- **Length:** The length of the coordinate is accessible as *spanned* length (i.e., end-start+1) as well as *covered* number of bases.
- **Coordinate Model:** The coordinate model can be accessed as structured model or as list of consecutive *coordinate atoms.*
- **Overlap:** The number of overlapping bases can be computed between a coordinate and a range of positions, or between two coordinates (taking into account the complex coordinate models).
- **Distances:** The distance can be computed between a coordinate and another coordinate, optionally ignoring the case when the two coordinates are on opposite strands. Furthermore, genetic coordinates define the concept of *Anchors* which can be used for distance computations. The distance between any given position and a genetic coordinate can be computed based on different anchors: `FROM` and `TO` are strand-independent, their strand-specific equivalents are `UPSTREAM` and `DOWNSTREAM`, further anchors are `CENTER` as well as `CLOSEST`.
- **Comparisons:** The `compare` function defines an order by (1) chromosome (including species), (2) start position, (3) end position, (4) strand, (5) *coordinate atoms'* `compare` function, i.e., by (5.1) coordinate atom start, (5.2) coordinate atom end, (5.3) coordinate atom strand. Further functions for strand-specific comparisons are provided with `isUpstreamOf` and `isDownstreamOf`.
- **Further functions:** Serialization, Java's `toString` and `hashCode` functions.

AbstractLocusChromosome

- **Basic properties:** Chromosome name, species, length, number of objects stored
- **Access to coordinates:** Methods are implemented to access all objects *covering* a certain base, stretch of bases, or bases specified by a complex coordinate model. Equivalent methods are implemented to access all objects *spanning* the given positions.
- **Coverage:** The coverage (objects per base) can be computed for the whole chromosome or for a specified range of bases. Furthermore, efficient functions

[2] public abstract class AbstractLocusGeneticCoordinate<T extends AbstractLocusChromosome> extends AbstractGeneticCoordinate

exist to check if any object overlaps (resp. spans) a specified region, as well as whether a specified region is completely covered (i.e., no base has zero overlapping objects).

- **Iteration:** Associated objects can be iterated (a) in unsorted fashion, (b) sorted by their start position, (c) sorted by their end position, as well as (d) sorted by their length. Base positions can also be iterated, such that the iteration goes over (a) only positions where an object starts, (b) only positions where an object ends, (c) positions where objects start or end, or (d) positions covered by any object.
- **Cluster selection:** Starting with a given region, all objects covering that region can be extracted, together with all objects covering any of the extracted objects until no further objects can be added. This function is used by Mayday's genome browser (see section 4.3) to find all reads that need to be considered together during the layout phase of rendering. A similar function is implemented using *spanning* instead of *covering* as selection criterion.
- **Further functions:** Compaction, Java's `toString`, `compareTo`, `equals` and `hashCode` functions

7.6.9. Containers for mapped read data

The `MappingStore` class uses the containers described in the previous sections and brings them together to store all information associated with an RNA-seq experiment's reads. Since eukaryotic organisms have several chromosomes, and datasets might even contain reads mapped to different species, the efficient access methods and the flyweight implementations mentioned before needed to be extended to several chromosomes. Furthermore, several mapping positions might exist for any given read and efficient access to all mapping positions of a given read as well as to the read associated with a given mapping position are needed. Three distinct sets of identifiers are used inside the `MappingStore`: Read indices, mapping indices and coordinate indices.

In the `MappingStore`, each read is identified by its *read index* into several data structures storing the read name, the read index of the read's mate pair (if paired-end sequencing was used) as well as the read's mapping positions (as shown in figure 7.14). Each mapping position is also stored with a *mapping index* which gives access to the read index of the associated read, the start of the alignment in read coordinates, the quality value reported by the aligner and, finally, the *coordinate index* of the mapping. This coordinate index can be used to retrieve the genomic coordinates of the mapping (the lower two bytes of the coordinate index specify the `LocusChromosomeLong` instance containing the mapping, the remaining 6 bytes specify the internal index of the coordinate inside the `LocusChromosomeLong`). The reverse link is provided by the `long` values that store the mapping index with each element inside the chromosomes.

As a result, the `MappingStore` provides $O(k \cdot \log n)$ access to all k reads mapped to (covering or spanning) a given genomic region on a chromosome of length n. Starting from a read, all its mapping positions as well as the positions of the mate pair (if

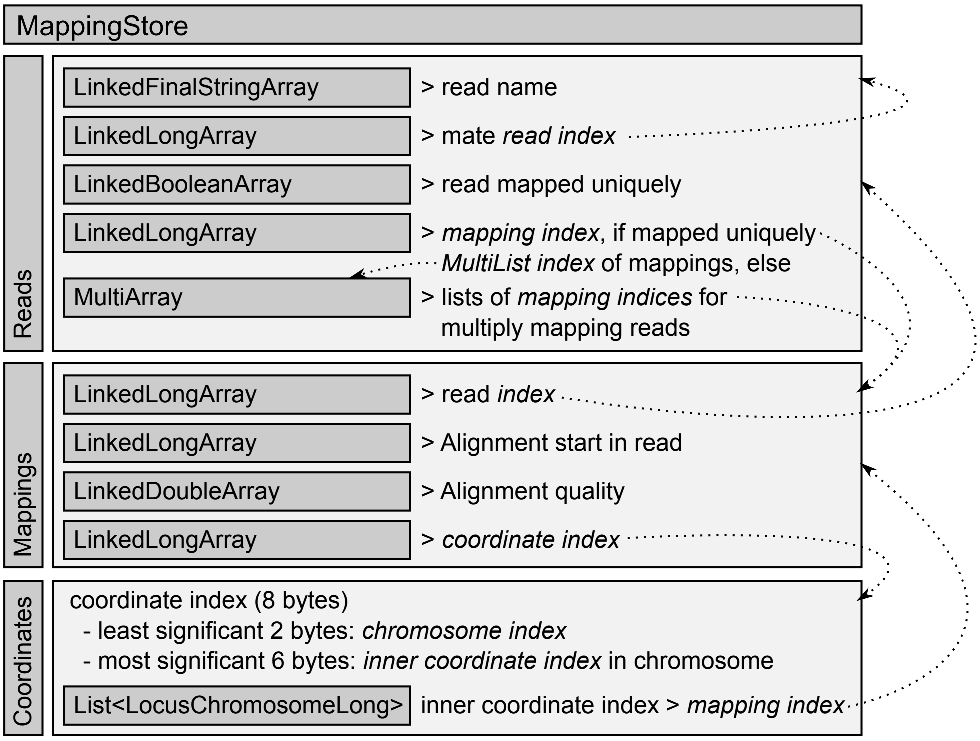

MappingStore
Reads
LinkedFinalStringArray
> read name
LinkedLongArray
> mate *read index*
LinkedBooleanArray
> read mapped uniquely
LinkedLongArray
> *mapping index*, if mapped uniquely
MultiList index of mappings, else
MultiArray
> lists of *mapping indices* for multiply mapping reads
Mappings
LinkedLongArray
> read *index*
LinkedLongArray
> Alignment start in read
LinkedDoubleArray
> Alignment quality
LinkedLongArray
> *coordinate index*
Coordinates
coordinate index (8 bytes)
- least significant 2 bytes: *chromosome index*
- most significant 6 bytes: *inner coordinate index* in chromosome
List<LocusChromosomeLong>
inner coordinate index > *mapping index*

Figure 7.14.: Internal representation of the `MappingStore` class storing information on reads and the positions in the genome that they were mapped to by an external mapping program, as well as providing fast access to all reads mapped at any genomic coordinate. Three different *indices* are used, one referencing the data belonging to one read, one referencing the data related to one mapping of a read, and one related to the mapping coordinate associated with that mapping. Dashed arrows indicate the direction of access, e.g., the *read index* stored for each mapping can be used to access data about the read while the *mapping index* stored with each read can be used to access data about the mapping.

Table 7.5.: `MappingStore` memory consumption analysis

		without MM		with MM and compaction		
Data element	type	main memory	per read	main memory	mapped	per read
— Data stored for each read						
read name	String	643,360,456	#128.0308	†1,280,504	140,421,232	28.1991
maps unique	bool	642,056	0.1277	642,056	–	0.1277
mapping index	long	40,257,616	8.0114	†80,504	*20,060,176	4.0081
— Data stored for each mapping						
read index	long	40,257,616	8.0114	†80,504	*20,060,176	4.0081
alg. start	long	40,257,616	8.0114	†80,504	♮5,015,044	1.0140
alg. quality	double	40,257,616	8.0114	†80,504	40,120,352	8.0001
coord. index	long	40,257,616	8.0114	†80,504	*20,060,176	4.0081
— Coordinate data						
	icsc	571,948,261	+4.7425	16,757,136	184,459,264	+1.6685
Totals				19,082,216	430,196,420	
		1,417,238,835			449,278,636	

Memory consumption of a `MappingStore` object storing and indexing 5,025,044 single-end reads of length 36bp with 28 character identifiers mapping uniquely to the human genome, measured with with and without memory mapping (MM). With memory mapping enabled, compaction is also used (as described in section 7.6.4). alg, alignment; coord, coordinate; icsc, `IndexedChromosomeSetContainer` mapping indices to genomic coordinates and vice-versa (see figure 7.14). #including 28 unicode characters of 2 bytes each plus 64 bytes `String` object overhead. †including memory required for storing the first block of 10,000 elements (see section 7.6.4). *using four bytes per `long` value as the maximal index to reference in this example is smaller than 2^{64}. ♮using one byte per `long` value as all alignments start at the first base. +per read and chromosome.

available) are accessible in constant time. Furthermore, memory locality for many algorithms is achieved by grouping mapping coordinates by chromosome. Care was taken to also limit memory consumption, e.g., by only allocating memory for one mapping position per read and extending this space if necessary by adding a layer of indirection and storing a list of mapping indices in a `MultiArray` instance.

To illustrate the memory consumption of the `MappingStore` class, an example dataset with about five million reads of length 36bp (each with a 28 character identifier) mapped to the human genome was loaded into SeaSight and the actual memory consumption (both in main memory as well as in on-disk memory-mapped files) was measured using the Eclipse Memory Analyzer (`http://eclipse.org/mat/`). Measurements were taken with and without memory mapping enabled, to allow for comparison of the two approaches (see table 7.5). Even without memory mapping, the storage structures introduce very little overhead, for instance 0.14% in the case of the `LinkedLongArray` used for storing read quality values. With memory mapping, the main memory footprint of the `MappingStore` object is almost constant (allowing for a few bytes to be used to reference on-disk mapped data). Furthermore, the use of compaction (see section 7.6.4) significantly reduces memory requirements while retaining flexibility for storing larger datasets.

The `MappingStore` structure stores a number of read details (qualities, multiple mapping coordinates per read, read identifiers, alignment starts) that are only needed for read visualization in Mayday ChromeTracks, and for (future) implementations of expression quantification methods using read alignment quality or paired-read information. For the simpler quantification approaches such as RPKM, SeaSight offers to import mapped read data into an alternative structure with much smaller storage footprint. For the example presented here, this would reduce the total memory consumption to 175 MB, equivalent to 36.5 bytes per read (instead of 89.5 bytes/read).

7.6.10. Matrices and Vectors

Beside data from RNA-seq experiments, Mayday SeaSight was also designed to handle raw data from microarray experiments. For these, expression values can be represented by a vector of named values, where the (unique) names denote features on the array. Several microarray experiments based on the same platform (i.e., the same chip from the same vendor) can be represented as a matrix with named rows (features) and named columns (experiments), similar to the well-known *expression matrix* which is the outcome of a SeaSight normalization.

Such vectors and matrices can also be used to hold the intermediate SeaSight data produced during the application of transformations, if the data can be regarded as numerical values associated with named features. During the implementation of Mayday SeaSight, a new framework for vector and matrix types was added to Mayday's core with the aim of simplifying the implementation of transformations. Some features of this framework were inspired by the implementation of similar types in `R`, such as summary functions (e.g., `sum`, `prod`, `sd`, `mean`, ...), per-element operations (e.g., `log`, vector-scalar addition, multiplication, ...), vector-vector operations (element-wise addition, multiplication, ...) as well as the *application* of custom functions. For the latter, the `R` syntax "`apply( matrix, dimension, function, extra arguments... )`" was closely followed, using Java's `Reflection` system to allow calls of the form "`Matrix.apply( dimension, object, 'function', extra arguments... )`" where the `object` implements a function named `'function'`.

Matrices can be created by providing a number of rows and columns, or by binding vectors together (similar to `R`'s `cbind` and `rbind` functions). Furthermore, Mayday's vectors and matrices support sorting, ranking, permutation, indexing by row/column name, $O(1)$ transposition, memory-efficient creation of shallow clones and submatrices, as well as some of the usual algebraic operations such as matrix multiplications. The matrix/vector framework was designed such that the internal representation of the vector resp. matrix data can be defined in different ways by subclasses of the respective abstract classes, allowing for very flexible implementations. For example, matrices can be implemented as *wrappers* around data stored in a completely different data structure, or the values of matrix cells can be computed on the fly during access to provide a (read-only) matrix of (e.g., for the identity transformation). General features, such as transposition, permutation and sorting are implemented independent of the underlying data structure as modifications to the indices during access. Thus, several *views* of the same data can be created (e.g., a view represent-

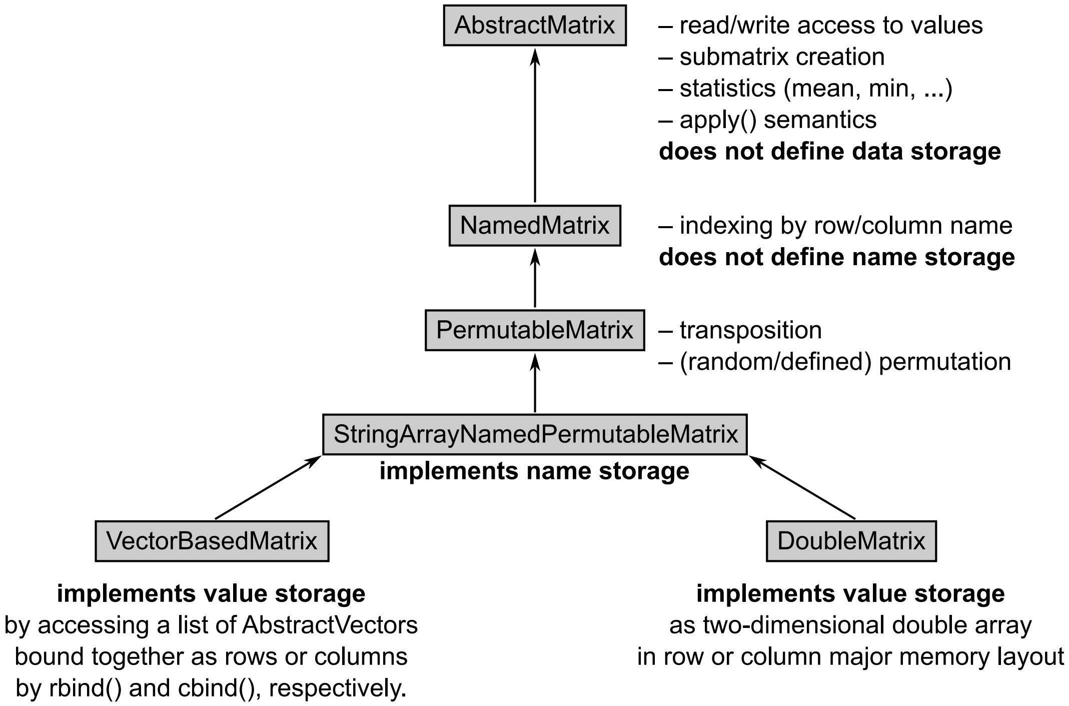

AbstractMatrix
– read/write access to values
– submatrix creation
– statistics (mean, min, ...)
– apply() semantics
does not define data storage
NamedMatrix
– indexing by row/column name
does not define name storage
PermutableMatrix
– transposition
– (random/defined) permutation
StringArrayNamedPermutableMatrix
implements name storage
VectorBasedMatrix
implements value storage
by accessing a list of AbstractVectors
bound together as rows or columns
by rbind() and cbind(), respectively.
DoubleMatrix
implements value storage
as two-dimensional double array
in row or column major memory layout

Figure 7.15.: Class hierarchy of Mayday's matrix framework showing the responsibilities of the different classes.

ing the original matrix and a permuted view on a transposed sub-matrix) without the need to store the same data twice. This approach also allows to efficiently perform computations that require sorting, changing values in the sorted matrix, and finally restoring the original order. Figure 7.15 gives an overview of main classes implementing the matrices.

7.7. Implementing efficient algorithms

Even when using primitive data types in the optimized structures described above, SeaSight still requires a lot of memory during application of the transformations to the raw data. The extendable plugin architecture at the same time allows *and* requires programmers to write their transformations without taking into account other transformations that were applied to the data before, or will be applied at a later stage during normalization. The only exception to this are the data properties discussed in 7.5.2 which are used to check whether a specific transformation is applicable to the data available as its prospective input and to predict the state of the data after applying said transformation.

This independence of the transformations results in some implementation problems that are addressed at the level of the SeaSight framework. The aim is to avoid needless (re-)computations wherever possible without using too much memory for the storage of intermediate results which might never be needed again. This is achieved by two strategies, namely caching (described in 7.7.1) and lazy evaluation (see 7.7.2). However, transformation authors still need to take care to implement their methods in an efficient manner. Some useful considerations are mentioned in section 7.7.3.

7.7.1. Caching

At some points, data needs to be converted from one representation into another. These conversions can result in tiny objects (such as when creating a flyweight object representing a certain mapped read or genomic coordinate) or in very large objects (such as when an unsorted list of genetic coordinates is converted into an indexed object allowing by-base access to the contained coordinates). Transformation programmers accessing objects usually do not know whether the access function requires such an on-the-fly conversion, nor should they have to think about this question.

Rather, it is up to SeaSight's data structures to ensure that objects resulting from such conversions are kept in memory long enough to make repeated access efficient (by preventing the recreation of the conversion object for each access). At the same time, large conversion results must be dropped from active memory when they are no longer needed to free the space they occupy.

Basically, SeaSight uses two methods to accomplish this aim: Firstly, `SoftReferences` are used to keep pointers to conversion objects as long as possible. `SoftReferences` are a kind of Java pointers which *do not* prevent garbage collection of the pointed-to object. Java offers several different `Reference` types upon which an ordering is defined [64]:

$$\begin{array}{c}\text{Strong}\\ \text{reference}\end{array} \succ \begin{array}{c}\texttt{Soft}\\ \texttt{Reference}\end{array} \succ \begin{array}{c}\texttt{Weak}\\ \texttt{Reference}\end{array} \succ \begin{array}{c}\texttt{Phantom}\\ \texttt{Reference}\end{array} \succ \text{(no reference)}$$

Consistent with this ordering, the Java standard defines the concept of *Reachability* from the point of view of the garbage collector: Objects which can be accessed by at least one strong reference are *strongly reachable*, otherwise they are *softly reachable* if a `SoftReference` points to them, otherwise they are *weakly reachable* if referenced by a `WeakReference` object, and so on:

$$\underset{\text{Object is accessible}}{\underline{\begin{array}{c}\text{strongly}\\ \text{reachable}\end{array} \succ \begin{array}{c}\text{softly}\\ \text{reachable}\end{array} \succ \begin{array}{c}\text{weakly}\\ \text{reachable}\end{array}}} \succ \underset{\text{inaccessible}}{\underline{\begin{array}{c}\text{phantom}\\ \text{reachable}\end{array}}} \succ \underset{\text{non-existent}}{\underline{\text{unreachable}}}$$

A more detailed explanation is given below:

- **Strong reference:** The corresponding object is directly referenced via a variable, it will not be garbage-collected.
- **`SoftReference`:** During garbage collection, the JVM may remove the referenced object from memory. However, the Java standard "encourages" implementations to only clear the object in the case when not clearing it would result in an `OutOfMemoryError`. Furthermore, more recently accessed referents should be cleared after other softly referenced objects. If the referent is removed from memory, all `SoftReference`s referring to it will be cleared.

- **`WeakReference`:** During garbage collection, all objects that are only weakly referenced will be removed from memory and all `WeakReference`s pointing to the object are cleared.
- **`PhantomReference`:** The corresponding object *cannot* be accessed via the reference object (the `get` method always returns `null`) and thus will never become strongly, softly or weakly reachable again. `PhantomReference`s can be used to have the garbage collector notify the user's code about the fact that the referred object has been found to only be phantom reachable and will be removed from memory. This notification can be used to perform further 'pre-mortem' cleanup operations depending on the object referred to. After such actions are completed, the reference must be *cleared*, which will change the referent from being phantom reachable to being unreachable.
- **No reference:** Objects that are unreachable are reclaimed by the garbage collector and their memory is returned to the Java Heap memory pool.

Reachability is a transitive property. All reachable Java `Thread`s are by definition *strongly reachable*. For any reachable object O, there exists a directed path of references $p(O)$ connecting it to one of the root `Thread` objects. The *reachability* of O is defined as the minimum reachability of all its predecessors on $p(O)$. For example, if $p(O) = (\texttt{Thread}_i\text{:strong}, O_1\text{:strong}, O_2\text{:weak}, O\text{:strong})$, then O is *weakly reachable* from the garbage collector's point of view. Obviously, the graph of references can contain cycles. Only references belonging to the greedy (in terms of reachability) spanning tree that starts at the respective `Thread` object are considered for the determination of *reachability*.

Using `SoftReferences`, SeaSight's first strategy is to try to hold on to conversion intermediates for as long as possible, i.e., until the Java VM determines that new objects require heap space to be freed, and the cached objects are claimed by the garbage collector.

A second strategy uses more sophisticated caching for conversion objects that are created based on an additional parameter (e.g., based on the name of a chromosome an object might be created containing all genetic coordinates located on that chromosome). If the same parameter is supplied repeatedly, the object already created in the first invocation (stored via a `SoftReference` as described above) is returned, otherwise the cache is cleared and a new conversion object is created.

7.7.2. Lazy evaluation

Many transformations can be implemented in a *lazy* fashion, especially such transformations that only affect individual values in an experiment and do not need the context of the other values to compute the result. For instance, consider the logarithm. To compute a logarithm of a set of numbers, each number can be dealt with individually, without knowledge of the full set. In Mayday SeaSight, many transformations use *lazy evaluation*, among them the log transformation and most read-based measures of transcriptional strength, such as RPKM, RPM, RPK, DCPM and the naïve counting method. When such a transformation is applied to the data, a *wrapper* function is added such that as soon as the data is accessed, the transformed value is computed and returned. The benefit of the lazy approach is that only those values need to be transformed that are actually requested by a downstream transformation. In the case of the read-based measures of transcriptional strength, this approach is the only feasible method since computing these measures requires the actual genomic coordinates of the transcript to evaluate. Computing RPKM values for all *possible* transcript coordinates is obviously infeasible and deferring the computation to the point where the coordinates of interest are known is the only sensible implementation choice.

For the purpose of illustration, consider the computation of $\log_2$ transformed RPKM values: To get the final value for transcript t with coordinate $c(t)$, first the `getExpression` method of the class `LogTransformedExperiment` is called with $c(t)$ which calls the same method from `RPKMExperiment`. In this method, the number of reads overlapping with $c(t)$ is obtained from the underlying `LocusReadCountExperimentData` object, normalized with the total exon length of $c(t)$ and the number of reads in the experiment (in millions). The resulting value is returned, the logarithm is computed and the final result handed back to the calling function.

7.7.3. Considerations for transformation authors

Authors using the efficient data structures provided by SeaSight still need to take some considerations into account to make sure that their implementation can be executed in an efficient manner. Besides the usual Java-specific efficiency improvements, such as avoiding the creation of millions of short-lived `Objects` if existing instances can be re-used, *memory locality* is the most important aspect to keep in mind.

The use of memory-mapped data structures and SeaSight's caching mechanism can only be efficient if the implemented method accesses data in the correct order. This involves working on data in a chromosome-by-chromosome fashion, as chromosomes are the top-level object with respect to the distribution of data to the memory-mapped files. Furthermore, caching of conversion intermediates is also handled with the assumption that most of the time, consecutive requests will target the same chromosome and that chromosome switching is a rare event.

Secondly, as read data is usually imported from sorted alignment files (such as BAM files), algorithms accessing locus-associated objects should try to access proximal objects first, before moving on to more distant objects. This allows the virtual memory manager to work on one page after another, instead of requiring it to constantly move pages in and out of physical memory.

As an example of how these two considerations can be realized in an implementation, consider a transformation that computes expression values from aligned reads and a set of genomic locations of genes. An efficient implementation would first sort the gene locations by chromosome to exploit the benefits of SeaSight's caching mechanisms, and then by their base coordinate to optimally use the memory-mapped access to the mapped read data. Based on the sorted genes, reads overlapping the genes' locations can now efficiently be retrieved.

Table 7.6.: Human liver and kidney sample sequencing data

Sequencing run		reads	mappable	%	alignments/read
Kidney	SRR002320	39,221,626	12,741,216	32.49	1.74
	SRR002324	17,292,434	7,314,865	42.30	1.86
	SRR002325	27,137,793	9,184,290	33.85	1.80
Liver	SRR002321	54,797,551	17,039,752	31.09	1.81
	SRR002322	18,437,696	7,251,171	39.33	1.93
	SRR002323	14,761,931	6,604,737	44.07	1.28
Totals		171,649,031	60,136,031	35.03	1.75

7.8. Application example: Human kidney vs. liver tissue transcriptomes

As an example for the application of Mayday SeaSight, data from a comparative study of RNA-Seq and microarray experiments was used [105]. In this study, total RNA from liver and kidney samples of a single human male were extracted and each sample was hybridized to three Affymetrix HG-U133 Plus 2.0 microarrays as well as sequenced on multiple lanes of an Illumina Genome Analyzer. Of the technical replicates reported in the original publication, only data from three sequencing lanes per tissue are used here.

Raw read data was downloaded from the NCBI's short read archive (`http://www.ncbi.nlm.nih.gov/sra/`, accession numbers `SRR002320`–`SRR002325`) and mapped to the human genome (NCBI version 37.2) using Bowtie [95] with a maximum of two mismatches and excluding reads with more than three mapping locations (using the options `-v2 -m3 -k3 --best --tryhard --sam`), resulting in six SAM files (see table 7.6). Microarray data was obtained from GEO [10] in Affymetrix CEL file format (accessions `GSM279060`–`GSM279065`). The array description (CDF) file for the HG-U133 Plus 2.0 microarray was downloaded from Affymetrix' website, as were the 604,258 probe sequences, which were also mapped to the human genome to allow for locus-based data integration as described in section 7.4.6. 525,721 probes could be mapped uniquely, covering 53,100 probesets (of a total of 54,675).

Array data and mapped reads were imported into SeaSight. Arrays were normalized using Mayday's implementation of the optimized RMA method [62] consisting of RMA background correction, computing the logarithm to base two, quantile-normalizing the experiments, and applying the median polish method to summarize probe values into probeset expression values. To test the effect of different methods on the comparability of RNA-seq and microarray data, sequencing data was processed using several different pipelines (see figure 7.16): First, the 'pseudocount' transformation was applied to ensure that each queried region contained at least one (synthetic) transcript. This removes the problem of zero expression which complicates the computation of the logarithm later. Secondly, expression was quantified using either the genomic locations of the microarray probes as query set, or the

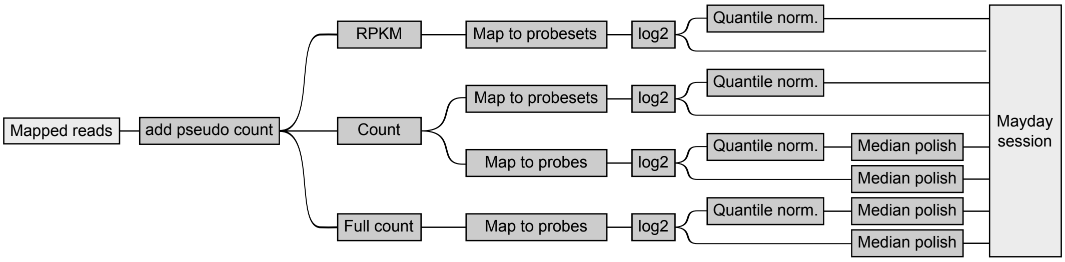


Figure 7.16.: Transformation pipelines used for the analysis of the human transcriptome RNA-seq data [105]. A total of eight different normalized `DataSets` were produced based on three different expression quantification methods, quantification on the level of probesets or on the level of probes with subsequent summarization into probeset expression values, and an additional quantile normalization step. See text for details.

genomic regions covered by the probesets (from first to last probe). The logarithm (base two) of the resulting expression values was computed. When using the probe regions, the median polish method was applied to compute probeset expression values comparable to those of the microarray. Thus, the sequencing data contained expression values for 53,100 probesets. These four processing pipelines (probeset RPKM, probeset count, probe count, probe full count) were run both with and without an inter-experiment quantile normalization step which, depending on the set of query regions, was applied either as the final normalization or directly before summarization (exactly as in the RMA procedure), to test its influence on the comparability of the different platforms.

For the primary expression quantification from read data, four different methods were used:

- **RPKM on probesets.** The expression is computed as the number of reads covering at least one base of the probeset region, divided by the region's length (in kilobases) and the total number of reads (in millions). From this, the logarithm (base 2) is computed.
- **Count on probesets.** Reads in the probeset region are counted and the logarithm (base 2) is computed.
- **Count on probes.** The number of reads covering at least one base of each probe's region (25bp) is counted. The median polish method is used to summarize based on the logarithms of these numbers.
- **Full count on probes.** The number of probes covering all 25 bases of the probe region is counted. The expectation is that this most closely mimics the biochemical process of hybridization. The median polish method is used to summarize based on the logarithms of these numbers.

Parsing the SAM files (32GB) required 101 minutes, resulting in 3.8GB of memory-mapped files. Executing the transformation matrix required 1,5 h for the array data and the four sequencing `DataSets` using quantile normalization (4GB Java Heap space, working on a single 2.5GHz core). If normalizing without quantile normalization, the time required is slightly shorter. The transformation matrix was stored

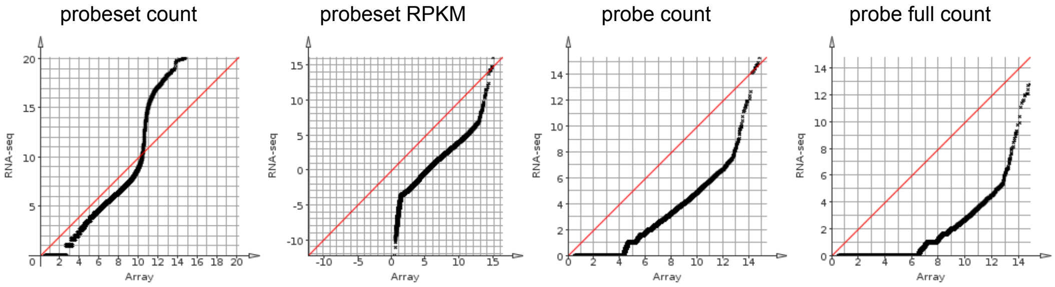


Figure 7.17.: Quantile-quantile plots comparing the distributions of logarithmic expression values obtained by processing RNA-seq data with one of four methods (see text) with the distribution of expression values obtained from a microarray experiment.

to the SeaSight file format in six minutes, resulting in a 626MB file which could be loaded again in under five minutes. Each of the two final Mayday snapshot files containing five normalized `DataSets` amounted to 25MB and can be loaded in 40 seconds.

For each of the resulting `DataSets`, four questions were addressed:

1. What are the differences between the distributions of the expression value of the samples (i.e., expression matrix columns) when comparing RNA-seq to microarray data?
2. What is the correlation between fold-changes derived from either technology?
3. How many probesets' expression values (expression matrix rows, per tissue) follow a normal distribution?
4. How large is the overlap between statistically significantly differentially expressed (SDE) genes detected by either technology?

7.8.1. Distributions of expression values per sample

To compare the distribution of transcript expression values in a sample analyzed using RNA-seq with the distribution in a traditional microarray sample, quantile-quantile plots [178] were created, comparing one sequencing sample (`SRR002320`) with the corresponding array sample (`GSM279060`) for each of the four normalization strategies without quantile normalization (see figure 7.17). Using quantile normalization, the plots showed almost identical shapes (data not shown).

The probeset count method, which does not normalize by the length of the transcripts, clearly overestimates the expression strengths of highly expressed transcripts in comparison to the microarray data, while the RPKM method which explicitly models the transcript length dependent bias achieves a more linear correlation with the distribution of the array expression values. Here, it is clearly visible that even extremely lowly expressed transcripts show a high baseline expression due to background noise on the microarray.

Table 7.7.: Fold-change correlations between RNA-seq and microarray data

	without quantile normalization		with quantile normalization	
	all	only expressed transcripts	all	only expressed transcripts
Probeset count	0.75	0.87 (13984 transcripts, 26.3%)	0.75	0.88 (11200 transcripts, 21.1%)
Probeset RPKM	0.75	–	0.75	–
Probe count	0.77	0.95 (1620 transcripts, 3.1%)	0.77	0.95 (1132 transcripts, 2.1%)
Probe full count	0.64	0.93 (293 transcripts, 0.6%)	0.63	0.94 (189 transcripts, 0.4%)

Correlations between the fold-change values computed from normalized microarray data and expression quantified from RNA-seq data using different quantification methods (see text) either with or without a final quantile normalization step. Correlations were computed on all 53,100 transcripts, or on the set of expressed transcripts only, regarding as expressed each transcript with an expression value of at least five in at least one sample, corresponding to 32 mapped reads.

The two probe-based quantification approaches show a similar profile as RPKM, because they also (implicitly) control for transcript length, by only considering a fixed-length subsequence of length 325bp (13 probes of 25bp each) for each transcript. On the other hand, this means that the probe-based methods only use a small part of the sequencing data, resulting in expression values of zero for lowly expressed transcripts. Only using reads which completely overlap a probe's region further increases this problem.

Thus, to get an accurate representation of transcript expression, one should either use a quantification method that corrects for transcript length explicitly (such as RPK or RPKM), or drastically increase the depth of sequencing such that the probe-based methods have enough data to work on. Obviously, the latter choice would result in a huge amount of sequencing reads which, after mapping, are not used for quantification at all, i.e., a huge amount of data would be expensively produced, processed, and then discarded. In chapter 8, an alternative to the common RNA-seq protocol is presented together with a specialized processing algorithm, which avoids this wasteful data generation while implicitly controlling for transcript length.

7.8.2. Fold-change correlations

The correlations between the expression fold-changes derived from the 2 × 3 technical array replicates with those derived from the 2 × 3 sequencing replicates was computed. These correlations are quite high (see table 7.7), as observed in the original publication of Marioni *et al.* [105]. Even without correcting for the library size (either by using RPKM or by the additional quantile normalization step), the correlations are above 0.75 for the probeset methods as well as for the probe count method. The latter shows the highest correlation, possibly because it is quite close to the process used to compute expression values from the array data. The full count method shows a smaller correlation, most likely due to the fact that only very few reads are actually counted for the present data, as the read length (36bp) was not much longer than the probe length (25bp). In the 'count' method, a read is counted when it overlaps with *any* of the bases of the probe, thus it will be counted if its start

Table 7.8.: Transcripts with normally distributed expression values

	without quantile normalization			with quantile normalization		
	transcripts	%	failed tests	transcripts	%	failed tests
Probeset count	48965	92.2	25	50931	95.9	0
Probeset RPKM	53000	99.8	38	52992	99.8	28
Probe count	50369	94.9	34	51305	96.6	36
Probe full count	51658	97.3	25	52227	98.4	40
Array data				52956	99.7	1

The distributions of transcripts' expression values per tissue were tested for normality using the Shapiro-Wilk test, for a total of 53,100 transcripts. Transcripts with p-values for both tissues above 0.05 were considered to be normally distributed, otherwise they were counted as not following a normal distribution. The `R` implementation of the Shapiro-Wilk test failed with an internal error for a small number of transcripts. These were regarded as being not normally distributed.

maps in the region from −35 upstream of the probe's start to +25 bp downstream, i.e., a region of 60 bp in size. The 'full count' method requires the read to map such that it completely overlaps the probe, i.e., its start has to map in the region of −11 upstream to 0 (coinciding with the probes's start position), a region of only 12 bp in size.

When only considering transcripts with a certain baseline expression in at least one sample, the correlation of the fold-changes rises considerably. As suggested in the original publication, the threshold was set such that only transcripts were considered with at least 32 mapped reads (equivalent to a $\log_2$ expression of five). As in the original publication, the correlations observed for this smaller set were significantly higher than for the full complement of genes, supporting the conclusion that the technologies mostly differ regarding the lowly-expressed genes. Such differences are likely due to microarray background noise and the sampling of too few reads from very lowly expressed genes in RNA-seq experiments.

Interestingly, the probe-based quantification methods profit more strongly from this than the probeset based method. However, this might be due to the small number of transcripts with sufficiently high expression. This test was not performed for RPKM, where expression values can not be directly related to the number of reads mapped.

7.8.3. Normality of the distributions of transcripts' expression values

An argument often advanced as justification for highly sophisticated new statistical methods to find significantly differentially expressed (SDE) genes is that the expression values obtained from RNA-seq experiments do not follow a normal distribution. To test if this was the fact for the data used here, the expression values of each transcript in each tissue were tested using the Shapiro-Wilk test for normality [144] as implemented in `R` (via Mayday RLink).

A transcript was counted as being 'normal', if both its expression values in the liver sample and its expression values in the kidney sample were found to come

Table 7.9.: Overlap between the sets of differentially expressed transcripts

	differentially expressed transcripts				overlap in percent	
	seq total	only seq	overlap	only array	of seq	of array
without quantile normalization						
– Probeset count	1917	596	1321	1867	68.9	41.4
– Probeset RPKM	1893	583	1310	1878	69.2	41.1
– Probe count	1811	431	1380	1808	76.2	43.3
– Probe full count	1519	535	984	2204	64.8	29.9
with quantile normalization						
– Probeset count	1961	624	1337	1788	68.2	41.9
– Probeset RPKM	1922	610	1312	1876	68.3	41.2
– Probe count	1967	521	1446	1742	73.5	45.4
– Probe full count	1763	744	1019	2169	57.8	31.9

Transcripts were considered differentially expressed (DE) if their Rank Product *pfp* value (100 permutations) was below 0.05. A total of 53,100 transcripts were tested, of which 3,188 were called DE based on the microarray data.

from (possibly different) normal distributions. For each vector of three replicates, this was assumed to be the fact if either all values were identical (a case which the `shapiro.test` function will not evaluate), or the p-value for rejecting the null hypothesis was larger than 0.05. Of 53,100 transcripts, at least 92% were found to contain normally distributed expression values (see table 7.8).

As there were only three replicates per tissue, the basis for this test is very thin. However, the results still suggest that for this particular data, the normal distribution might be a valid assumption.

7.8.4. Overlap of the sets of differentially expressed transcripts

Despite the high number of transcripts with normally distributed expression values, the nonparametric Rank Product method [30] was used to determine DE transcripts, both from the microarray data as well as from the `DataSet`s created from RNA-seq data. Mayday offers a memory-efficient implementation of that method. For n transcripts, k permutations and two classes with m_1 and m_2 replicates, respectively, it requires $O(2n)$ memory as opposed to the $O(nm_1m_2 + nk)$ of the original algorithm without an increase in runtime requirements. This algorithm was recently also included as an improvement to the original `RankProd` package [71] for `R`. Here, it was run with 100 permutations on each `DataSet` and all transcripts with a $pfp < 0.05$ were selected.

To allow for a clearer discussion of the following results, the array data is considered as a ‘gold standard’. Transcripts called as DE based on the RNA-seq data can thus be classified either as *true positives* (TP) if they are also called as DE based on the array data, as *false positive* (FP) otherwise, and transcripts only called as DE based on the array data can be classified as *false negatives* (FN).

The largest overlap between the set of DE transcripts from the array data and that from the sequencing data is found with the 'probes count' method (see table 7.9), which is not surprising given that it involves expression quantification and summarization steps quite similar to the ones used for the array data. Again, one might expect an even larger agreement for the 'full count' method, because it models the hybridization process on the array more closely. This expected result is probably not seen due to the small number of reads that can be used for quantification with this method (as discussed in section 7.8.2).

Testing only for the overlap of DE transcripts which also have an expression value ≥ 5 in at least one sample, or which additionally also have an absolute fold-change of at least three between tissues, reduces the number of false positives, but also greatly increases the number of false negatives, simply because the number of DE transcripts in the RNA-seq data gets smaller (data not shown). One might also argue that an expression level of five in the array data is not sufficiently high to ensure that all DE transcripts are actually expressed above the level of the background noise. Increasing the minimum expression threshold for the array data reduces the number of false negatives (as seen from the sequencing data), but also reduces the number of true positives (data not shown).

Interestingly, the number of transcripts detected as differentially expressed in the RNA-seq data is much smaller than if using the array data, regardless of normalization method, ranging from 48.7% to 63.1% in size. This is likely due to the higher variance in the expression values computed from the mapped reads, which could be alleviated either by using more replicates, or by advances in sequencing technology (including protocols). To test whether the variances might be a problem, the three sets of transcripts (TP, FP, FN) were analyzed separately by computing the quotient of the coefficient of variation divided by the fold-change. This results in a measure of observed variation per fold-change detected, irrespective of the absolute expression strength. The results clearly show (figure 7.18) that the false negatives are not detected as DE in the sequencing data because of their high variance in relation to their fold-changes. DE transcripts found based on the sequencing data, but not found based on the array data also show a somewhat elevated variance, yet much less pronounced.

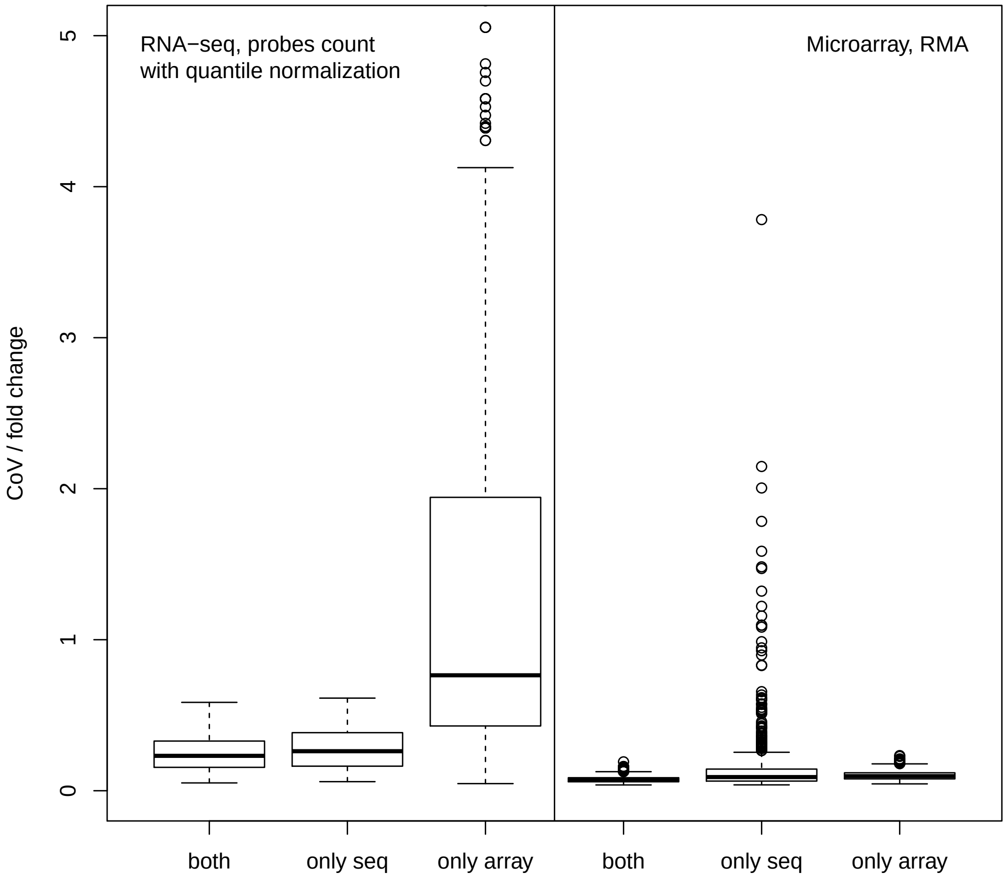


Figure 7.18.: Variance of differentially expressed (DE) transcripts found based on the sequencing data, the array data, or both, and computed, based on the expression values of each technology, as the ratio of the coefficient of variation (standard deviation divided by mean) and the fold-change. This measure shows the relationship between variance and fold-change, normalized by the relative expression strength.

8. Passage: Efficient RNA-seq clustering

Mayday SeaSight can quantify expression and build expression matrices from raw data in many formats obtained using different technologies such as microarrays or RNA-seq, but also data from specialized protocols. One such protocol is Passage, which will be introduced in this chapter.

Microarrays have been widely used for expression analyses. However, there are limitations to this technology. Most importantly, probe design requires previous knowledge about the transcripts of interest. RNA-seq does not have this limitation: By *mapping* the sequenced reads against a reference genome sequence, RNA-seq analyses can be used for *de novo* transcriptomics, i.e., for measuring the transcription without prior definition of interesting targets.

RNA-seq experiments usually involve the sequencing of millions of short reads covering the whole transcriptome (after suppression of highly abundant ribosomal RNAs from the sample). Besides expression quantification, this wealth of data can be used to address a very wide range of research questions, spanning detection of previously unknown coding as well as non-coding transcripts, reannotation of known transcripts, splicing analysis, and characterisation of single nucleotide polymorphisms (SNPs) as well as larger mutations.

However, all these analyses depend on the mapping of sequenced transcripts against a known reference sequence for the organism (or organisms) under study. This dependence prevents the application of RNA-seq analyses to samples from organisms for which genome sequences are not available, or, in the field of metatranscriptomic analyses, samples for which the organism content is not known. Even though more and more genome sequences are produced, genome sequencing (and genome assembly in particular) remains very time-consuming as the advent of faster sequencing methods only managed to move the bottleneck from wet-lab work to data processing [116]. Thus RNA-seq is not easily applicable to analyses of presently unsequenced organisms. As the vast majority of organisms (above 99.95% [100]) fall into this class, this problem will not be resolved quickly.

In the case of metatranscriptomic samples and samples of unknown organisms, Blast [3] is usually employed to map the sequenced transcript fragments against the database of all known sequences. Based on the mapping results, researchers try to assess organism content and to compute a transcriptional profile for each sample. Such analyses require enormous computing resources for storing the large number of reads sequenced as well as for performing the mapping step.

Many RNA-seq experiments, however, are performed with the specific aim to detect differential expression (of protein-coding genes) between two classes of samples. For this kind of project, the sheer volume of data produced by RNA-seq is an obstacle to analysis rather than a blessing. A lot of time and effort is spent to quantify

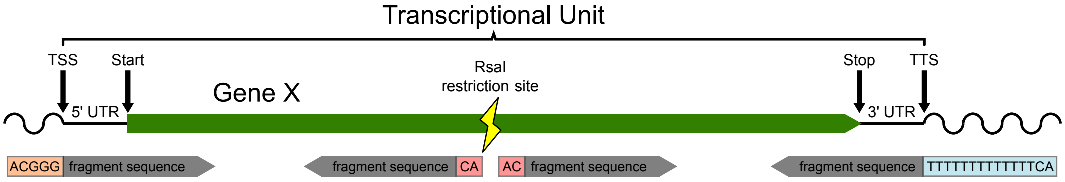


Figure 8.1.: Schematic view of fragments created by the Passage protocol from a transcriptional unit. *RsaI* restriction sites (signature `GTAC`) give rise to internal fragments starting with the `AC` prefix. Additional fragments are created downstream of the transcription start site (TSS), covering part of the 5' untranslated region (UTR) and starting with the prefix `ACGGG`, as well as upstream from the transcription termination site (TTS), covering part of the 3' UTR and starting with `ACT`^{13}.

expression based on the reads and many of the features of sequencing data (single-base resolution, detection of SNPs, etc.) are not considered at all. In addition, the (still) high cost per experiment (resp. sequencing 'run') is still a prohibitive factor.

8.1. The Passage idea

Passage is a novel protocol [75] which adds a new option to the transcriptomics toolbox. In particular, it offers a solution to the reference genome problem introduced above. In the following, the general Passage idea will be introduced with the computational challenge it presents, followed by a description of the algorithmic solution [12] to this challenge.

The Passage idea is an extension of the SAGE method [167] for expression quantification. Instead of sequencing a random subset of fragments generated from all transcribed RNA molecules (excluding most of the ribosomal RNA), the Passage protocol is designed to only sequence certain well-defined fragments from each gene, as shown in figure 8.1. Prior to sequencing, the extracted RNA is cut using the *RsaI* restriction enzyme (restriction site motif `GT·AC`), resulting in fragments starting with the bases `AC`. Sequencing primers are used that specifically bind only to such fragments. In addition, fragments in the 3' and 5' untranslated region (UTR) of the genes are also sequenced. For the 3' UTR, a primer binding to the poly-`A` tail of the transcript is used, introducing a dangling `AC` end. To capture the 5' UTR, adapters containing the `ACGGG` tag are ligated to the start of the mRNA. These UTR fragments ensure that transcriptional units which do not contain an *RsaI* restriction site are also represented in the read pool.

The Passage protocol is based on Illumina sequencing of these fragments producing reads of a fixed length. Restricting the sequencing step to a set of well-defined transcripts results in several advantages that makes Passage highly useful for certain applications. These come at the expense of some of RNA-seq's wide range of application domains. The differences between generic RNA-seq and Passage are as follows:

Advantages of Passage

- Significantly reduced data volume
- Higher multiplexing levels are possible, reducing the cost per experiment.
- Reads share a common prefix (`AC`) such that contamination (e.g., by linker sequences) can easily be detected and removed.
- Reads either overlap completely (if they are from the same site), or not at all (different sites). Time-consuming overlap computation thus is not necessary and read clustering can be implemented very efficiently, as described below.
- As only reads starting at restriction sites (and the two UTR sites) are considered, the RNA-seq specific length bias is no problem for Passage. Instead of generating reads proportional to the expression strength *and* the length of the transcript, reads are only generated proportional to the expression strength. Thus normalization of the resulting expression estimates is less complicated than for generic RNA-seq data where a nonlinear dependence of read counts to transcript length is observed [33].

Disadvantages of Passage

- No single-nucleotide precision for the detection of transcript boundaries.
- Polymorphisms (SNPs) can only be found in very limited transcript regions, i.e., those regions that are directly up- resp. downstream of restriction sites and are thus covered by fragments.
- As SNPs, splice sites can only be detected in the regions covered by fragments. A reference sequence is required and splice sites can be discovered by mapping the fragments to the genomic sequence and finding fragments that only map partially because they span an intronic region.
- Distinguishing different isoforms of one gene is not straightforward because the different fragments originating from one gene do not overlap (except in some rare cases where the distance between restriction sites is less than twice the sequenced fragment length). Mapping fragments to a reference sequence and modelling the observed expression estimates for each fragment as the sum of expression values of the different isoforms it is contained in would be one possible route to computing isoform expression.
- The protocol is currently limited to eukaryotic coding genes. A modified protocol for prokaryotes as well as for non-coding transcripts is in preparation.

Based on Passage output, analyses of differential expression are possible in different settings. If no reference genome is available, differentially expressed transcript fragments can be determined which can later be the basis for Blast searches or for primer design to fully characterise the corresponding transcripts. If, on the other hand, the reference genome of the respective organism is available, the clustered and quantified reads can be mapped using common mapping tools such as Bowtie [95] or RazerS [174]. Users then benefit from drastically reduced computational costs as clustering typically reduces the number of reads by about 85-95% (for details, see [12]).

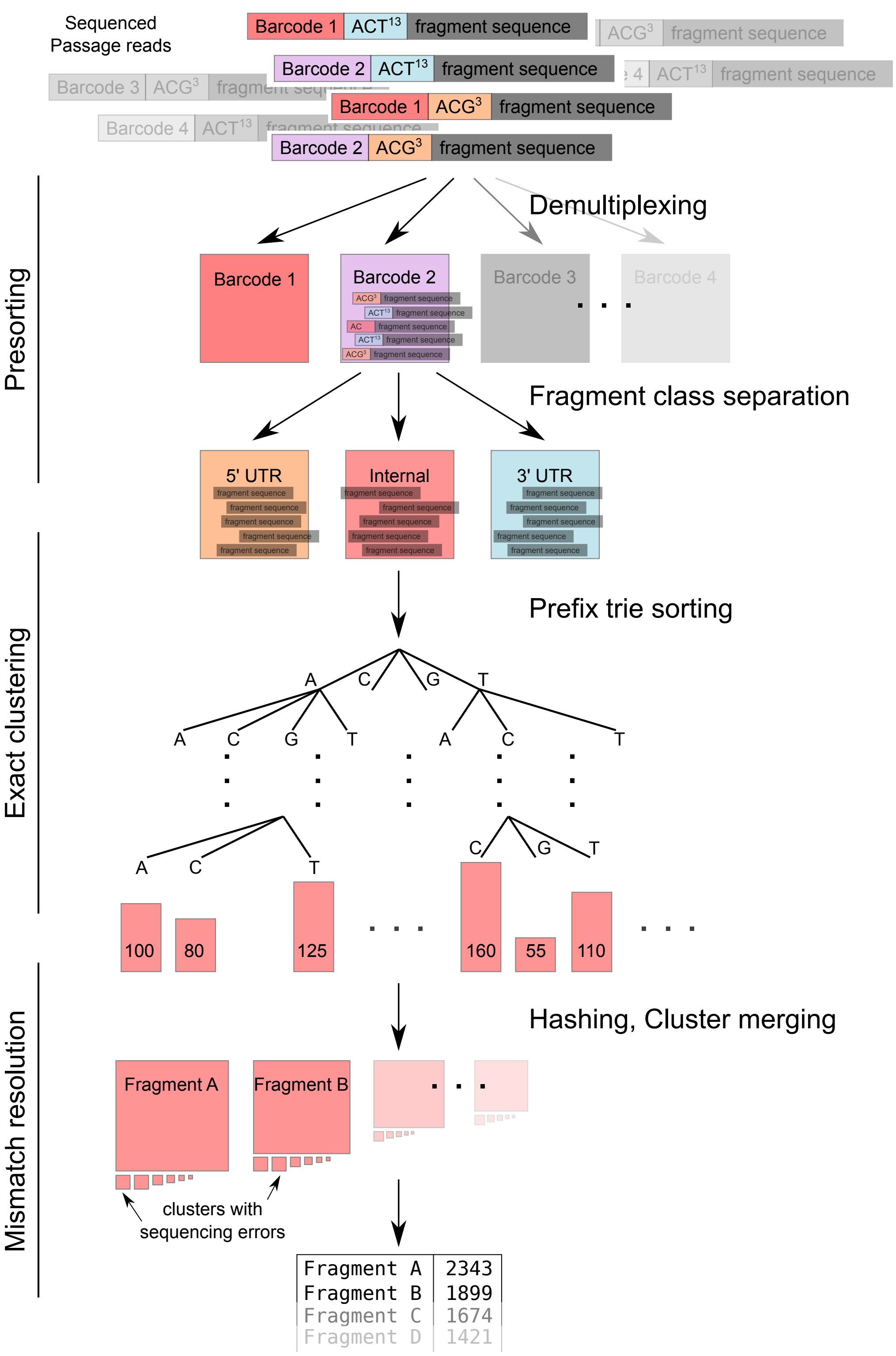
Data generated based on the Passage protocol and algorithm (see 8.2) is made up of a list of identifiers (fragment sequences) and the quantified expression level (available as absolute reads and reads per million). Such data can be analyzed using MAYDAY just like any other tabular expression data. If normalization steps are desired, data import into MAYDAY should be performed using SEASIGHT.

8.2. A mismatch-tolerant clustering algorithm for reads

The aim of the PASSAGE program is to cluster the reads resulting from sequencing of a sample treated with the PASSAGE wet-lab protocol, i.e., to count the number of occurrences of each fragment. The algorithm has to be able to allow for a certain number of mismatching bases while still being efficient. The vast majority of such mismatching bases occur due to imperfections in sequencing methods. Additionally, in diploid organisms, single nucleotide polymorphisms can cause a divergence between the maternal and the paternal allele of a gene. The special properties of the reads, in particular the fact that all reads belonging to the same fragment overlap completely, make an efficient implementation possible. Reads are clustered in three consecutive steps (see figure 8.2):

1. **Presorting:** All reads share the common prefix `AC`. Since PASSAGE drastically reduces the number of reads required for accurate transcript expression quantification, so-called *barcode sequences* can be added in front of each read. This allows for multiplexing, i.e., sequencing several samples in one *lane*. The presorting step separates reads according to their multiplexing barcode and their fragment class (3', 5', or internal) based on their prefixes. As barcode sequences are designed to be mismatch-tolerant, the presorting algorithm must allow for a certain number of mismatches in the barcodes.
2. **Exact clustering:** The majority of reads are assumed to be error-free. In the exact clustering step, a prefix trie data structure is used to collect identical copies of reads, deriving from the same transcript fragment. The number of identical copies corresponds to the expression level of the respective gene. Thus this step reduces the data volume by discarding duplicate reads and summarizing expression strength as the number of identical copies.
3. **Mismatch resolution:** Sequencing errors can arise during to base-calling (where they only affect one read), or during PCR amplification (where several, if not all reads originating from the same locus are affected). The latter problem is solved by new third-generation sequencing methods which eliminate the amplification step, as well as by specialized amplification-free protocols for second-generation sequencing [104]. The base-calling error, however, remains. In the final step of the PASSAGE algorithm, reads are clustered together based on a maximal number of allowed mismatches. This reduces the data volume by removing the effect of sequencing errors.

   The mismatch resolution step of the algorithm (see listing 8.1) starts by sorting the exact clusters according to their size, i.e., the number of reads with identical sequence. Starting with the largest such cluster, the procedure is as follows: Use the sequence of the largest cluster as *center sequence*. Find all other exact clusters that have at most k mismatches to the center sequence.



Fragment A	2343
Fragment B	1899
Fragment C	1674
Fragment D	1421

Figure 8.2.: Steps of the PASSAGE algorithm: In the presorting step, reads are sorted according the their multiplexing barcode and the fragment class. A prefix trie data structure based on the DNA alphabet is used to cluster reads together if their sequences are identical. Only nodes for existing sequences are added. The final step, mismatch resolution, uses hashing to merge clusters that likely derive from the same genomic sequence but contain mismatches due to sequencing errors. The final result is a data table summarizing expression strengths for observed fragments.

```
Let $L = \{C_1, ..., C_n\}$        // the set of exact clusters, sorted descendingly by size.
Let $M = \emptyset$                                    // the set of mismatch clusters
Create hashmaps $H_1, \ldots, H_{k+1}$          // for $k+1$ seeds ($k$ mismatches allowed)

For each $C_j \in L$ do
  Let $S(C_j)$ be the sequence of cluster $C_j$.
  Split $S(C_j)$ into $k+1$ substrings $s_1, \ldots, s_{k+1}$
  For $\ell = 1$ to $k+1$ do
    Put $< s_\ell, C_j >$ into $H_\ell$
  done
done

while $|L| > 0$ do
  Let $C$ be the largest cluster in $L$.          // $S(C)$ is the current center sequence.
  $K \leftarrow \{C\}$
  Split $S(C)$ into $k+1$ substrings $s_1, \ldots, s_{k+1}$
  For $\ell = 1$ to $k+1$ do
    Let $K_\ell$ be the set of clusters stored in $H_\ell$ at key $s_\ell$
    For each candidate $C' \in K_\ell$ do
      if $mismatches(S(C), S(C')) < k$ then
        $K \leftarrow K \cup \{C'\}$
      fi
    done
  done
  Remove all $C' \in K$ from $H_1, \ldots, H_{k+1}$
  $M \leftarrow M \cup \{mergecluster(K)\}$
  $L \leftarrow L \setminus K$
done

return $M$
```

Listing 8.1: The Passage mismatch resolution algorithm

Merge them with the initial cluster and remove the resulting cluster (and all its corresponding reads) from the data. (Note that the center sequence is stable and does not change as a result of the merging of clusters, i.e., the clusters' centroids do not move.) Thus, each cluster can be merged exactly once. Repeat until all clusters have been processed. Efficiently finding clusters with up to k mismatches to a given center sequence is achieved using hashing. Sequences are split into $k+1$ substrings. For each $i \in [1, k+1]$, a hashmap is created mapping the i^{th} substring to all clusters in which it is contained.

The assumption at the basis of this procedure is that erroneous bases are much less likely to occur than correct bases. As a result, the majority of reads should correctly reflect the genomic sequence (excluding introns) of the respective gene. In fact, we found this to be the case for most genes. For genes with a very low number of reads (e.g., less than 5), however, the correct sequence cannot reliably be identified, as erroneous reads are more likely to dominate the distribution in such cases.

8.3. Results

The results obtained using the Passage method were validated using microarray experiments (see [12] for details), showing a correlation of the expression values of 0.69, a figure that is comparable to the correlations reported between microarray and conventional RNA-seq experiments elsewhere (for an example, see Fu *et al.* [54] who found a correlation of 0.67).

As expected, summarizing expression values greatly reduces the size of the dataset by removing redundancies. Observed average reduction rates varied between 85% for reads of length 76bp and 94% for shorter 40bp reads, if two mismatches are allowed during mismatch resolution (see [12] for details). It is to be noted, however, that longer reads are expected to have a higher number of mismatches. This has several reasons: Firstly, a constant error rate *per base* automatically results in a larger number of mismatches for reads with more bases. And secondly, per-base error rates for the Illumina sequencing technology are not uniform but increase over the length of each read with each additional base sequenced. These error profiles are not easy to implement in a program like Passage, however.

Assuming position-independent and equal error rates for both read lengths, one has to allow for four mismatches for the 76 bp reads. To complement the results already presented, Passage was run with this higher number of allowed mismatches using the same eight datasets of 76bp reads as in the original publication. The resulting data volume reduction was in the range of 84% to 94% (mean 91%) for the longer reads, showing that data volume reduction is consistently high if per-base error rates are maintained independent from read length.

Theoretically, the average runtime of Passage is $O(n)$, which was found to hold true in practice (as described in [12]). To assess the runtime and memory requirements of the clustering implementation described here and to find out if other programs can perform the same task with similar effort, Passage was compared to programs designed for transcript assembly. Although such programs are not designed for the same task, they can be used for such a comparison as the problem at hand is a special case of transcript assembly (with only 0% or 100% overlap between fragments to assemble).

The test dataset consisted of 4,610,214 reads of length 40bp. To define a 'gold standard' of the expected number of clusters, the read sequences were trimmed to remove the `ACTTTTTTTTTTTTT`, `ACGGG` and `AC` prefixes and then mapped against the reference genome using Bowtie [95] and allowing for two mismatches. The resulting number of unique mapping position was used as the expected number of clusters to be generated. This is based on the assumption that read sequences derive from genomic fragments and that each cluster created by Passage should correspond to one genomic locus covered by a group of identical reads. For want of a 'true' gold standard this method was chosen as the best approximation, even though there are several caveats with this assumption:

- In the case of genomic duplications with identical sequence (large-scale duplications or smaller duplicated elements such as pseudogenes), the mapping process would create several mapping positions while PASSAGE would create only one cluster. If all mapping positions were counted, this might lead to an overestimation of the expected number of clusters. For the test dataset, only *unique* mappings were considered, i.e., reads mapping to more than one position with identical alignment score were discarded.
- Bowtie can not map reads to the genomic sequence if they span an exon-intron junction as this requires splitting the read into two parts and aligning them separately. Thus, such reads would not be mapped and would not be contributing to the number of mapping positions. PASSAGE, on the other hand, would have no problem clustering these reads. As a result, the mapping process could underestimate the number of clusters to be found. To counteract this problem, only reads mappable by Bowtie were included in the test dataset.

3,442,541 reads (74.67%) could be aligned to the genome to 40,023 unique positions. Based on this gold standard, a clustering of reads performed without the reference genome should approximately generate forty thousand clusters. If a program produces significantly fewer clusters, it is likely to merge clusters stemming from different genomic loci. As a result, the expression estimates obtained from that clustering will be too high and/or many fragments will be missed. In addition, such a clustering would result in clusters with either very vague consensus sequences (using ambiguous bases) or consensus sequences longer than the well-known read length (violating the 0%-or-100%-overlap rule). On the other hand, if a program produces significantly more clusters, it will underestimate the expression of the fragments because clusters arising due to sequencing errors would not have been merged correctly.

The tests clearly showed that the tested assembly programs use more memory and have a higher runtime than our implementation, and, most importantly, can not solve the PASSAGE clustering problem (see table 8.1), either producing too few or too many clusters. Interestingly, Mira3 not only produces too many clusters, but these clusters also have very ambiguous consensus sequences. Assembly programs with their (for this problem) very generic approach are clearly not useful for our purpose.

The development of new tools for second-generation sequencing data lead to the publication of new software tools specifically designed for clustering second-generation DNA-seq reads. One such tool is SEED [7] which aims at reducing the redundancy found in NGS datasets by finding so-called *center sequences* and by clustering each read with the center sequence if their distance is less than a predefined threshold. As such, SEED is very similar to PASSAGE, and the authors claim that clustering reads with their software also improves sequence assembly based on the clustered reads. To complement the comparisons described above, PASSAGE's performance was also compared with that of SEED and the results are presented in table 8.1 alongside those of the other programs. Though SEED produces almost the same number of clusters as PASSAGE, its runtime is more than five times as high.

Table 8.1.: Comparison of PASSAGE to other programs

Program	CPU time	memory	clusters/contigs	*mcl*
Bowtie[b] [95]	22 min	160 MB	[c]40,023	–
Assemblers				
Cap3[a] [72]	–	>16000 MB	–	–
Mira3 [37]	22 h	3000 MB	70,121	40.23
Locas [90]	35 min	4100 MB	2,537	40.10
Velvet [184]	1 min	600 MB	291	42.12
Read clustering tools				
SEED [7]	5 min	2500 MB	40,808	40
PASSAGE	<1 min	1200 MB	40,759	40

Runtimes and resulting number of clusters/contigs for all tested programs. *mcl*, mean consensus length. [a]Cap3 did not complete due to memory restrictions; [b]Bowtie was used to establish a gold standard for the expected number of clusters; [c]unique mapping sites. Table updated and extended based on [12].

The number of clusters reported by PASSAGE is closest to the estimate produced using Bowtie. A small discrepancy is to be expected. In fact, there are explanations for differences in either direction, resulting in too few or too many clusters created by read clustering programs such as PASSAGE. Both cases result from the fact that clustering is performed without knowledge of the genomic reference sequence:

- **Too many clusters** can occur if the genomic sequence of a locus is never sequenced perfectly. A lowly-expressed gene, for instance, could result in two reads, each having two mismatches (sequencing errors, SNPs) to the genomic reference sequence. If the errors in the two reads do not occupy the same positions, the distance between those reads would be four. The genomic sequence would lie 'between' the read sequences, in terms of the distances. A mapping tool such as Bowtie would be able to align both reads to the same genomic locus with two mismatches, creating a single mapping coordinate. During read clustering, however, these reads would result in the creation of two separate clusters, as they are too far apart for merging.
- **Too few clusters** can occur because PASSAGE assumes that two clusters with less than k mismatches come from the same genomic locus. In some cases, mapping these two clusters' sequences against the genomic reference might reveal that there exist, in fact, two loci with almost identical sequences. Here, the number of mapping positions would be larger than the number of clusters created.

Considering that the number of clusters found is only slightly larger than the expected number (by 1.8%), and that this 'gold standard' itself is also just an estimate of the 'true' situation, the PASSAGE clustering is likely to be close to perfect.

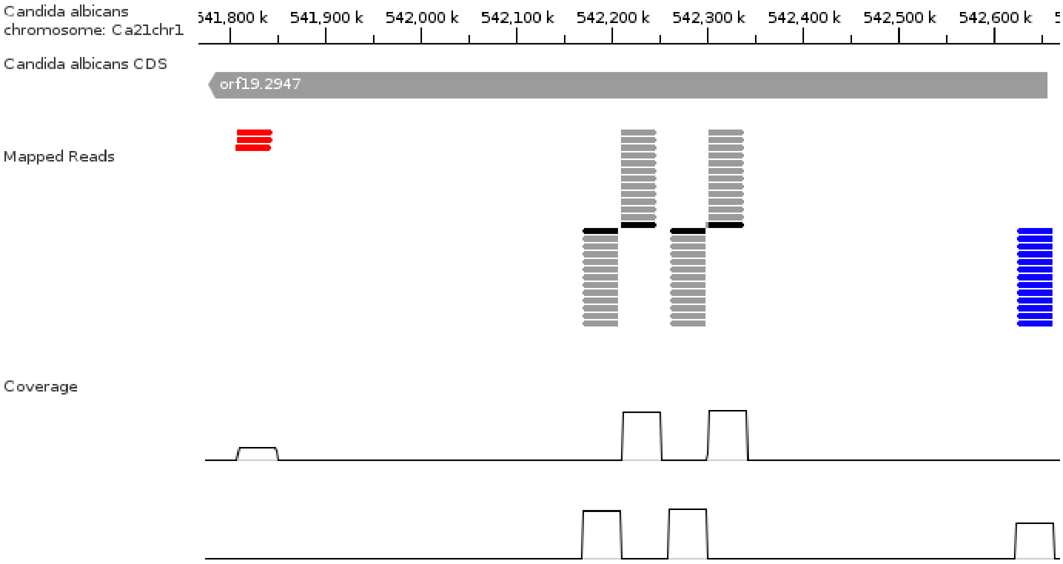


Figure 8.3.: PASSAGE fragments along a transcript, shown for orf19.2947 on *Candida albicans* chromosome 1 (antisense strand). Reads are colored according to their PASSAGE fragment class: Blue, 5' UTR; gray, internal; red, 3' UTR. The *RsaI* restriction sites result in internal fragments sequenced in both directions. Internal fragments create many more reads than the UTR fragments, causing many reads to be drawn overlapping each other due to space constraints. The log coverage curve at the bottom clearly shows these differences.

8.4. Recovering the transcripts

One interesting question regarding PASSAGE is whether it is possible to recover which fragments belong together, i.e., which fragments come from the same transcript. As fragments do not overlap and the reference genome is not available for mapping, this information can not be obtained by computing overlaps, comparing mapping positions to existing gene annotations or by grouping together fragments based on their distance in the genome.

A simple idea would be to group fragments by their expression estimate, assuming that the 5', internal and 3' fragments of one transcript should have very similar expression (allowing for small differences due to the sampling of reads). However, this procedure does not work because expression of the different fragments of one transcript can in fact be highly different [182, 25]. The same effect was found for PASSAGE data where the genome sequence was available (see figure 8.3 for an example). Thus, there is presently no method to reconstruct the relationship between fragments generated by the PASSAGE protocol without using a known reference genome or, at least a library of known transcript sequences.

9. Discussion

Many challenges are posed by biological research focusing more and more on large-scale, comprehensive studies in general, and by the introduction of new experimental techniques for high-throughput experiments in particular. For the field of bioinformatics, these challenges are all centered around the theme of *integration*. Bioinformaticians are called to develop methods and tools which simplify and assist in the task of deriving knowledge from the experimental data. This task starts with processing data from different sources, with specific biases to take into account, and stored in different data formats. It includes the combined application of methods from statistics and visualization in the context of meta information, and extends up to the merging of different research groups' findings into a global view of the biological system under investigation.

My contribution in the course of my PhD period was the creation of a flexible, extensible software tool for visual omics data analysis which offers solutions to some of these challenges. The new Mayday software

1. integrates many stand-alone methods into a common application,
2. integrates many visualizations, enhanced by the use of meta-information,
3. integrates interactive analysis and ad-hoc programming,
4. integrates with other Systems Biology applications,
5. integrates results from different physical locations via GaggleBridge, and
6. integrates new types of transcriptomics data into existing analysis pipelines.

In this dissertation, I have presented major changes to the existing Mayday framework, involving the complete re-design of several existing systems. The code base has grown considerably over the time of this dissertation, as new facilities were added to Mayday's core (e.g., data structures, the `Settings` framework). At the same time, functionality that was previously implemented several times in different sub-systems (written by different authors) was consolidated and moved into the appropriate core packages. Thus, the overall increase in the size of the project is a combination of a growth due to new, well-designed core classes/packages and new plugins, and a decrease resulting from the removal of (duplicated) ad-hoc solutions.

These developments followed two major aims: Firstly, the creation of a comprehensive, 'one stop' solution for transcriptome analysis tasks. This was clearly achieved. Mayday now integrates a large range of methods covering different analysis tasks (e.g., filtering, statistics, inclusion of meta data, and visualization), giving users a powerful software tool for their transcriptomics analyses. From the user's point of view, the user interface is now much richer, exposing more of the data model to direct inspection and interaction. The re-use of core components instead of per-plugin implementations results in a more consistent user experience. Mayday's power was proven in the course of the SysMO Stream project, during which many new extensions were conceived, implemented and put to use. Mayday's strengths were also recognized by external users as given by currently 35 original papers citing one of the Mayday publications (the core publications [46, 16], or the sub-system publications [156, 155, 14]). Recently, a review of microarray analysis software [93] found that Mayday was one of the top three tools in this field, together with MeV [133] and Chipster [87] (see also section 9.1).

The second aim was to establish a solid foundation for further software development. This aim was also achieved as evidenced by several new developments which are based on the new Mayday core. From a programmer's point of view, implementing new methods as well as complete sub-systems for Mayday has become easier, with less time spent on writing user interface components, task management, and data handling, to name only a few, and more time available for the implementation of the actual method. This aspect is discussed further in section 9.2. During method development, but also during complex or non-standard analyses, interactive programming can be highly useful to explore new ideas and hypotheses. Mayday's 'universal shell' package provides a user interface and a range of functionality required for the implementation of interactive scripting and querying on Mayday's data structures. On this basis, Mayday RLink provides an interface to the extremely powerful R language for statistical programming, a framework for method development in the JavaScript language which allows for very tight integration with Mayday's Java core, and an interface for data analysis using the database query language SQL. Other programming and query languages can now easily be added as plugins.

The integration of sequencing-based expression data and a comprehensive data preprocessing framework was the third aim and major focus of my work. Based on the strong foundation laid with the new Mayday core, SeaSight allows users to work on data from recently introduced transcript quantification methods based on new high-throughput sequencing technologies as well as from traditional, well-established microarray platforms. This extends Mayday's applicability to new data types, to the full transcriptomics analysis pipeline starting from raw data, and to new kinds of projects, as described in section 9.3.

The penultimate section 9.4 presents some ideas for further development which have become possible with the new Mayday core and the large additions to Mayday's functionality presented in this dissertation. It is followed by a final conclusion of my work in section 9.5.

database query language SQL. Other programming and query languages can be added as plugins.

9.1. Mayday as a powerful framework for expression analysis

Mayday was used for data analysis in the context of a large study of time-series data [111, 11, 171] during which its stability, the flexible visualizations, and the possibility to quickly add new features proved to be very useful.

Apart from work done in the *Integrative Transcriptomics* group in Tübingen, Mayday is also used by other researchers, and was recently analyzed as part of a large survey of microarray analysis software [93]. In that survey, Mayday was found to offer *"the second-most complete range of features"*, which is remarkable given the fact that most other free microarray analysis software packages only present a graphical user interface to the functions offered as part of the Bioconductor [58] packages for R [125], while the respective functions in Mayday have all been (re-)implemented in Java. As the authors further note, this makes Mayday's analyses *"run comparatively fast"*. The wealth of methods and the flexibility it provides result in a large number of configurable options that are missing from other software (such as the choice of distance measure for k-means clustering, for instance), which the authors recognize as a double-edged sword, stating that *"the functionally-rich interface, that also allows to set various parameters for every method, is clearly geared towards the expert and can be demanding for less experienced or occasional users."*

As a stand-alone application, Mayday has none of the setup and configuration costs that are common to client/server-based applications (and even some applications running on top of an R installation) which demand that such installations *"should only be attempted by an experienced engineer. These systems need to have a database installed (often in a specific version) which needs to be configured properly"*. While one would expect that this extra cost is amortized by the benefits of having a database-backed client/server solution, Koschmieder *et al.* find that in contrast to Mayday, most database-based systems do *not* work well with large experiments, or fail completely when trying to import the data.

Finally, as many published microarray analysis packages are no longer actively developed, and a large number is no longer available for download, the fact that Mayday is actively developed and support requests are quickly answered is specifically highlighted by the authors. Of 78 tools included in the initial survey list, the authors find that only seven are open-source, available for download, and could successfully be installed.

Thus, even though Mayday appears to be only one tool in a vast ocean of such programs, it is in fact (at least according to Koschmieder *et al.*) one of only six tools that are actually really *available* to the scientific community, one of four able to deal with a large human transcriptome dataset, and one of only three programs that can be used without a complicated installation procedure.

9.2. Mayday as a powerful platform for development

The powerful core structure implemented in Mayday since version 2.5 greatly reduces the amount of work necessary for programmers to add new functionality to the program, especially if they wish to extend an existing part. As a result, Mayday can be used as a platform for rapid prototyping of new ideas for visualizations and statistical measures, amongst others.

The current implementation of Mayday's heatmap (described in section 4.2) is a good example. Itself being a visualization *plugin*, the heatmap can delegate most of its work to Mayday's core to minimize implementation effort: Everything related to settings (including storage, user interface generation, and communication of changes due to user interaction) is handled by the `Settings` framework; the discovery, instantiation and management of column and header renderers is done by the plugin manager; and everything connected with the data being visualized is the responsibility of the Vis3 framework. If a row, column, or header renderer has user-configurable settings, these are integrated automatically into the relevant menus and dialogs. As the same `Setting` types are used by other MAYDAY plots, they are already familiar to users. A combination of Vis3 and core data structures, `Settings`, and specific rendering code is used to implement meta-information dependent rendering, which was one of the prime features of the original *enhanced* MAYDAY heatmap [57].

While the new heatmap is an excellent example of powerful, rich-interface tools that can be written based on plugins, settings and the visualization framework, it is by far not the only one. The largest and most complete illustration of MAYDAY's power as a development platform is given by Stephan Symons' MAYDAY GRAPH VIEWER [155]. A third example is the time-series analysis tool, TIALA, which was described in section 4.4.2. Further, more recent examples not described here are the GenomeRing visualization of multiple genome alignments [68], and Günter Jäger's eQTL analysis tool REVEAL [86]

Besides such extensions which are relatively self-contained, MAYDAY's architecture also allows programmers to implement changes that deeply affect MAYDAY's core operations without actually touching a single line of the core codebase. The dynamic filtering framework introduced in section 3.5 gives a good example of how far-reaching modifications to core facilities such as the `ProbeList` can be implemented as MAYDAY plugins without requiring any change of the core data structures.

9.3. Sequencing for transcriptome analyses

When the new high-throughput sequencing technologies were introduced, software tools needed to be created to analyze the new type of data generated by these technologies. Quality control and read mapping tools were among the first programs developed. Using mapped read data as input, variant detection and expression quantification methods could be implemented, followed by programs for the computation of differential expression. However, as shown in section 7.2, the early tools were all single-method implementations for either expression quantification, or statistical testing, or visualization. Some of them were later included in larger frameworks, such as the Galaxy web service, for instance.

MAYDAY SEASIGHT was one of the first, probably even *the* first, implementation of a complete analysis pipeline covering all steps from importing mapped read data from a variety of specialized read mapping tools, via expression quantification and normalization, filtering and statistical testing, up to highly interactive and flexible visualization of results. At each step, researchers can apply different methods to process data from RNA-seq experiments in a variety of ways. With this 'one stop shop' approach to data analysis, MAYDAY SEASIGHT marks a major achievement in transcriptome bioinformatics. It allows researchers to focus on interpreting their data instead of memorizing how to apply a series of programs specialized for just one single task and writing custom processing scripts for each project. Getting started on an analysis with Mayday SeaSight takes only a few minutes as no complicated installation procedure is required and interactive visualizations can be used to quickly inspect the data.

As the importance of RNA-seq for transcriptomics research is increasing, traditional, microarray-based transcriptome studies will remain an essential method to research the roles of individual genes as well as groups of genes defined by, e.g., metabolic pathways, functional families or common intra-cellular localization of the gene products. For well-characterized organisms where the transcriptomic potential is already known, microarrays are still considerably cheaper than full-scale sequencing experiments, and their analysis is less fraught with problems of handling large volumes of data. Thus, Mayday's strong capabilities in this field will remain highly important for future research. The flexibility provided by SeaSight allows for the relatively quick incorporation of new or improved normalization methods to extract as much useful information from such experiments as possible. This has the benefit of enabling users to analyze their transcriptome data in a single software package, irrespective of the technological platform used for data acquisition, reducing the need for adjusting to a completely different program for each data type.

In addition, Mayday (with SeaSight) can now be used for combined analysis of both RNA-seq and microarray data. In the future, studies will make even more use of combined approaches. For example, in the field of genome-wide association studies (GWAS [91]), sequencing experiments of a subset of a large population can be used to determine the set of variants which are then verified and analyzed in large cohort studies with specifically designed (and cheaper) microarrays. One such approach is proposed by Illumina with their so-called 'Omni' line of products [76]. Similar developments are likely to occur in the field of gene expression analysis.

High-throughput sequencing is becoming an established tool for transcriptome studies (as well as for a range of other areas such as genome sequencing, variant detection, epigenome studies [176, 74], and taxonomic classification of metagenome samples [73, 59], for example). A large part of the new technology's utility lies in the fact that RNA-seq can be used to create an unbiased view of a cell's transcriptome, independent of a previously chosen set of known transcripts. In conjunction with a known genome sequence, this allows for the detection of novel transcripts [157] and isoforms [127, 151], as well as for correcting existing gene annotations, for instance including exons only found in rare splice variants.

However, the fact that RNA-seq currently requires the existence of a reference sequence for read mapping can prove to be a significant obstacle for researchers working with samples from organisms which are unknown or hard to cultivate, as the obtained reads can not be mapped to a genome. Two possible directions can be taken to use traditional RNA-seq data in the absence of a reference sequence. Transcript assembly tools can be employed to try to recover full-length transcripts from the sequencing reads. Alternatively, the reads can be mapped directly to a database of all known sequences (e.g., using Blast [3]) to try to gain an insight into the organism community and gene content of a (meta-) transcriptome sample.

A third alternative described in this work is the application of the PASSAGE protocol for sequencing followed by the read clustering algorithm presented in chapter 8. With this method, comparative transcriptomics experiments are possible even for samples with unknown reference genomes. Data volume is greatly reduced due to the restriction of the sequenced reads to precise locations (starting at *RsaI* restriction sites). As a result, smaller sample volumes and shorter read lengths can be used, leading to a significant reduction in cost as compared to full-scale RNA sequencing experiments.

The PASSAGE algorithm for read clustering makes use of the special properties of the reads produced by the wet-lab protocol to efficiently group them into clusters transcribed from the same genomic locus and quantify the expression for each transcript fragment. Compared to other tools addressing the generic problem of transcript assembly and more recent algorithms developed particularly for clustering of short reads from high-throughput sequencing, the implementation presented here is superior both in memory demands as well as regarding computing time. Most importantly, results indicate that it produces results most similar to the 'gold standard' defined by counting unique mapping positions for the sequenced reads.

9.4. Outlook

The MAYDAY project has come a long way, thanks in a large part to the work of students implementing new methods, collaborators and other researchers evaluating existing and requesting new functionality, and, most importantly, the other core developers, Nils Gehlenborg, Janko Dietzsch, Stephan Symons, and Günter Jäger, under the supervision of Kay Nieselt.

However, there always remain improvements which were not implemented for lack of time, or have just become possible with the latest developments in MAYDAY, regarding both new analysis technologies (e.g., RNA-seq), as well as changes in the scope of research (such as the wide approach taken by Systems Biology projects).

For SEASIGHT, one very interesting area would be the inclusion of more sophisticated quantification methods, such as the model used by rQuant [25], or the statistical inference method proposed by Jiang and Wong [84], and their evaluation against less complicated methods in terms of sensitivity and specificity of detection of differentially expressed transcripts. Furthermore, while read mapping qualities are already part of SEASIGHT's data model, they are not used during transcript quantification so far. It might be worth studying whether integrating this additional information into the quantification step results in a notable improvement, or whether such quality information should rather be used at an earlier step in the RNA-seq pipeline, for instance to filter alignments during read mapping.

As microarrays will likely remain a standard tool for transcriptome research, support for further array formats could be added, for example for the bead-based microarrays [113] manufactured by Illumina.

Regarding PASSAGE, research is currently under way to extend the wet-lab protocol so that it can be applied to prokaryotic genomes and to the study of non-coding

RNAs. The Passage read clustering algorithm could also benefit from further development, for example by parallelizing the algorithm where possible, by reducing memory consumption, and by evaluating the use of sophisticated, lock-free data structures for efficient parallel computations, based on atomic compare-and-swap operations.

As transcriptome projects grow and the community begins to settle on a few generally accepted algorithms for data processing (as has happened in the microarray world, for instance, where Affymetrix arrays are usually normalized with the RMA method [77]), automating analyses will become more and more interesting. Mayday already offers some limited support for automation [13], but the new plugin system as well as the `Settings` framework open the door for automated processing on a much larger scale. One solution, already sketched (but not implemented), could be the addition of a simple interface to Mayday's plugin system. Called, say, `ScriptablePlugin`, it would add only two methods (without interfering with the existing plugin interfaces): The first method would return the expected size of the output of running the plugin (number of `ProbeLists`, number of meta information groups, number of `DataSets`, etc., depending on the plugin type). The second method would be used to determine the mode of operation for a subsequent call to the plugin's `run` method, with three modes available: Init (to initialize `Settings` based on parameters supplied to the `run` method, Auto (to start non-interactive processing when `run` is called), and Interactive (to start normal plugin execution with user interaction). Using these two methods together with the existing `run` and `getSettings` method, complex processing pipelines could be built using any of Mayday's plugins, as well as some additional methods for iteration, branching and combination of results.

A fourth important field for future development are solutions for collaborative analyses. The integration of Mayday with the Gaggle is a step in that direction. Future improvements could be the use of specialized objects for communication between Mayday instances, perhaps based on the Gaggle 'tuple' data type. Alternatively, collaborative analyses could be enabled independent of the Gaggle system, either based on a separate RMI-based protocol, or using a web-based client-server solution. With the growth of Systems Biology projects both in terms of data volume per 'omics' type, as well as in terms of the number of different 'omics' types, and the number of people and research groups involved, collaboration between researchers working on different aspects of one system will become increasingly important. Software developers should support this process by minimizing the time spent on data exchange and integration tasks, which are at the same time trivial from the point of view of the research question, and complicated regarding the implementation of good solutions.

A fifth area of high interest are analyses of associations between genotypic variations (single nucleotide polymorphisms, SNPs) and phenotypic outcomes such as disease susceptibility, speed of progression, or drug resistance. Traditionally, so-called *genome-wide association studies* (GWAS [91]) were performed to find statistically significant links between genotypes and phenotypes. *Expression quantitative trait locus* (eQTL [138]) studies extend this concept to include expression data, linking genotypic differences with differential expression and phenotypic outcomes. Based on Mayday, Günter Jäger has now started to develop the eQTL analysis tool Reveal [86] which incorporates methods presented at the IEEE Visweek 2011 as part of the BioVis 2011 challenge, where the approach was awarded the 'Visualization Experts Favorite' award. It includes ideas from the previously published iHAT [166] which offered hierarchical aggregation methods to analyze GWAS data. Reveal opens up a completely new direction of development for the Mayday project, where the data structures for genomic locations will likely be highly useful, among others.

9.5. Conclusion

Over the course of five years, Mayday evolved from a small program, which was mostly used as a basis for teaching (both for programming assignments as well as for development related to theses), into a powerful application for transcriptome (and other 'ome') data analysis with a strong focus on visual exploration, which is of great use for Systems Biology studies and recognized as one of the leading academic programs in this field [93].

As Systems Biology projects try to elucidate how the different biochemical processes in living cells are integrated, i.e., how the interplay of genomic information, protein-coding as well as functionally active transcripts, proteins and their modifications, and metabolites results in a living cell or organism interacting with its environment, the integration of different data types into one comprehensive image remains one of the most important areas of research in bioinformatics. In this context, Mayday offers a well-integrated, powerful foundation for further development to keep up with technological advances, and to include new analysis methods and innovative visualizations.

Bibliography

[1] C. Adessi, G. Matton, G. Ayala, G. Turcatti, J. Mermod, P. Mayer, and E. Kawashima. Solid phase DNA amplification: characterisation of primer attachment and amplification mechanisms. *Nucleic Acids Research*, 28(20):e87, 2000.

[2] Affymetrix. Data Sheet: GeneChip Human Genome Arrays, 2003. [`http://media.affymetrix.com/support/technical/datasheets/human_datasheet.pdf`, accessed 04-November-2011].

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[4] S. Anders and W. Huber. Differential expression analysis for sequence count data. *Genome Biology*, 11(10):R106, 2010.

[5] M. Ashburner, C. Ball, J. Blake, D. Botstein, H. Butler, J. Cherry, A. Davis, K. Dolinski, S. Dwight, J. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock. Gene Ontology: tool for the unification of biology. *Nature Genetics*, 25(1):25, 2000.

[6] A. Bairoch, R. Apweiler, C. Wu, W. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, R. Lopez, M. Magrane, M. J. Martin, D. A. Natale, C. O'Donovan, N. Redaschi, and L.-S. L. Yeh. The universal protein resource (UniProt). *Nucleic Acids Research*, 33(suppl 1):D154, 2005.

[7] E. Bao, T. Jiang, I. Kaloshian, and T. Girke. SEED: efficient clustering of next-generation sequences. *Bioinformatics*, 27(18):2502–2509, 2011.

[8] J. Bare, T. Koide, D. Reiss, D. Tenenbaum, and N. Baliga. Integration and visualization of systems biology data in context of the genome. *BMC Bioinformatics*, 11(1):382, 2010.

[9] J. Bare, P. Shannon, A. Schmid, and N. Baliga. The Firegoose: two-way integration of diverse data from different bioinformatics web resources with desktop applications. *BMC Bioinformatics*, 8(1):456, 2007.

[10] T. Barrett, D. Troup, S. Wilhite, P. Ledoux, D. Rudnev, C. Evangelista, I. Kim, A. Soboleva, M. Tomashevsky, K. Marshall, K. Phillippy, P. M. Sherman, R. N. Muertter, and R. Edgar. NCBI GEO: archive for high-throughput functional genomic data. *Nucleic Acids Research*, 37(suppl 1):D885–D890, 2009.

[11] F. Battke, A. Herbig, A. Wentzel, O. M. Jakobsen, M. Bonin, D. A. Hodgson, W. Wohlleben, T. E. Ellingsen, and K. Nieselt. A Technical Platform for Generating Reproducible Expression Data from Streptomyces coelicolor Batch Cultivations. In H. R. R. Arabnia and Q.-N. Tran, editors, *Software Tools and*

Algorithms for Biological Systems, volume 696 of *Advances in Experimental Medicine and Biology*, pages 3–15. Springer New York, 2011.

[12] F. Battke, S. Körner, S. Hüttner, and K. Nieselt. Efficient sequence clustering for RNA-seq data without a reference genome. *Lecture Notes in Informatics (LNI) – Proceedings*, P-157:21, 2010.

[13] F. Battke. A Processing Framework for Mayday. Studienarbeit, Universität Tübingen, 2006.

[14] F. Battke and K. Nieselt. Mayday SeaSight: Combined Analysis of Deep Sequencing and Microarray Data. *PLoS one*, 6(1):e16345, 01 2011.

[15] F. Battke, S. Symons, A. Herbig, and K. Nieselt. GaggleBridge: Collaborative data analysis. *Bioinformatics*, 2011.

[16] F. Battke, S. Symons, and K. Nieselt. Mayday – Integrative analytics for expression data. *BMC Bioinformatics*, 11(1):121, 2010.

[17] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 289–300, 1995.

[18] Y. Benjamini and D. Yekutieli. The control of the false discovery rate in multiple testing under dependency. *Annals of Statistics*, pages 1165–1188, 2001.

[19] R. Benne, J. Van Den Burg, J. Brakenhoff, P. Sloof, J. Van Boom, and M. Tromp. Major transcript of the frameshifted coxII gene from trypanosome mitochondria contains four nucleotides that are not encoded in the DNA. *Cell*, 46(6):819–826, 1986.

[20] S. Bennet. Solexa ltd. *Pharmacogenomics*, 5(4):433–438, 2004.

[21] S. Berget, C. Moore, and P. Sharp. Spliced segments at the 5'terminus of adenovirus 2 late mRNA. *PNAS*, 74(8):3171, 1977.

[22] M. Berthold, N. Cebron, F. Dill, T. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel. KNIME: The Konstanz information miner. *Data Analysis, Machine Learning and Applications*, pages 319–326, 2008.

[23] A. Blanchard, R. Kaiser, and L. Hood. High-density oligonucleotide arrays. *Biosensors and Bioelectronics*, 11(6-7):687–690, 1996.

[24] D. Blankenberg, A. Gordon, G. Von Kuster, N. Coraor, J. Taylor, A. Nekrutenko, and the Galaxy Team. Manipulation of FASTQ data with Galaxy. *Bioinformatics*, 26(14):1783–1785, 2010.

[25] R. Bohnert, J. Behr, and G. Rätsch. Transcript quantification with RNA-Seq data. *BMC Bioinformatics*, 10(suppl 13):P5, 2009.

[26] S. Boireau, P. Maiuri, E. Basyuk, M. De La Mata, A. Knezevich, B. Pradet-Balade, V. Bäcker, A. Kornblihtt, A. Marcello, and E. Bertrand. The transcriptional cycle of HIV-1 in real-time and live cells. *The Journal of Cell Biology*, 179(2):291, 2007.

[27] B. Bolstad, R. Irizarry, M. Åstrand, and T. Speed. A comparison of normalization methods for high density oligonucleotide array data based on variance

and bias. *Bioinformatics*, 19(2):185, 2003.

[28] C. Bonferroni. Il calcolo delle assicurazioni su gruppi di teste. *Studi in Onore del Professore Salvatore Ortu Carboni*, 13, 1935.

[29] C. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilitá*. Libreria internazionale Seeber, 1936.

[30] R. Breitling, P. Armengaud, A. Amtmann, and P. Herzyk. Rank products: a simple, yet powerful, new method to detect differentially regulated genes in replicated microarray experiments. *FEBS Letters*, 573:83–92, 2004.

[31] C. Broelemann. Kooperative Analyse von Hochdurchsatzdaten aus heterogenen Quellen durch Einbindung von Gaggle in Mayday. Diplomarbeit, Universität Tübingen, 2011.

[32] P. Bruns. Entwicklung eines effizienten Frameworks zur interaktiven Visualisierung in Mayday. Bachelor thesis, Universität Tübingen, 2008.

[33] J. Bullard, E. Purdom, K. Hansen, and S. Dudoit. Evaluation of statistical methods for normalization and differential expression in mRNA-Seq experiments. *BMC Bioinformatics*, 11(1):94, 2010.

[34] T. Cech. The generality of self-splicing RNA: relationship to nuclear mRNA splicing. *Cell*, 44(2):207–210, 1986.

[35] T. Cech. RNA editing: world's smallest introns? *Cell*, 64(4):667–669, 1991.

[36] C. Chang and R. Stewart. The two-component system: Regulation of Diverse Signalling Pathways in Prokaryotes and Eukaryotes. *Plant Physiology*, 117(3):723, 1998.

[37] B. Chevreux, T. Wetter, and S. Suhai. Genome Sequence Assembly Using Trace Signals and Additional Sequence Information. *Computer Science and Biology: Proceedings of the German Conference on Bioinformatics (GCB)*, 99:45–56, 1999.

[38] J. Clarke, H. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nature Nanotechnology*, 4(4):265–270, 2009.

[39] P. Cohen. The regulation of protein function by multisite phosphorylation – a 25 year update. *Trends in Biochemical Sciences*, 25(12):596–601, 2000.

[40] A. Cornish-Bowden. Nomenclature for incompletely specified bases in nucleic acid sequences. *Nucleic Acids Research*, 13(9):3021, 1985.

[41] R. Côté, P. Jones, L. Martens, S. Kerrien, F. Reisinger, Q. Lin, R. Leinonen, R. Apweiler, and H. Hermjakob. The Protein Identifier Cross-Referencing (PICR) service: reconciling protein identifiers across multiple source databases. *BMC Bioinformatics*, 8(1):401, 2007.

[42] A. Cozzone. Protein phosphorylation in prokaryotes. *Annual Reviews in Microbiology*, 42(1):97–125, 1988.

[43] J. Darnell. Implications of RNA-RNA splicing in evolution of eukaryotic cells. *Science*, 202(4374):1257, 1978.

[44] J. Darnell, W. Jelinek, and G. Molloy. Biogenesis of mRNA: genetic regulation in mammalian cells. *Science*, 181(106):1215, 1973.

[45] F. Denoeud, J. M. Aury, C. Da Silva, B. Noel, O. Rogier, M. Delledonne, M. Morgante, G. Valle, P. Wincker, C. Scarpelli, O. Jaillon, and F. Artiguenave. Annotating genomes with massive-scale RNA sequencing. *Genome Biology*, 9(12):R175, 2008.

[46] J. Dietzsch, N. Gehlenborg, and K. Nieselt. Mayday – a microarray data analysis workbench. *Bioinformatics*, 22(8):1010–1012, Apr 2006.

[47] M. Droege and B. Hill. The Genome Sequencer FLX System–longer reads, more applications, straight forward bioinformatics and more complete data sets. *Journal of Biotechnology*, 136(1-2):3–10, Aug 2008. [PubMed:18616967] [doi:10.1016/j.jbiotec.2008.03.021].

[48] S. Dudoit, J. Shaffer, and J. Boldrick. Multiple hypothesis testing in microarray experiments. *Statistical Science*, pages 71–103, 2003.

[49] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, G. Otto, P. Peluso, D. Rank, P. Baybayan, B. Bettman, A. Bibillo, K. Bjornson, B. Chaudhuri, F. Christians, R. Cicero, S. Clark, R. Dalal, A. Dewinter, J. Dixon, M. Foquet, A. Gaertner, P. Hardenbol, C. Heiner, K. Hester, D. Holden, G. Kearns, X. Kong, R. Kuse, Y. Lacroix, S. Lin, P. Lundquist, C. Ma, P. Marks, M. Maxham, D. Murphy, I. Park, T. Pham, M. Phillips, J. Roy, R. Sebra, G. Shen, J. Sorenson, A. Tomaney, K. Travers, M. Trulson, J. Vieceli, J. Wegener, D. Wu, A. Yang, D. Zaccarin, P. Zhao, F. Zhong, J. Korlach, and S. Turner. Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133, 2009.

[50] D. Fink, N. Weissschuh, J. Reuther, W. Wohlleben, and A. Engels. Two transcriptional regulators GlnR and GlnRII are involved in regulation of nitrogen metabolism in Streptomyces coelicolor A3 (2). *Molecular Microbiology*, 46(2):331–347, 2002.

[51] R. A. Fisher. The correlation between relatives on the supposition of Mendelian inheritance. *Transactions of the Royal Society of Edinburgh*, 52(2):399–433, 1918.

[52] R. A. Fisher. On the "Probable Error" of a Coefficient of Correlation Deduced from a Small Sample. *Metron*, 1:3–32, 1921.

[53] S. Fodor, J. Read, M. Pirrung, L. Stryer, A. Lu, and D. Solas. Light-directed, spatially addressable parallel chemical synthesis. *Science*, 251(4995):767, 1991.

[54] X. Fu, N. Fu, S. Guo, Z. Yan, Y. Xu, H. Hu, C. Menzel, W. Chen, Y. Li, R. Zeng, and P. Khaitovich. Estimating accuracy of RNA-Seq and microarrays with proteomics. *BMC Genomics*, 10(1):161, 2009.

[55] F. Galton. Co-relations and their measurement, chiefly from anthropometric data. *Proceedings of the Royal Society of London*, 45(273-279):135, 1888.

[56] N. Gehlenborg. Mayday – Microarray Data Analysis. Studienarbeit, Universität Tübingen, 2003.

[57] N. Gehlenborg, J. Dietzsch, and K. Nieselt. A Framework for Visualization of Microarray Data and Integrated Meta Information. *Information Visualization*, 4(3):164–175, June 2005.

[58] R. C. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber,

S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. Yang, and J. Zhang. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology*, 5(10):R80+, 2004.

[59] W. Gerlach and J. Stoye. Taxonomic classification of metagenomic shotgun sequences with CARMA3. *Nucleic Acids Research*, 39(14):e91–e91, 2011.

[60] B. Giardine, C. Riemer, R. Hardison, R. Burhans, L. Elnitski, P. Shah, Y. Zhang, D. Blankenberg, I. Albert, J. Taylor, W. Miller, J. W. Kent, and A. Nekrutenko. Galaxy: a platform for interactive large-scale genome analysis. *Genome Research*, 15(10):1451–1455, 2005.

[61] G. Gilliland, S. Perrin, K. Blanchard, and H. Bunn. Analysis of cytokine mRNA and DNA: detection and quantitation by competitive polymerase chain reaction. *PNAS*, 87(7):2725, 1990.

[62] F. Giorgi, A. Bolger, M. Lohse, and B. Usadel. Algorithm-driven Artifacts in median polish summarization of Microarray data. *BMC Bioinformatics*, 11(1):553, 2010.

[63] A. Goncalves, A. Tikhonov, A. Brazma, and M. Kapushesky. A pipeline for RNA-seq data processing and quality assessment. *Bioinformatics*, 27(6):867, 2011.

[64] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java (TM) Language Specification*, 2005.

[65] J. Hacker, L. Bender, M. Ott, J. Wingender, B. Lund, R. Marre, and W. Goebel. Deletions of chromosomal regions coding for fimbriae and hemolysins occur in vitro and in vivo in various extra intestinal *Escherichia coli* isolates. *Microbial Pathogenesis*, 8(3):213–225, 1990.

[66] T. Hardcastle and K. Kelly. baySeq: Empirical Bayesian methods for identifying differential expression in sequence count data. *BMC Bioinformatics*, 11(1):422, 2010.

[67] C. Heid, J. Stevens, K. Livak, and P. Williams. Real time quantitative PCR. *Genome Research*, 6(10):986, 1996.

[68] A. Herbig, G. Jäger, F. Battke, and K. Nieselt. GenomeRing: alignment visualization based on SuperGenome coordinates. *Bioinformatics*, 28(12):i7–i15, 2012.

[69] L. Heyer, S. Kruglyak, and S. Yooseph. Exploring expression data: identification and analysis of coexpressed genes. *Genome Research*, 9(11):1106, 1999.

[70] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, pages 65–70, 1979.

[71] F. Hong, B. Wittner, and contributions from R. Breitling, C. Smith, and F. Battke. *RankProd: Rank Product method for identifying differentially expressed genes with application in meta-analysis*, 2011. R package version 2.28.0.

[72] X. Huang and A. Madan. CAP3: A DNA sequence assembly program. *Genome Research*, 9(9):868–877, Sep 1999.

[73] D. Huson, A. Auch, J. Qi, and S. Schuster. MEGAN analysis of metagenomic data. *Genome Research*, 17(3):377–386, 2007.

[74] M. Huss. Introduction into the analysis of high-throughput-sequencing based epigenome data. *Briefings in Bioinformatics*, 11(5):512, 2010.

[75] S. Hüttner. *Genexpressionsanalyse.* German Patent DE 102009058298, Dt. Patentamt München, 2011.

[76] Illumina. Next-Generation GWAS, 2011. [`http://www.illumina.com/landing/gwas_ebook/video/files/gwas_ebook_secured.pdf`, accessed 15-December-2011].

[77] R. Irizarry, B. Hobbs, F. Collin, Y. Beazer-Barclay, K. Antonellis, U. Scherf, and T. Speed. Exploration, normalization, and summaries of high density oligonucleotide array probe level data. *Biostatistics*, 4(2):249, 2003.

[78] G. Jäger. QT-Clustering for Mayday. Bachelor thesis, Universität Tübingen, 2008.

[79] G. Jäger, F. Battke, and K. Nieselt. TIALA – Time Series Alignment Analysis. In *Biological Data Visualization (BioVis), 2011 IEEE Symposium on*, pages 55–61, Oct. 2011.

[80] B. Jagla, B. Wiswedel, and J. Coppée. Extending KNIME for Next Generation Sequencing data analysis. *Bioinformatics*, 2011.

[81] G. Janaway, C. Inman, and J. Beechem. Single molecule nucleic acid sequencing using multiphoton fluorescence excitation. *US Patent 13/009,442*, 2011.

[82] A. Jasper. A Mayday plugin for Affymetrix Chip Analysis. Studienarbeit, Universität Tübingen, 2008.

[83] Java Community Process. JSR 223: Scripting for the Java Platform, 2006. [`http://www.jcp.org/en/jsr/detail?id=223`; accessed 26-Sept-2011].

[84] H. Jiang and W. Wong. Statistical inferences for isoform expression in RNA-Seq. *Bioinformatics*, 25(8):1026–1032, 2009.

[85] W. Johansen. *Elemente der exakten Erblichkeitslehre.* Gustav Fischer, Jena, 1909.

[86] G. Jäger, F. Battke, and K. Nieselt. Reveal – Visual eQTL Analytics. *Bioinformatics*, 28(18):i542–i548, 2012.

[87] M. Kallio, J. Tuimala, T. Hupponen, P. Klemela, M. Gentile, I. Scheinin, M. Koski, J. Kaki, and E. Korpelainen. Chipster: user-friendly analysis software for microarray and other high-throughput data. *BMC Genomics*, 12(1):507, 2011.

[88] M. Kanehisa and S. Goto. KEGG: Kyoto encyclopedia of genes and genomes. *Nucleic Acids Research*, 28(1):27, 2000.

[89] P. Kapranov, J. Cheng, S. Dike, D. A. Nix, R. Duttagupta, A. T. Willingham, P. F. Stadler, J. Hertel, J. Hackermüller, I. L. Hofacker, I. Bell, E. Cheung, J. Drenkow, E. Dumais, S. Patel, G. Helt, M. Ganesh, S. Ghosh, A. Piccolboni, V. Sementchenko, H. Tammana, and T. R. Gingeras. RNA maps reveal new RNA classes and a possible function for pervasive transcription. *Science*, 316(5830):1484, 2007.

[90] J. Klein, S. Ossowski, K. Schneeberger, D. Weigel, and D. Huson. LOCAS – A low coverage assembly tool for resequencing projects. *PloS one*, 6(8):e23455,

2011.

[91] R. J. Klein, C. Zeiss, E. Y. Chew, J. Y. Tsai, R. S. Sackler, C. Haynes, A. K. Henning, J. P. SanGiovanni, S. M. Mane, S. T. Mayne, M. B. Bracken, F. L. Ferris, J. Ott, C. Barnstable, and J. Hoh. Complement factor H polymorphism in age-related macular degeneration. *Science*, 308(5720):385, 2005.

[92] R. Kornberg. Eukaryotic transcriptional control. *Trends in Biochemical Sciences*, 24(12):M46–M49, 1999.

[93] A. Koschmieder, K. Zimmermann, S. Trißl, T. Stoltmann, and U. Leser. Tools for managing and analyzing microarray data. *Briefings in Bioinformatics*, 2011.

[94] J. Lajugie and E. Bouhassira. GenPlay, a multi-purpose genome analyzer and browser. *Bioinformatics*, 2011.

[95] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[96] A. Leśniewska and M. Okoniewski. rnaSeqMap: a Bioconductor package for RNA sequencing data exploration. *BMC Bioinformatics*, 12(1):200, 2011.

[97] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754, 2009.

[98] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078, 2009.

[99] R. Ling. A computer generated aid for cluster analysis. *Communications of the ACM*, 16(6):355–361, 1973.

[100] K. Liolios, I. Chen, A. Min, K. Mavromatis, N. Tavernarakis, P. Hugenholtz, V. Markowitz, and N. Kyrpides. The Genomes On Line Database (GOLD) in 2009: status of genomic and metagenomic projects and their associated metadata. *Nucleic Acids Research*, 38(suppl 1):D346, 2010.

[101] T. Loua. *Atlas statistique de la population de Paris*. J. Dejey & cie, 1873.

[102] E. Lundberg, L. Fagerberg, D. Klevebring, I. Matic, T. Geiger, J. Cox, C. Älgenäs, J. Lundeberg, M. Mann, and M. Uhlen. Defining the transcriptome and proteome in three functionally different human cell lines. *Molecular Systems Biology*, 6:450, 2010.

[103] T. Maier, M. Guell, and L. Serrano. Correlation of mRNA and protein in complex biological samples. *FEBS Letters*, 583(24):3966–3973, 2009.

[104] L. Mamanova, R. Andrews, K. James, E. Sheridan, P. Ellis, C. Langford, T. Ost, J. Collins, and D. Turner. FRT-seq: amplification-free, strand-specific transcriptome sequencing. *Nature Methods*, 7(2):130–132, 2010.

[105] J. Marioni, C. Mason, S. Mane, M. Stephens, and Y. Gilad. RNA-seq: an assessment of technical reproducibility and comparison with gene expression arrays. *Genome Research*, 18(9):1509, 2008.

[106] A. Maxam and W. Gilbert. A new method for sequencing DNA. *PNAS*, 74(2):560, 1977.

[107] U. Meier and G. Blobel. A nuclear localization signal binding protein in the nucleolus. *The Journal of Cell Biology*, 111(6):2235, 1990.

[108] K. Michael, L. Taylor, S. Schultz, and D. Walt. Randomly ordered addressable high-density optical sensor arrays. *Analytical Chemistry*, 70(7):1242–1248, 1998.

[109] A. Mortazavi, B. Williams, K. McCue, L. Schaeffer, and B. Wold. Mapping and quantifying mammalian transcriptomes by RNA-Seq. *Nature Methods*, 5(7):621–628, 2008.

[110] K. Mullis, F. Faloona, S. Scharf, R. Saiki, G. Horn, and H. Erlich. Specific enzymatic amplification of DNA in vitro: the polymerase chain reaction. In *Cold Spring Harb Symp Quant Biol*, volume 51, pages 263–273, 1986.

[111] K. Nieselt, F. Battke, A. Herbig, P. Bruheim, A. Wentzel, O. M. Jakobsen, H. Sletta, M. T. Alam, M. E. Merlo, J. Moore, W. A. Omara, E. R. Morrisey, M. A. Juarez-Hermosillo, A. Rodriguez-Garcia, M. Nentwich, L. Thomas, M. Iqbal, R. Legaie, W. H. Gaze, G. L. Challis, R. C. Jansen, L. Dijkhuizen, D. A. Rand, D. L. Wild, M. Bonin, J. Reuther, W. Wohlleben, M. C. Smith, N. J. Burroughs, J. F. Martin, D. A. Hodgson, E. Takano, R. Breitling, T. E. Ellingsen, and E. M. Wellington. The dynamic architecture of the metabolic switch in Streptomyces coelicolor. *BMC Genomics*, 11(10):10–10, Jan 2010.

[112] M. Nirenberg, P. Leder, M. Bernfield, R. Brimacombe, J. Trupin, F. Rottman, and C. O'neal. RNA codewords and protein synthesis, VII. On the general nature of the RNA code. *PNAS*, 53(5):1161, 1965.

[113] A. Oliphant, D. L. Barker, J. R. Stuelpnagel, and M. S. Chee. BeadArray technology: enabling an accurate, cost-effective approach to high-throughput genotyping. *Biotechniques*, 32(6):56–58, 2002.

[114] A. Oshlack and M. Wakefield. Transcript length bias in RNA-seq data confounds systems biology. *Biology Direct*, 4(1):14, 2009.

[115] H. Parkinson, U. Sarkans, N. Kolesnikov, N. Abeygunawardena, T. Burdett, M. Dylag, I. Emam, A. Farne, E. Hastings, E. Holloway, N. Kurbatova, M. Lukk, J. Malone, R. Mani, E. Pilicheva, G. Rustici, A. Sharma, E. Williams, T. Adamusiak, M. Brandizi, N. Sklyar, and A. Brazma. ArrayExpress update – an archive of microarray and high-throughput sequencing-based functional genomics experiments. *Nucleic Acids Research*, 39(suppl 1):D1002, 2011.

[116] K. Paszkiewicz and D. Studholme. De novo assembly of short sequence reads. *Briefings in Bioinformatics*, 11(5):457, 2010.

[117] K. Pearson. On the general theory of skew correlation and non-linear regression. *Biometrika*, 4:172–212, 1905.

[118] A. Pease, D. Solas, E. Sullivan, M. Cronin, C. Holmes, and S. Fodor. Light-generated oligonucleotide arrays for rapid DNA sequence analysis. *PNAS*, 91(11):5022, 1994.

[119] S. Pepke, B. Wold, and A. Mortazavi. Computation for ChIP-seq and RNA-seq studies. *Nature Methods*, 6:S22–S32, 2009.

[120] S. Polonsky, S. Rossnagel, and G. Stolovitzky. Nanopore in metal-dielectric sandwich for DNA position control. *Applied Physics Letters*, 91:153103, 2007.

[121] M. Polz and C. Cavanaugh. Bias in template-to-product ratios in multitemplate PCR. *Applied and Environmental Microbiology*, 64(10):3724, 1998.

[122] G. J. Porreca, J. Shendure, and G. M. Church. Polony DNA sequencing. *Current Protocols in Molecular Biology*, Chapter 7, Nov 2006. [PubMed:18265387] [doi:10.1002/0471142727.mb0708s76].

[123] K. Pruitt, T. Tatusova, W. Klimke, and D. Maglott. NCBI Reference Sequences: current status, policy and new initiatives. *Nucleic Acids Research*, 37(suppl 1):D32, 2009.

[124] J. Quackenbush. Microarray data normalization and transformation. *Nature Genetics*, 32:497, 2002.

[125] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2006. ISBN 3-900051-07-0.

[126] V. Ramakrishnan. Ribosome structure and the mechanism of translation. *Cell*, 108(4):557–572, 2002.

[127] H. Richard, M. Schulz, M. Sultan, A. Nürnberger, S. Schrinner, D. Balzereit, E. Dagand, A. Rasche, H. Lehrach, M. Vingron, S. A. Haas, and M. L. Yaspo. Prediction of alternative isoforms from exon expression levels in RNA-Seq experiments. *Nucleic Acids Research*, 38(10):e112–e112, 2010.

[128] T. Ries. Automatisierung von Mayday mit Java Script. Bachelor thesis, Universität Tübingen, 2010.

[129] D. Risso, K. Schwartz, G. Sherlock, and S. Dudoit. GC-Content Normalization for RNA-Seq Data. *UC Berkeley Division of Biostatistics Working Paper Series*, page 291, 2011.

[130] M. Robinson, D. McCarthy, and G. Smyth. edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26(1):139, 2010.

[131] M. Robinson and A. Oshlack. A scaling normalization method for differential expression analysis of RNA-seq data. *Genome Biology*, 11(3):R25, 2010.

[132] J. M. Rothberg, W. Hinz, T. M. Rearick, J. Schultz, W. Mileski, M. Davey, J. H. Leamon, K. Johnson, M. J. Milgrew, M. Edwards, J. Hoon, J. F. Simons, D. Marran, J. W. Myers, J. F. Davidson, A. Branting, J. R. Nobile, B. P. Puc, D. Light, T. A. Clark, M. Huber, J. T. Branciforte, I. B. Stoner, S. E. Cawley, M. Lyons, Y. Fu, N. Homer, M. Sedova, X. Miao, B. Reed, J. Sabina, E. Feierstein, M. Schorn, M. Alanjary, E. Dimalanta, D. Dressman, R. Kasinskas, T. Sokolsky, J. A. Fidanza, E. Namsaraev, K. J. McKernan, A. Williams, G. T. Roth, and J. Bustillo. An integrated semiconductor device enabling non-optical genome sequencing. *Nature*, 475(7356):348–352, 2011.

[133] A. I. Saeed, V. Sharov, J. White, J. Li, W. Liang, N. Bhagabati, J. Braisted, M. Klapa, T. Currier, M. Thiagarajan, A. Sturn, M. Snuffin, A. Rezantsev, D. Popov, A. Ryltsov, E. Kostukovich, I. Borisovsky, Z. Liu, A. Vinsavich, V. Trush, and J. Quackenbush. TM4: a free, open-source system for microarray data management and analysis. *Biotechniques*, 34(2):374, 2003.

[134] F. Sanger and A. Coulson. A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. *Journal of Molecular Biology*, 94(3):441–446, 1975.

[135] F. Sanger, S. Nicklen, and A. Coulson. DNA sequencing with chain-terminating inhibitors. *PNAS*, 74(12):5463, 1977.

[136] Sanger Institute. Protein classification scheme, 2010. [`ftp://ftp.sanger.ac.uk/pub/S_coelicolor/classwise.txt`; accessed 30-Sept-2011].

[137] M. Sawadogo and A. Sentenac. RNA polymerase B (II) and general transcription factors. *Annual Review of Biochemistry*, 59(1):711–754, 1990.

[138] E. E. Schadt, S. A. Monks, T. A. Drake, A. J. Lusis, N. Che, V. Colinayo, T. G. Ruff, S. B. Milligan, J. R. Lamb, G. Cavet, P. S. Linsley, M. Mao, R. B. Stoughton, and S. H. Friend. Genetics of gene expression surveyed in maize, mouse and man. *Nature*, 422(6929):297–302, 2003.

[139] E. Schadt, S. Turner, and A. Kasarskis. A window into third-generation sequencing. *Human Molecular Genetics*, 19(R2):R227, 2010.

[140] M. Schena, D. Shalon, R. Davis, and P. Brown. Quantitative monitoring of gene expression patterns with a complementary DNA microarray. *Science*, 270(5235):467, 1995.

[141] P. Schimmel. Aminoacyl tRNA synthetases: general scheme of structure-function relationships in the polypeptides and recognition of transfer RNAs. *Annual Review of Biochemistry*, 56(1):125–158, 1987.

[142] J. Schuchhardt, D. Beule, A. Malik, E. Wolski, H. Eickhoff, H. Lehrach, and H. Herzel. Normalization strategies for cDNA microarrays. *Nucleic Acids Research*, 28(10):e47, 2000.

[143] P. Shannon, D. Reiss, R. Bonneau, and N. Baliga. The Gaggle: an open-source software system for integrating bioinformatics software and data sources. *BMC Bioinformatics*, 7(1):176, 2006.

[144] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[145] P. Sharp. Splicing of messenger RNA precursors. *Science*, 235(4790):766, 1987.

[146] M. Simonsen, T. Mailund, and C. Pedersen. Rapid neighbour-joining. *Algorithms in Bioinformatics*, pages 113–122, 2008.

[147] S. Singh-Gasson, R. Green, Y. Yue, C. Nelson, F. Blattner, M. Sussman, and F. Cerrina. Maskless fabrication of light-directed oligonucleotide microarrays using a digital micromirror array. *Nature Biotechnology*, 17(10):974–978, 1999.

[148] G. Smyth. Limma: linear models for microarray data. *Bioinformatics and computational biology solutions using R and bioconductor*, pages 397–420, 2005.

[149] E. Southern. Detection of specific sequences among DNA fragments separated by gel electrophoresis. *Journal of Molecular Biology*, 98(3):503–517, 1975.

[150] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[151] M. Sultan, M. H. Schulz, H. Richard, A. Magen, A. Klingenhoff, M. Scherf, M. Seifert, T. Borodina, A. Soldatov, D. Parkhomchuk, D. Schmidt, S. O'Keeffe, S. Haas, M. Vingron, H. Lehrach, and M. L. Yaspo. A global

view of gene activity and alternative splicing by deep sequencing of the human transcriptome. *Science*, 321(5891):956, 2008.

[152] J. Supper, M. Strauch, D. Wanke, K. Harter, and A. Zell. EDISA: extracting biclusters from multiple time-series of gene expression profiles. *BMC Bioinformatics*, 8:334, 2007.

[153] H. Swerdlow and R. Gesteland. Capillary gel electrophoresis for rapid, high resolution DNA sequencing. *Nucleic Acids Research*, 18(6):1415, 1990.

[154] S. Symons. Analysis and Visualization of Gene Expression Data. Dissertation, Universität Tübingen, 2011.

[155] S. Symons and K. Nieselt. MGV: A Generic Graph Viewer for Comparative Omics Data. *Bioinformatics*, 2011.

[156] S. Symons, C. Zipplies, F. Battke, and K. Nieselt. Integrative Systems Biology Visualization with MAYDAY. *Journal of Integrative Bioinformatics*, 7(3):115, 2010.

[157] P. 't Hoen, Y. Ariyurek, H. Thygesen, E. Vreugdenhil, R. Vossen, R. De Menezes, J. Boer, G. Van Ommen, and J. Den Dunnen. Deep sequencing-based expression analysis shows major advances in robustness, resolution and inter-lab portability over five microarray platforms. *Nucleic Acids Research*, 36(21):e141, 2008.

[158] The Linux Documentation Project. Memory management, 2011. [`http://tldp.org/LDP/tlk/mm/memory.html`; accessed 21-Sept-2011].

[159] L. Thomas, D. A. Hodgson, A. Wentzel, K. Nieselt, H. Sletta, T. E. Ellingsen, J. Moore, E. R. Morrissey, R. Legaie, The STREAM Consortium, W. Wohlleben, A. Rodríguez-García, J. F. Martín, N. J. Burroughs, E. M. H. Wellington, and M. C. M. Smith. Metabolic switches and adaptations deduced from the proteomes of *Streptomyces coelicolor* wild type and *phoP* mutant grown in batch culture. *Molecular & Cellular Proteomics*, 11(2), 2012.

[160] C. Trapnell, L. Pachter, and S. Salzberg. TopHat: discovering splice junctions with RNA-Seq. *Bioinformatics*, 25(9):1105–1111, 2009.

[161] C. Trapnell, B. Williams, G. Pertea, A. Mortazavi, G. Kwan, M. Van Baren, S. Salzberg, B. Wold, and L. Pachter. Transcript assembly and quantification by RNA-Seq reveals unannotated transcripts and isoform switching during cell differentiation. *Nature Biotechnology*, 28(5):511–515, 2010.

[162] N. Trunk. Illumina array algorithms for Mayday. Diplomarbeit, Universität Tübingen, 2009.

[163] S. Urbanek. A fast way to provide R functionality to applications. In *Proceedings of DSC*, page 2, 2003.

[164] S. Urbanek. *rJava: Low-level R to Java interface*, 2010. R package version 0.8-8.

[165] Z. Šidák. Rectangular confidence regions for the means of multivariate normal distributions. *Journal of the American Statistical Association*, pages 626–633, 1967.

[166] C. Vehlow, J. Heinrich, F. Battke, D. Weiskopf, and K. Nieselt. iHAT: Interactive hierarchical aggregation table. In *Biological Data Visualization (BioVis),*

2011 IEEE Symposium on, pages 63–69, Oct. 2011.

[167] V. Velculescu, L. Zhang, B. Vogelstein, and K. Kinzler. Serial Analysis of Gene Expression. *Science*, 270(5235):484, 1995.

[168] W. Venables and B. Ripley. *S Programming*, 2000.

[169] G. Von Heijne. The structure of signal peptides from bacterial lipoproteins. *Protein Engineering*, 2(7):531, 1989.

[170] M. C. Wahl, C. L. Will, and R. Lührmann. The Spliceosome: Design Principles of a Dynamic RNP Machine. *Cell*, 136(4):701 – 718, 2009.

[171] E. Waldvogel, A. Herbig, F. Battke, R. Amin, M. Nentwich, K. Nieselt, T. E. Ellingsen, A. Wentzel, D. A. Hodgson, W. Wohlleben, and Y. Mast. The P_{II} protein GlnK is a pleiotropic regulator for morphological differentiation and secondary metabolism in *Streptomyces coelicolor*. *Applied Microbiology and Biotechnology*, pages 1–18, 2011.

[172] Y. Wang, G. Mehta, R. Mayani, J. Lu, T. Souaiaia, Y. Chen, A. Clark, H. J. Yoon, L. Wan, O. V. Evgrafov, J. A. Knowles, E. Deelman, and T. Chen. RseqFlow: Workflows for RNA-Seq data analysis. *Bioinformatics*, 2011.

[173] Z. Wang, M. Gerstein, and M. Snyder. RNA-Seq: a revolutionary tool for transcriptomics. *Nature Reviews Genetics*, 10(1):57–63, 2009.

[174] D. Weese, A. Emde, T. Rausch, A. Döring, and K. Reinert. RazerS – fast read mapping with sensitivity control. *Genome Research*, 19(9):1646, 2009.

[175] X. Wen, S. Fuhrman, G. Michaels, D. Carr, S. Smith, J. Barker, and R. Somogyi. Large-scale temporal gene expression mapping of central nervous system development. *PNAS*, 95(1):334, 1998.

[176] T. Werner. Next generation sequencing in functional genomics. *Briefings in Bioinformatics*, 11(5):499, 2010.

[177] Wikipedia, The Free Encyclopedia. Memory-mapped file, 2011. [`http://en.wikipedia.org/w/index.php?title=Memory-mapped_file&oldid=448078496`; accessed 21-Sept-2011].

[178] M. Wilk and R. Gnanadesikan. Probability plotting methods for the analysis for the analysis of data. *Biometrika*, 55(1):1–17, 1968.

[179] L. Wilkinson and M. Friendly. The history of the cluster heat map. *The American Statistician*, 63(2):179–184, 2009.

[180] R. Williams, S. Peisajovich, O. Miller, S. Magdassi, D. Tawfik, and A. Griffiths. Amplification of complex gene libraries by emulsion PCR. *Nature Methods*, 3(7):545, 2006.

[181] Z. Wu, B. Jenkins, T. Rynearson, S. Dyhrman, M. Saito, M. Mercier, and L. Whitney. Empirical bayes analysis of sequencing-based transcriptional profiling without replicates. *BMC Bioinformatics*, 11(1):564, 2010.

[182] Z. Wu, X. Wang, and X. Zhang. Using non-uniform read distribution models to improve isoform expression inference in RNA-Seq. *Bioinformatics*, 27(4):502, 2011.

[183] Y. Yang, S. Dudoit, P. Luu, D. Lin, V. Peng, J. Ngai, and T. Speed. Normalization for cDNA microarray data: a robust composite method addressing single

and multiple slide systematic variation. *Nucleic Acids Research*, 30(4):e15, 2002.

[184] D. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research*, 18(5):821, 2008.

[185] C. Zipplies. Skalierbare Methoden zur interaktiven Visualisierung genomweiter Genexpressionsdaten. Diplomarbeit, Universität Tübingen, 2009.

[186] M. Zschunke. Connecting R to Mayday. Studienarbeit, Universität Tübingen, 2004.

[187] M. Zschunke. Visualisierung und Analyse von ChIP on Chip-Experimenten. Diplomarbeit, University of Tübingen, 2006.

A. Publications

A.1. Articles

2008

- **Florian Battke**, Carsten Müller-Tidow, Hubert Serve, and Kay Nieselt. *Post-Hybridization Quality Measures for Oligos in Genome-Wide Microarray Experiments.* Proceedings of the 8th international workshop on Algorithms in Bioinformatics 2008: 64–75

2010

- **Florian Battke**, Stephan Symons, and Kay Nieselt. *Mayday – Integrative analytics for expression data.* BMC Bioinformatics 2010, 11:1
- **Florian Battke, Stephan Körner**, Steffen Hüttner, and Kay Nieselt. *Efficient sequence clustering for RNA-seq data without a reference genome.* Lecture notes in Informatics (LNI) – Proceedings (2010), P-173:21–30
- **Kay Nieselt, Florian Battke, Alexander Herbig**, Per Bruheim, Alexander Wentzel, Øyvind Jakobsen, Håvard Sletta, Tauqueer Alam, Elena Merlo, Jay Moore, Walid Omara, Edward Morrissey, Miguel Juarez-Hermosillo, Lubbert Dijkhuizen, David Rand, David Wild, Michael Bonin, Jens Reuther, Wolfgang Wohlleben, Margaret Smith, Nigel Burroughs, Juan Martín, David Hodgson, Eriko Takano, Rainer Breitling, Trond Ellingsen, and Elizabeth Wellington. *The dynamic architecture of the metabolic switch in Streptomyces coelicolor.* BMC Genomics 2010, 11:10
- **Stephan Symons**, Christian Zipplies, Florian Battke, and Kay Nieselt. *Integrative Systems Biology Visualization with Mayday.* Journal of Integrative Bioinformatics 2010, 7:3

2011

- **Florian Battke** and Kay Nieselt. *Mayday SeaSight: Combined Analysis of Deep Sequencing and Microarray Data.* PLoS ONE 2011, 6:1
- **Florian Battke, Alexander Herbig**, Alexander Wentzel, Øyvind M. Jakobsen, Michael Bonin, Dave A. Hodgson, Wolfgang Wohlleben, Trond E. Ellingsen, and Kay Nieselt. *A Technical Platform for Generating Reproducible Expression Data from Streptomyces coelicolor Batch Cultivations.* Advances in Experimental Medicine and Biology: Software Tools and Algorithms for Biological Systems

- **Florian Battke, Stephan Symons**, Alexander Herbig, and Kay Nieselt. *GaggleBridge: Collaborative data analysis.* Bioinformatics (2011) 27 (18): 2612–2613
- **Günter Jäger**, Florian Battke, and Kay Nieselt. *TIALA – Time Series Alignment Analysis.* Biological Data Visualization (BioVis), 2011 IEEE Symposium on, pages 55–61, Oct. 2011
- **Lucía Spangenberg**, Florian Battke, Martín Graña, Kay Nieselt, and Hugo Naya. *Identification of associations between amino acid changes and meta information in alignments.* Bioinformatics (2011) 27 (20): 2782–2789
- **Corinna Vehlow**, Julian Heinrich, Florian Battke, Daniel Weiskopf, and Kay Nieselt. *iHat – the interactive hierarchical aggregation table.* Biological Data Visualization (BioVis), 2011 IEEE Symposium on, pages 55–61, Oct. 2011
- **Eva Waldvogel, Alexander Herbig**, Florian Battke, Merle Nentwich, Kay Nieselt, Trond E. Ellingsen, David A. Hodgson, Wolfgang Wohlleben, and Yvonne Mast. *The P_{II} protein GlnK is a pleiotropic regulator for morphological differentiation and secondary metabolism in Streptomyces coelicolor.* Applied Microbiology and Biotechnology 2011: 1–18

2012

- **Alexander Herbig, Florian Battke, Günter Jäger**, and Kay Nieselt. *GenomeRing: alignment visualization in SuperGenome coordinates.* Bioinformatics 2012, 28(12):i7–i15 (Proceedings of the ISMB 2012)
- **Günter Jäger**, Florian Battke, and Kay Nieselt. *Reveal – Visual eQTL analytics.* Bioinformatics 2012, 28(18):i542–i548 (Proceedings of the ECCB 2012)
- **Peter Nestorov**, Florian Battke, Mitchell P. Levesque, and Matthias Gerberding. *The maternal transcriptome of the crustacean Parhyale hawaiiensis is inherited asymmetrically to invariant cell lineages of the ectoderm and mesoderm.* (under review)
- **Julian Heinrich**, Corinna Vehlow, Florian Battke, Günter Jäger, Daniel Weiskopf, and Kay Nieselt. *iHAT: interactive Hierarchical Aggregation Table for Genetic Association Data.* BMC Bioinformatics 2012, 13(Suppl 8):S2
- **Christopher W. Bartlett**, Soo Yeon Cheong, Liping Hou, Jesse Paquette, Pek Yee Lum, Günter Jäger, Florian Battke, Corinna Vehlow, Julian Heinrich, Kay Nieselt, Ryo Sakai, Jan Aerts, and William C. Ray. *An eQTL biological data visualization challenge and approaches from the visualization community.* BMC Bioinformatics 2012, 13(Suppl 8):S8

A.2. Posters & Presentations

2006

- Stephan Symons, Florian Battke, Janko Dietzsch, Matthias Zschunke, and Kay Nieselt. *Automated Processing and Machine Learning Tools for Mayday.* German Conference on Bioinformatics 2006

2007

- Stephan Symons, Christian Schillinger, Janko Dietzsch, Florian Battke, and Kay Nieselt. *GeneMining in Mayday, a feature selection framework for binary classification.* German Conference on Bioinformatics 2007
- Carsten Müller-Tidow, Claudia Homme, Hans-Ulrich Klein, Antje Hascher, Steffen Koschmieder, Anke Becker, Yipeng Wang, Michael McClelland, Udo zur Stadt, Florian Battke, Kay Nieselt, Christian Thiede, Gerhard Ehninger, Wolfgang E. Berdel, Martin Dugas, and Hubert Serve. *Defining the Leukemia Epigenome: Distinct Genome Wide Histone H3 Modification Patterns Exist in AML, ALL and Healthy Hematopoietic Progenitor Cells.* Blood (ASH Annual Meeting Abstracts) 2007, 110:11: Abstract 2124

2008

- Florian Battke, Stephan Symons, Michael Piechotta, Philipp Bruns, Karin Zimmermann, and Kay Nieselt. *Mayday – Microarray data analysis.* German Conference on Bioinformatics 2008

2009

- Florian Battke, Stephan Symons, and Kay Nieselt. *Mayday and RLink – Integrated Expression Analysis.* German Conference on Bioinformatics 2009
- Florian Battke, Stephan Symons, and Kay Nieselt. *Mayday RLink – The best of two worlds.* **Presentation** at useR 2009
- Florian Battke. *Mayday - Visual Analytics for Expression Data.* **Invited Talk** at Humbold University Berlin, Knowledge Management in Bioinformatics

2010

- Florian Battke, Stephan Körner, Steffen Hüttner, and Kay Nieselt. *Efficient sequence clustering for RNA-seq data without a reference genome.* **Presentation** at the German Conference on Bioinformatics 2010
- Stephan Symons, Florian Battke, Christian Zipplies, and Kay Nieselt. *Mayday – Integrative Visual Analytics for Transcriptomics.* VizBi 2010

2011

- Florian Battke, Steffen Hüttner, and Kay Nieselt. *PASSAGE: A fast and efficient sequence clustering method for RNA-seq data without a reference genome.* Dechema Seminar “Functional Genomics – Next Generation Applications and Technologies” 2011
- Florian Battke, Stephan Symons, Günter Jäger, Aydın Can Polatkan, Alexander Herbig, and Kay Nieselt. *GenomeRing: Visual Comparison of Multiple Genomes.* **Video Presentation** at the Illumina iDEA challenge, San Diego – Winner of the **Most Creative Algorithm** award (academic entries)

- Florian Battke, and Kay Nieselt. *Microarray-Analyseprogramm Mayday: Rettungsanker in der Datenflut.* Laborjournal 4/2010 (software presentation, not peer reviewed)
- Günter Jäger, Florian Battke, and Kay Nieselt. *Tiala - Visual Time Series Alignment Analysis.* German Conference on Bioinformatics 2011
- Julian Heinrich, Corinna Vehlow, Kay Nieselt, Florian Battke, and Daniel Weiskopf. *iHAT – interactive Hierarchical Aggregation Table.* VizBi 2011