

# IOWA STATE UNIVERSITY

## Digital Repository

---

### Retrospective Theses and Dissertations

---

2008

## Interactive graphics, graphical user interfaces and software interfaces for the analysis of biological experimental data and networks

Michael Lawrence  
*Iowa State University*

Follow this and additional works at: <http://lib.dr.iastate.edu/rtd>

 Part of the [Bioinformatics Commons](#), [Computer Sciences Commons](#), and the [Statistics and Probability Commons](#)

---

#### Recommended Citation

Lawrence, Michael, "Interactive graphics, graphical user interfaces and software interfaces for the analysis of biological experimental data and networks" (2008). *Retrospective Theses and Dissertations*. Paper 15881.

This Dissertation is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact [digirep@iastate.edu](mailto:digirep@iastate.edu).

**Interactive graphics, graphical user interfaces and software interfaces for the  
analysis of biological experimental data and networks**

by

Michael Lawrence

A dissertation submitted to the graduate faculty  
in partial fulfillment of the requirements for the degree of  
**DOCTOR OF PHILOSOPHY**

Major: Bioinformatics and Computational Biology

Program of Study Committee:  
Dianne Cook, Co-major Professor  
Eve Wurtele, Co-major Professor  
Heike Hofmann  
Julie Dickerson  
Les Miller  
Jean Peccoud

Iowa State University

Ames, Iowa

2008

Copyright © Michael Lawrence, 2008. All rights reserved.

UMI Number: 3307049

#### INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.



---

UMI Microform 3307049  
Copyright 2008 by ProQuest LLC  
All rights reserved. This microform edition is protected against  
unauthorized copying under Title 17, United States Code.

---

ProQuest LLC  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106-1346

**TABLE OF CONTENTS**

<b>LIST OF TABLES . . . . .</b>	viii
<b>LIST OF FIGURES . . . . .</b>	ix
<b>INTRODUCTION . . . . .</b>	1
1    Problem Statement . . . . .	1
2    Overview . . . . .	1
3    Scope . . . . .	7
<b>RGTK2: A GRAPHICAL USER INTERFACE TOOLKIT FOR R . . . . .</b>	9
Abstract . . . . .	9
1    Introduction . . . . .	9
2    Fundamentals . . . . .	13
2.1    GTK+ Widgets . . . . .	13
2.2    GTK+ Widgets in R . . . . .	15
2.3    Widget Layout . . . . .	21
3    Basic GUI Construction . . . . .	25
3.1    A Dialog with the User . . . . .	25
3.2    Giving the User More Options . . . . .	28
3.3    The CRAN Mirrors Dialog . . . . .	31
3.4    Embedded R Graphics . . . . .	34
4    A Sample Application . . . . .	38
4.1    Main window . . . . .	39
4.2    Menu bar and tool bar . . . . .	39
4.3    Status bar . . . . .	44

4.4	Spreadsheet . . . . .	45
4.5	Loading a spreadsheet . . . . .	47
5	Advanced Features . . . . .	49
5.1	Additional Library Support . . . . .	49
5.2	A GObject Primer . . . . .	51
5.3	Interfacing With External GObject-based Applications . . . . .	53
5.4	Defining GObject Classes . . . . .	55
6	Language binding design and generation . . . . .	59
6.1	Goals and Scope . . . . .	59
6.2	Automatic Binding Generation . . . . .	60
6.3	Type Conversion . . . . .	69
6.4	Autogeneration of the Documentation . . . . .	73
7	Technical Language Binding and GUI Issues . . . . .	74
7.1	Fully Programmatic Binding Generation . . . . .	74
7.2	RGtk2 as a Base for Other GObject Bindings . . . . .	75
7.3	Event Loop Issues . . . . .	76
8	Comparison of RGtk2 to other R GUI toolkit bindings . . . . .	77
9	Impact and Future Work . . . . .	79
<b>THE PLUMBING OF INTERACTIVE GRAPHICS . . . . .</b>		<b>81</b>
1	Introduction . . . . .	81
2	Related work . . . . .	82
3	What is a pipeline? . . . . .	83
4	Coordinating the pipeline . . . . .	84
5	Pipes vs plumbing . . . . .	87
6	Motivation and implementation . . . . .	89
7	Conclusions . . . . .	90
<b>AN INTRODUCTION TO RGGOBI . . . . .</b>		<b>92</b>
1	Introduction . . . . .	92

2	Data . . . . .	93
3	Modifying observation-level attributes, or “automatic brushing” . . . . .	93
4	Displays . . . . .	95
5	Animation . . . . .	97
6	Edge data . . . . .	98
6.1	Longitudinal data . . . . .	99
7	Case study . . . . .	100
8	Conclusion . . . . .	103
<b>EXTENDING THE GGOBI PIPELINE FROM R . . . . .</b>		105
	Abstract . . . . .	105
1	Introduction . . . . .	105
2	Low-level interface to GGobi . . . . .	108
3	Application to interactive network layout . . . . .	111
4	Conclusion . . . . .	119
<b>RSBML: AN R PACKAGE FOR IMPORTING SBML DOCUMENTS USING LIBSBML . . . . .</b>		120
	Abstract . . . . .	120
1	Introduction . . . . .	120
2	Features . . . . .	121
3	Demonstration . . . . .	122
4	Future . . . . .	124
<b>AN EXTENSIBLE SOFTWARE SYSTEM FOR INTERACTIVE NETWORK VISUALIZATIONS IN THE CONTEXT OF DATA ANALYSIS . . . . .</b>		125
	Abstract . . . . .	125
1	Introduction . . . . .	125
2	Existing graph layout techniques . . . . .	131
2.1	Orthogonal layout . . . . .	131
2.2	Force-directed layout . . . . .	131

2.3	Hierarchical layout . . . . .	132
2.4	Constraint-based layout . . . . .	133
2.5	The IPSep-CoLa algorithm . . . . .	134
3	Interactive graph layout . . . . .	135
4	The rcola package . . . . .	136
4.1	Integration with IPSep-CoLa . . . . .	137
4.2	Network views . . . . .	140
4.3	The GUI . . . . .	142
5	Conclusion . . . . .	142
 <b>INTERACTIVE BIOCHEMICAL NETWORK VISUALIZATION FOR EXPERIMENTAL DATA ANALYSIS . . . . .</b> 144		
Abstract . . . . . 144		
1	Introduction . . . . .	144
2	Existing biology-aware layout algorithms . . . . .	147
3	Biochemical network layout using separation constraints . . . . .	151
4	Software implementation . . . . .	153
5	Conclusion . . . . .	156
 <b>EXPLORASE: MULTIVARIATE EXPLORATORY ANALYSIS AND VISUALIZATION FOR SYSTEMS BIOLOGY . . . . .</b> 157		
Abstract . . . . . 157		
1	Introduction . . . . .	158
2	Overview . . . . .	160
3	Methods . . . . .	163
3.1	Numerical methods . . . . .	163
3.2	Graphical methods . . . . .	166
4	GUI features . . . . .	170
4.1	Main panels . . . . .	170
4.2	Toolbar . . . . .	173

4.3	Menubar . . . . .	174
5	Getting started . . . . .	178
6	Demonstration . . . . .	180
7	Technical considerations . . . . .	187
7.1	Suggested limit on number of samples . . . . .	187
7.2	Software infrastructure . . . . .	188
8	Related work . . . . .	188
9	Conclusion . . . . .	189
<b>METHODS AND DIAGNOSTIC TOOLS FOR THE ANALYSIS OF GC-MS METABOLOMICS DATA . . . . .</b>		190
	Abstract . . . . .	190
1	Introduction . . . . .	191
2	Raw Input Data . . . . .	194
2.1	Description . . . . .	194
2.2	Preparation . . . . .	194
3	Data Analysis Pipeline . . . . .	195
3.1	Baseline Subtraction . . . . .	195
3.2	Peak Detection . . . . .	203
3.3	Component Detection . . . . .	211
3.4	Grouping Components Between Samples . . . . .	216
3.5	Retention Time Correction . . . . .	223
3.6	Summarization . . . . .	229
3.7	Normalization . . . . .	232
4	Discussion . . . . .	235
4.1	Peak Convolution . . . . .	235
4.2	Metabolite Identification . . . . .	238
5	Conclusion . . . . .	238

<b>CONCLUSION . . . . .</b>	<b>240</b>
1    Impact . . . . .	240
2    Future work . . . . .	241
3    Summary . . . . .	242
<b>APPENDIX</b>	
Data Analysis Examples . . . . .	243
<b>BIBLIOGRAPHY . . . . .</b>	<b>275</b>

**LIST OF TABLES**

Table 1	Interactive graphics software with an explicit pipeline. . . . .	89
Table 1	Filenaming conventions . . . . .	180
Table A.1	Identities and functions of the genes identified to be shifting to a higher state in the wildtype after 90 minutes of light stress. . . . .	255
Table A.2	Genes responding in the mutant but not in wildtype. Specifically, these are the genes increasing at 15 minutes and decreasing at 3 hours. . . .	261
Table A.3	Number of replicates for each biological sample analyzed by metabolite profiling. . . . .	271
Table A.4	List of metabolites negatively correlated with PHB in plants grown under long day condition. . . . .	272

## LIST OF FIGURES

Figure 1	Widget hierarchies in GTK+ . . . . .	13
Figure 2	The GTK+ class hierarchy . . . . .	15
Figure 3	Hello World in GTK+ . . . . .	16
Figure 4	Packing widgets into boxes . . . . .	23
Figure 5	Message dialog . . . . .	27
Figure 6	Message dialog with checkbox . . . . .	29
Figure 7	Message dialog with radio buttons . . . . .	30
Figure 8	Message dialog with combo box . . . . .	32
Figure 9	Embedded scatterplots . . . . .	35
Figure 10	Embedded scatterplots with adjusted alpha . . . . .	35
Figure 11	Spreadsheet application . . . . .	40
Figure 12	Example of defs format . . . . .	61
Figure 1	Simple pipeline . . . . .	85
Figure 2	Simple pipeline with reverse pipeline . . . . .	85
Figure 3	Pipeline with multiple plots. . . . .	86
Figure 4	Pipeline with plot-to-plot updates. . . . .	86
Figure 5	Pipeline with central commander to control flow of updates. . . . .	87
Figure 6	Pipeline with two-stage event model, where events follow the pipeline.	88
Figure 7	Goal of new GGobi pipeline . . . . .	90
Figure 1	Application to interactive graph layout . . . . .	112

Figure 1	Cytoscape spring-embedded layout of a few steps of biotin synthesis and the production of MCCase . . . . .	129
Figure 2	Dot layout (in Cytoscape) of the same network shown in 1 . . . . .	130
Figure 3	Design of rcola . . . . .	138
Figure 4	The GGobi-based rcola network view . . . . .	141
Figure 1	Compound layout issues . . . . .	150
Figure 2	Reaction constraint . . . . .	154
Figure 3	GGobi network view with reaction constraint GUI . . . . .	155
Figure 1	Main GUI of exploRase . . . . .	161
Figure 2	Limma results . . . . .	167
Figure 3	Filter GUI . . . . .	171
Figure 4	Experimental design table . . . . .	172
Figure 5	Interactive R cluster tree graphic . . . . .	175
Figure 6	Pattern finder . . . . .	176
Figure 7	Linear modeling front-ends . . . . .	177
Figure 8	Subsetting GUI . . . . .	178
Figure 9	Calculating differences . . . . .	181
Figure 10	GGobi control panel . . . . .	184
Figure 11	Limma results for Biotin data . . . . .	185
Figure 1	Raw profile image . . . . .	196
Figure 2	Raw chromatogram at 51 m/z . . . . .	197
Figure 3	RBE fit at 51 m/z . . . . .	199
Figure 4	Median filter fit at 51 m/z . . . . .	200
Figure 5	Baseline-subtracted profile image . . . . .	202
Figure 6	Baseline subtraction diagnostic visualization . . . . .	203
Figure 7	Local maxima detection at 51 and 72 m/z . . . . .	206
Figure 8	Gaussian and EGH peak fits . . . . .	209

Figure 9	Peak image . . . . .	210
Figure 10	Peak detection diagnostic visualization . . . . .	212
Figure 11	Component image . . . . .	214
Figure 12	Scatterplot of components by $\sigma$ and $\sigma_t$ . . . . .	215
Figure 13	Component detection diagnostic visualization . . . . .	217
Figure 14	Experimental design tree . . . . .	219
Figure 15	Scatterplot of groups with $\sigma_t$ on horizontal axis and $\mu_d$ on vertical axis.	220
Figure 16	Component grouping diagnostic visualization . . . . .	222
Figure 17	Robust loess fits for retention time correction . . . . .	226
Figure 18	Overlaid Total Ion Current (TIC) chromatograms using raw and corrected times . . . . .	227
Figure 19	Barchart comparing the distribution of the group sizes before and after applying the retention time window constraint. . . . .	228
Figure 20	Retention time correction diagnostic visualization . . . . .	230
Figure 21	Comparison of quantity distribution before and after normalization . .	232
Figure 22	Summarization diagnostic visualization . . . . .	233
Figure 23	Scatterplot matrix comparing the normalized metabolite levels between the control replicates. . . . .	236
Figure A.1	Scatterplot matrix of 0 and 15 minute replicates . . . . .	248
Figure A.2	Scatterplot matrix of 0 and 90 minute replicates . . . . .	249
Figure A.3	Pattern-finder results . . . . .	250
Figure A.4	Entity table sorted by pattern magnitude . . . . .	251
Figure A.5	Results of fitting linear model . . . . .	252
Figure A.6	Parallel coordinate plot of genes with significant effects . . . . .	253
Figure A.7	Barcharts of the maxima and minima of the set of $t$ -statistics for each gene . . . . .	254
Figure A.8	The distribution of biological functions for the genes found to be reaching a higher state at 90 minutes in the wildtype. . . . .	257

Figure A.9	Scatterplots of the time coefficients vs. their p-values for each gene. . .	258
Figure A.10	Parallel coordinate plot of genes that were not significant with respect to time in the wildtype . . . . .	258
Figure A.11	Barcharts for the maxima and minima of the set of <i>t</i> -statistics calculated for each gene in the mutant. . . . .	259
Figure A.12	Barchart of the biological function distribution of the genes in Table 2	263
Figure A.13	Plot of genes correlated to AT3G02310 (SEP2) . . . . .	267
Figure A.14	Profile of ACLA-2 . . . . .	268
Figure A.15	Profiles of three genes related to starch synthesis . . . . .	269
Figure A.16	Plots of relative intensity versus PHB accumulation for each of the compounds listed in Table A.4. . . . .	272
Figure A.17	Screenshot of exploRase during analysis . . . . .	273
Figure A.18	Screenshots of limma results . . . . .	274

## INTRODUCTION

### 1 Problem Statement

Biologists need to analyze and comprehend increasingly large and more complex experimental data. These experimental data are multivariate, where each row corresponds to a biological entity, and each column corresponds to the level of an experimental treatment. Biological experiments often produce multiple data sets, each describing one aspect of the system, such as the transcriptome recorded by a microarray or metabolome recorded using gas chromatography mass spectrometry (GC-MS). A biochemical network model provides a conceptual system-level framework for integrating data from different sources.

Effective use of graphics enhances the comprehension of data, and interactive graphics permit the analyst to actively explore data, check its integrity, satiate curiosities and reveal the unexpected. Interactive graphics have not been widely applied as a means for understanding data from biological experiments.

This thesis addresses these needs by providing new methods and software that apply interactive graphics in coordination with numerical methods to the analysis of biological data, in a manner that is accessible to biologists.

### 2 Overview

This thesis focuses on graphics-intensive analysis of biological data. The overall research themes are:

- Combine interactive graphics and numerical methods for the analysis of biological experimental data and networks.

- Incorporate biochemical network views into experimental data analysis, as a framework for combining data from different sources, and to understand the experimental data results in the context of cellular functions.
- Improve access to sophisticated data analysis tools for biologists through the use of graphical user interfaces.
- Leverage, improve and interface existing and new software in support of biological data analysis.

It is organized as a collection of nine independent papers, each related to the themes listed above. The first six papers each contain a paper describing the software foundation that supports the biology-specific tools and methods presented in the last three papers. The appendix contains examples of using the software to analyze biological data. A synopsis of each paper is given below.

**RGtk2: A Graph User Interface Toolkit for R** [RGtk2](#) is a low-level software interface between R and the GTK+ 2.0 widget library. The purpose of this package is to enable R programmers to provide a graphical user interface (GUI), consisting of menus, buttons, sliders, etc, to their analytical methods. A separate package, cairoDevice, embeds R graphics, as drawn by the Cairo graphics library, in RGtk2 interfaces. GUIs enable beginning, or transient, R users to apply the methods to their data without the need to learn the R language. GUIs are particularly useful in interactive graphics applications, so RGtk2 helps bring about the synergy between interactive graphics and numerical methods. More generally, RGtk2 is a research project in the domain of interfacing software systems. In addition to being a low-level interface to a GUI toolkit, RGtk2 interfaces two systems, the GObject C library and R, that are very different in design and purpose. Bridging these semantic differences is the primary challenge of RGtk2 development. Also challenging is the large scope of the interface. The libraries bound by RGtk2 consist of hundreds of data structures and thousands of functions. The autogenerated interface is notable as an application of metaprogramming.

**The Plumbing of Interactive Graphics** The design of the GGobi data pipeline is introduced. GGobi is a tool for viewing data in interactive multivariate displays, such as scatterplots, parallel coordinate plots and barcharts. The scatterplot display supports edges between points, enabling the visualization of networks and other structures. Interactive features include brushing, identification, pan and zoom, moving points, and editing edges. All of these features rely on the data pipeline, which is responsible for converting input data into a form suitable for plotting, as well as mapping user interaction back to the original data. The design of the pipeline is modular and extensible, facilitating the customization of interactive visualizations for a particular analysis task. The pipeline may be manipulated and modified by plugins or from environments, such as R, that are capable of embedding GGobi. There has also been work on making GGobi more accessible to users, such as most biologists, who are not software developers. The build system has been rewritten, facilitating cross-platform distribution of GGobi. This has led to reliable, automatic installers for GGobi on Windows and Mac OS X. The stability, functionality, usability and overall look and feel of the GGobi GUI was improved by the move to a modern GUI toolkit, GTK+ 2.0.

**An Introduction to rggobi** The [rggobi package](#) is an interface from R to GGobi. There are many statistical methods implemented in R, but R lacks interactive graphics. By connecting R to GGobi, rggobi forms the synergistic link between interactive graphics and numerical methods. The focus of the paper is on the high-level interface from R to GGobi that facilitates the movement of data between GGobi and R, the management of the visual attributes (i.e. color) on observations and the creation and configuration of plots. The high-level API is designed to be consistent, familiar, intuitive and convenient for the R programmer. There are facilities for loading longitudinal data and R graph objects [Gentleman et al., 2007] into GGobi as graphs. Integration with RGtk2 allows the user to modify the GGobi GUI as well as attach R callbacks to GGobi events.

**Extending the GGobi Pipeline from R** The [latest generation of rggobi](#) leverages RGtk2 to provide a low-level interface to GGobi. The low-level interface coexists with the

original high-level interface. With the low-level interface, R users can program directly against the public API of any GGobi module. It is possible to define classes in R that extend GGobi classes, including pipeline stages, to customize visualizations and other aspects of GGobi. This enables rggobi to serve as an environment for the rapid prototyping of interactive visualizations. While it offers more direct control over GGobi compared to the high-level interface, the low-level interface does not as seamlessly integrate with the design of R. The relative tradeoffs of high- and low-level software interfaces is the engineering research focus of rggobi.

**Rsbml: an R Package for Importing SBML Documents using libsbml** Incorporating biochemical networks into experimental data analysis depends on a means for representing network data within the data analysis environment. Currently, there is no standard for the general representation of complex biological networks. However, the Systems Biology Markup Language (SBML) is the de facto standard for the representation of metabolic pathways in XML. As there does not yet exist a viable solution for loading SBML into R, a new package, [rsbml](#), has been developed for this purpose. It is capable of converting SBML directly into R graph objects, which may be passed to GGobi and other tools.

**An Extensible Software System for Interactive Network Visualizations in the Context of Data Analysis** The graphs provided by rsbml need to be drawn, whether in GGobi or some other viewer, in a way that is appropriate for the current analysis task. As the focus of an analysis changes over time, the layout algorithm needs to adapt the graph drawing in an incremental, non-disruptive manner. The [rcola](#) package allows the user to specify high-level constraints on a selected set of nodes. For example, the user can spatially segregate nodes according to a categorical variable associated with the network. This functionality is based on IPSep-CoLa [Dwyer and Marriott, 2006], a force-directed layout algorithm that supports general separation constraints. The rcola package provides a high-level interface from R to libcola, a C++ implementation of IPSep-CoLa, as well as a GUI for viewing networks and applying layout constraints. The GUI is written against a toolkit-independent API provided by the gWidgets package [Verzani, 2007a]. This allows embedding network views implemented

in different GUI toolkits. A view based on GGobi is included with the package, but more can be added through the extensible API.

### **Interactive Biochemical Network Visualization for Experimental Data Analysis**

The rcola approach has been [extended](#) to incorporate biological semantics into network drawings. Biological information could be communicated through node color and shape schemes, but these are not always intuitive. Rather, it is often preferable to incorporate biological meaning directly into the layout of the network. Towards this end, we have developed an approach based on the constrained layout algorithm provided by the rcola package. Two additional constraints are provided on top of those from rcola. The first arranges reactions so that they are easier to interpret, and the other segregates the nodes into disjoint convex regions according to their compartment.

### **ExploRase: Multivariate Exploratory Analysis and Visualization for Systems Biology**

The [exploRase](#) package is designed for the exploratory analysis of preprocessed high-throughput experimental data. The exploRase package is part of the MetNet project [Wurtele et al., 2003] and provides a simple biologist-friendly GUI that acts as a front-end to R and GGobi. In this way, exploRase integrates numerical methods with interactive graphics. The exploRase GUI displays entities, such as genes or metabolites, along with their annotations and analysis results, in a sortable, searchable and filterable spreadsheet. The table is synchronized with GGobi plots by visual cues. The exploRase GUI provides access to a number of numerical methods for comparing entity levels across conditions, calculating distances between entities, finding entities with a given pattern, and clustering entities. In addition, there are two linear modeling methods, one based on the limma package [Smyth, 2005] and the other for time course modeling. As it is written in R, exploRase uses RGtk2 to constructs its GUI and rggobi for displaying interactive plots. The biochemical network visualization feature is based on rcola. An R Application Programming Interface (API) is available for scripting exploRase.

## Methods and Diagnostic Tools for the Analysis of GC-MS Metabolomics Data

We have developed [methods and software](#) for preprocessing data generated by Gas Chromatography Mass Spectrometry (GC-MS) metabolomics experiments. The methods build on the foundation presented thus far to achieve a synergy of numerical methods, interactive graphics and GUIs. The preprocessing steps include separating the metabolite signal, the peaks, from the noise and matching the peaks across samples to obtain the levels of each metabolite in each sample. The pipeline is implemented in an R package called chromatoplot. The name *chromatoplot* is derived from “chroma,” indicating the type of data, and “plots,” which emphasizes the use of statistical graphics. A visualization is provided for each preprocessing stage. The main purpose of the visualizations is to help the user evaluate the results of the algorithms and diagnose any problems. A secondary goal is to assist the user in understanding the effect of parameter settings on algorithm output. The package will provide a GUI to make its functionality accessible to those who are not expert users of R. The GUI design, currently being planned, is based on the wizard pattern, and the user is guided through the preprocessing stages step by step. The GUI to each stage consists of a interactive diagnostic visualization and a panel for specifying algorithm parameters. After a stage is completed, the user is allowed to progress to the next stage in the pipeline. It is possible to jump back to a previous stage at any time. The chromatoplot package is written in the R statistical language and is based on xcms, an R package that provides a common framework for analyzing LC-MS and GC-MS data [Smith et al., 2006]. The R language facilitates the implementation and prototyping of the statistical and graphical methods in the package. An R API to chromatoplot enables R programmers to script alternate data preprocessing tasks. GGobi is leveraged to provide interactive graphics via the rggobi package. The GUI, like that of exploRase, will be based on RGtk2.

The appendix contains examples which demonstrate the application of the methods and software to three biological datasets. Each of these is data collected on *Arabidopsis* plants, studying different functionality.

### 3 Scope

The work described in this thesis was conducted in collaboration with different people, or built closely upon existing research. Here is a description of my original contributions.

**RGtk2** The most significant software interface project is RGtk2. Although conceptually based on the original RGtk by Duncan Temple Lang, RGtk2 is a virtually complete rewrite of its predecessor. I am completely responsible for the development and maintenance of RGtk2, which has been available on CRAN for several years, and the paper is under revision to be published in the Journal of Statistical Software. In addition, I developed the cairoDevice entirely by myself, for embedding plots into RGtk2-based GUIs, and this is also available on CRAN.

**GGobi** The main contribution of this thesis to GGobi is the redesign of its core data pipeline. The overall design of the pipeline is the result of collaboration with Hadley Wickham and the rest of the GGobi team. I am primarily responsible for the implementation of the basic pipeline infrastructure, as well as several pipeline stages. Other contributions focus on increasing the availability of GGobi on popular platforms like Windows and Mac OS X and enhancing the usability of the GUI.

**rggobi** The rggobi package is largely a redesign of the original Rggobi package, created by Duncan Temple Lang. The R side of the interface is designed and implemented by Hadley Wickham, while I maintain the underlying C functions that interface with GGobi. This version rggobi is stable and available on CRAN. The low-level interface from R to GGobi is almost completely my work and is based on RGtk2. The implementation of this design is not yet fully realized, though an experimental version is available.

**rsbml** The rsbml package is purely the result of my efforts. The package is complete and available on CRAN and Bioconductor. The paper will be submitted to Bioinformatics as an application note.

**rcola** I have developed the design of rcola independently. The implementation is functional but not yet ready for public use.

**biocola** I have developed the design of biocola independently. The implementation is experimental and not yet ready for public use.

**exploRase** ExploRase has been developed in collaboration with biologists. In particular, biologists Eve Wurtele, Heather Babka and Suh-yeon Choi have contributed insights from the biological perspective that have led to improvements in usability and additional features that address the needs of biologists. Eun-kyung Lee and Dianne Cook have provided input from the statistics perspective. The design and implementation of the software, including the GUI, is largely my work. The application is a rewrite of the GeneGobi software created by Eun-kyung Lee. The most significant changes have been improved usability through an overhaul of the GUI and the addition of data subsetting features and statistical methods, including linear models. While it is a stable package and available on Bioconductor, the network features of exploRase are experimental. The paper is submitted to the Journal of Statistical Software, and a short communication will be submitted to Bioinformatics.

**chromatoplot**s The chromatoplot package implements a pipeline of numerical preprocessing methods and graphical diagnostics developed in collaboration with statisticians Heike Hofmann and Dianne Cook, as well as biologist Suh-yeon Choi. I have played the leading role in the development of the statistical methods and am solely responsible for the software implementation. The methods implemented in chromatoplot are relatively solid, but the work on the GUI and the diagnostic visualizations is still in progress. The current technical report will be made available through the Statistics department preprint series, and a shorter revised version will be submitted to a bioinformatics journal.

Each of the components of the research described in this thesis addresses different aspects of biological data analysis by providing new methods, frameworks, realizations and communication of the work through open software.

## R GTK2: A GRAPHICAL USER INTERFACE TOOLKIT FOR R

A paper published in Journal of Statistical Software

Michael Lawrence and Duncan Temple Lang

### Abstract

Graphical User Interfaces (GUIs) are growing in popularity as a complement or alternative to the traditional Command Line Interfaces (CLIs) to R. RGtk2 is an R package for creating GUIs in R. The package provides programmatic access to GTK+ 2.0, an open-source GUI toolkit written in C. To construct a GUI, the R programmer calls RGtk2 functions that map to functions in the underlying GTK+ library. This paper introduces the basic concepts underlying GTK+ and explains how to use RGtk2 to construct GUIs from R. The tutorial is based on simple and practical programming examples. We also provide more complex examples illustrating the advanced features of the package. The design of the RGtk2 API and the low-level interface from R to GTK+ are discussed at length. We compare RGtk2 to alternative GUI toolkits for R. The package is available from CRAN.

### 1 Introduction

An interface, in the most general sense, is the boundary across which two entities communicate. In most cases, the communication is bidirectional, involving input and output from both of the interfaced entities. In computing, there are two general types of interfaces: machine interfaces and user interfaces [Unwin and Hofmann, 1999]. A machine interface does not involve humans, while a user interface is between a human and a machine. In this paper, the

machines of interest are software, and the central software component is the R platform and language for statistical computing [R Development Core Team, 2005].

Two common types of user interfaces in statistical computing are the Command Line Interface (CLI) and the Graphical User Interface (GUI). The usual CLI consists of a textual console where the user types a sequence of commands at a prompt. The R console is an example of a CLI. A GUI is the primary means of interacting with desktops, like Windows and Mac OS, and statistical software like JMP [SAS Institute, 2007]. These interfaces are based on the WIMP (Window, Icon, Menu and Pointer) paradigm [Penners, 2007]. WIMP was developed at Xerox PARC in the 1970's and was popularized by the Apple Macintosh. On a WIMP desktop, application GUIs are contained within windows, and resources, such as documents, are represented by graphical icons. User controls are packed into hierarchical drop-down menus, buttons, sliders, etc. The user manipulates the windows, icons and menus with a pointer device, such as a mouse. The windows, icons, and menus, as well as other graphical controls such as buttons, sliders and text fields, have come to be known as *widgets*. The graphical event-driven, non-procedural nature and overall complexity of widgets makes their implementation a non-trivial task. To alleviate the burden on the application programmer, reusable widgets are collected into *widget toolkits*.

There is often debate over the relative merits of a CLI and a GUI lacking a console. The comparison largely depends on the skills and needs of the user [Unwin and Hofmann, 1999]. Effective use of a CLI requires the user to be proficient in the command language understood by the interface. For example, with a CLI, R users need to understand the R language. Learning a computer language often demands a significant commitment of time and energy; however, given a small amount of knowledge, one can use the language to perform arbitrary, rich tasks. A graphical interface is much less general and restrictive, but typically makes performing a specific task easier. It does this two different ways: a) stream-lining the steps involved in the task by providing a constrained context, and b) removing the need to remember function names and syntax. Different users benefit from the two different interfaces for different tasks. And there is little doubt that for occasional users of a language and for users focused a specific task,

a well-designed GUI is easier to learn and more accessible than a general purpose programming language.

Considering the widespread use and popular appeal of the R platform and the rich set of state-of-the-art statistical methodology it provides, it is desirable to try to make these available to a broader set of users by simplifying the knowledge needed to use such methods. The CLI has always been the most popular interface to R as it is the generic interface provided on all platforms and there has been much less focus in the R community on providing graphical interfaces for specific tasks. On some platforms, a CLI is a component of a larger GUI with menus containing various utilities for working with R. Examples of CLI-based R GUIs include the official Windows and Mac OS X GUIs, as well as the cross-platform Java GUI for R (JGR) [Helbig et al., 2004]. Although these interfaces are GUIs, they are still very much in essence CLIs, in that the primary mode of interacting with R is the same. Thus, these GUIs appeal mostly to the power users of R. A separate set of GUIs targets the second group of users, those learning the R language. Since this group includes many students, these GUIs are often designed to teach general statistical concepts in addition to R. A CLI component is usually present in the interface, though it is deemphasized by the surrounding GUI, which is analogous to a set of “training wheels” on a bicycle. Examples of these GUIs include Poor Man’s GUI (pmg) [Verzani, 2007b] and R Commander [Fox, 2007]. The third group of users, those who only require R for certain tasks and do not wish to learn the language, are targeted by task-specific GUIs. These interfaces usually do not contain a command line, as the limited scope of the task does not require it. If a task-specific GUI fits a task particularly well, it may even appeal to an experienced user. There are many examples of task-specific GUIs in R, including exploRase [Lawrence et al., 2006], limmaGUI [Smyth, 2005] and Rattle [Williams, 2006].

The task-specific GUIs, as well as more general R GUIs, are often implemented in the R language. The main advantage to writing a GUI in R is direct access to its statistical analysis functionality. The extensible nature of the R language and its support for rapid prototyping particularly facilitate the construction of task-specific GUIs. Building a GUI in R, as in any language, is made easier through the use of a widget toolkit. The tcltk package [Dalgaard, 2001,

2002], which provides access to tcl/tk [Ousterhout, 1994, Welch, 2003], is the most often used GUI toolkit for R. Others include RGtk [Temple Lang, 2004], based on GTK+ [Krause, 2007]; RwxWidgets [Temple Lang, 2007], based on wxWidgets [Smart et al., 2005]; and gWidgets [Verzani, 2007a], a simplified, common interface to several toolkits, including GTK+, tcl/tk and Java Swing. There are also packages for embedding R graphics in custom interfaces, such as gtkDevice [Drake et al., 2005] and cairoDevice [Lawrence and Drake, 2007] for GTK+ and tkplot [Tierney, 2007] for tcl/tk.

RGtk2 is a GUI toolkit for R derived from the RGtk package. Like RGtk, RGtk2 provides programmatic access to GTK+, a cross-platform (Windows, Mac, and Linux) widget toolkit. The letters *GTK* stand for the *GIMP ToolKit*, with the word *GIMP* recording the origin of the library as part of the GNU Image Manipulation Program. GTK+ is written in C, which facilitates access from languages like R that are also implemented in C. It is licensed under the *Lesser GNU Public License* (LGPL). GTK+ provides the same widgets on every platform, though it can be customized to emulate platform-specific look and feel. The original RGtk is bound to the previous generation of GTK+, version 1.2. RGtk2 is based on GTK+ 2.0, the current generation. Henceforth, this paper will only refer to RGtk2, although many of the fundamental features of RGtk2 are inherited from RGtk.

We continue with the fundamentals of the GTK+ GUI and the RGtk2 package. This is followed by a tutorial, including examples, on using RGtk2 to construct basic to intermediate GUIs. The paper then moves into a more technical domain, introducing the advanced features of the interface, including the creation of new types of widgets. We then present a technical description of the design and generation of the interface, which is followed by a discussion of more general binding issues. Next, we compare RGtk2 to existing GUI toolkits in R. We conclude by mentioning some applications of RGtk2 and explore directions for future development.

## 2 Fundamentals

This section begins with an introduction to the basic widgets and elements of the of the GTK+ library. We then turn our attention to the RGtk2 interface to GTK+, explaining how to create and manipulate widgets and how to respond to user input. The section concludes by introducing widget layout, the process of determining the size and position of each widget on the screen.

### 2.1 GTK+ Widgets

#### 2.1.1 The Widget Type Hierarchy

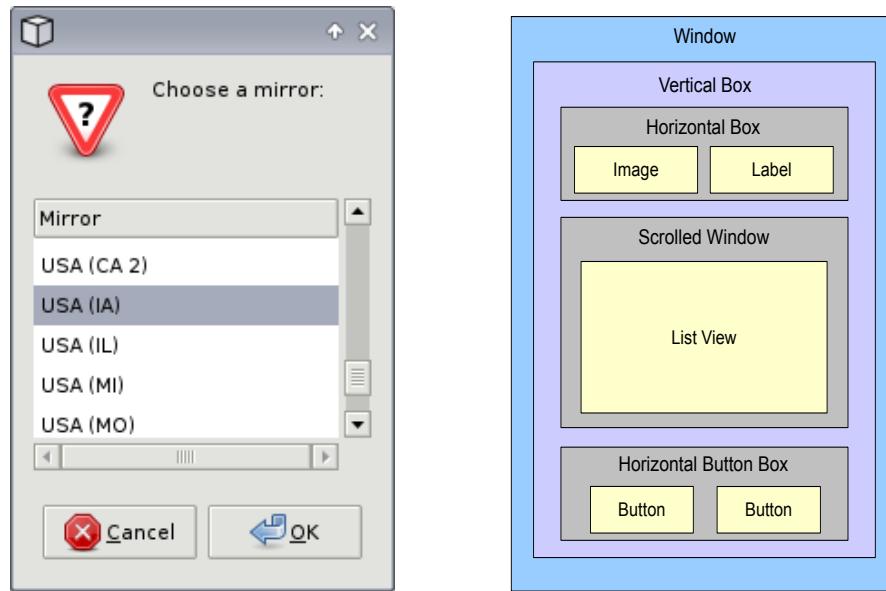


Figure 1 A dialog for selecting a CRAN mirror constructed using the RGtk2 package. The screenshot of the dialog is shown on the left. The user selects a mirror from the list and clicks the “OK” button to confirm the choice. In the image on the right, each rectangle corresponds to a widget in the GUI. The window is at the top-level, and each of the other widgets is geometrically contained within its parent. Many of the container widgets are invisible in the screenshot.

The left panel of figure 1 shows a GTK+ GUI that allows the user to select a CRAN

mirror for downloading R packages. This GUI is likely familiar to many R users, since a similar interface is present in the official Windows and Mac OS X R GUIs, among others. There are several different types of widgets in the CRAN mirrors GUI. A text label instructs the user to choose a mirror. A list control/widget contains the names of the available mirrors, and there are buttons for confirming or canceling the choice. The interface is enclosed by another type of widget, a window.

All of these widget types have functionality in common. For example, they are all drawn on the screen in a consistent style. To formalize this relationship and to simplify implementation by sharing code between widgets, GTK+ defines an inheritance hierarchy for its widget types, or classes. A small portion of the GTK+ class hierarchy is shown in Figure 2. For specifying the hierarchy, GTK+ relies on GObject, a C library that implements a class-based, single-inheritance object-oriented system. Each type of GTK+ widget is a GObject class that inherits from the base *GtkWidget* class which provides the general characteristics shared by all widget classes, e.g. properties giving the location, color; methods for hiding, showing and painting the widget. A GObject class encapsulates behaviors that all instances of the class share. Each class has a single parent from which it inherits the behaviors of its ancestors. A class can override any of its inherited behaviors. A more detailed and technical explanation of GObject is available in Section 5.2.

### 2.1.2 The Widget Tree

There is another tree hierarchy that is orthogonal to the class inheritance hierarchy. This hierarchy involves widget instances rather than widget classes. Each widget instance has a single parent instance in which it is contained, except for a top-level window which has no parent and serves as the root of the tree. Child widgets are contained within the rectangular region of their parents. In Figure 1, for example, the label, list of mirrors, and buttons are all contained within the top-level window, meaning that the window is the common ancestor of the other widgets. The right panel of figure 1 shows, in a simplified way, the two dimensional nesting of the widgets in the mirror selection example. Widgets that can contain other widgets

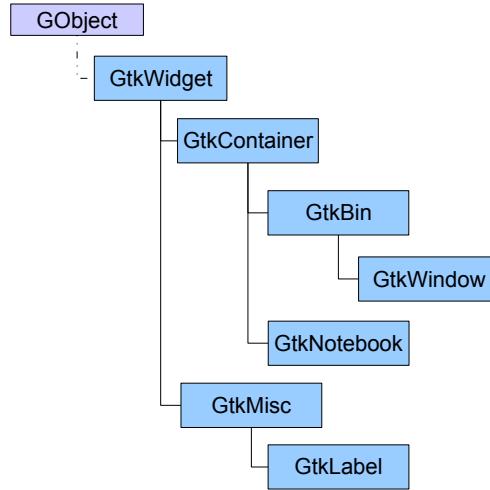


Figure 2 A small portion of the GTK+ class hierarchy. All widgets are derived from the *GtkWidget* class, which is derived, indirectly, from the *GObject* base class.

are called *containers* and their classes are derived from the *GtkContainer* class. Windows and tabbed notebooks are examples of containers. Combining primitive widgets like labels and icons within containers leads to more complex displays, such as menus, tool bars and even buttons which contain labels to display the text. A container is responsible for allocating its space to its children. This process is called layout management and is described in Section 2.3.

## 2.2 GTK+ Widgets in R

RGtk2 provides an Application Programming Interface (API) to the GTK+ library. A programmer uses an API to create an application based on functions implemented within a separate module. It is a contract that specifies in detail the functionality available to a programmer without specifying how that functionality is implemented.

As with other user interfaces, an API should be consistent and efficient to use. As an R package, RGtk2 primarily aims to be consistent with R conventions. This means hiding aspects of the GTK+ API that are foreign to R, such as explicit memory management. A secondary



Figure 3 “Hello World” in GTK+. A window containing a single button displaying a label with the text “Hello World”.

concern is consistency with the underlying GTK+ API. The developers of GTK+ have invested a significant amount of thought into its design. Thus, RGtk2 endeavors to interface R to the virtual entirety of GTK+, without leaving any gaps that may be unanticipated by the user. The only omissions are those that would violate consistency with R. For example, functions related to explicit memory management were excluded, as memory in R is managed by a garbage collector. Array length parameters are also excluded, as the length of a vector is always known in R. The RGtk2 API has also been designed for ease/efficiency of use. Towards this end, it specifies a default value for a function parameter whenever sensible and uses a special object-oriented syntax, as introduced by the SJava package [Temple Lang, 2006b].

To demonstrate the basic syntax and features of the RGtk2 API, we will construct a simple “Hello World” GUI, shown in Figure 3.

We will gradually progress from this trivial GUI to the aforementioned CRAN mirrors GUI and beyond. The first step is to create a top-level window to contain our GUI. Creating an instance of a GTK+ widget requires calling a single R function with its name matching the name of the class with the first character in lowercase. The following statement constructs an instance of the *GtkWindow* class.

```
window <- gtkWindow("toplevel", show = FALSE)
```

The first argument to the constructor for *GtkWindow* corresponds to the type of the top-level window. The set of possible window types is specified by what in C is known as an *enumeration*. Since enumerations are foreign to R, RGtk2 accepts string representations of enumeration values, like “toplevel”. For every GTK+ enumeration, RGtk2 provides an R vector that maps the nicknames to the underlying numeric values. In the above case, the vector is named *GtkWindowType*. The expression *names(GtkWindowType)* returns the names of the possible values of the *GtkWindowType* enumeration, and the same applies to all other enumerations. It is rarely necessary to explicitly use the enumeration vectors; specifying the nickname will work in most cases, including all method invocations and is preferable as it is easier for human readers to comprehend.

The *show* argument is the last argument for every widget constructor. It indicates whether the widget should be made visible immediately after construction. The default value of *show* is *TRUE*. In this case we want to defer showing the window until after we finish constructing our simple GUI.

The next steps are to create a “Hello World” button and to place the button in the window that we have already created. This depends on an understanding of how one programmatically manipulates widgets. Each widget class defines an API consisting of methods, properties, fields and signals. Methods are functions that take an instance of their class as the first argument and are used to instruct the widget to perform an action. Properties and fields store the public state of a widget. Examples of properties include the title of a window, the label on a button, and whether a widget has the keyboard focus. Signals are emitted as a result of events, such as user interaction with a widget. By attaching an R handler function to a widget’s signal, we can perform an action in response to all user inputs that generate that signal. We explain how one can interface R functions with each of these in the following sections as we continue with our “Hello World” example.

### 2.2.1 Invoking Methods

Methods are functions that operate on widgets inheriting from a particular class. The RGtk2 function for each GTK+ method is named according to the *classNameMethodName* pattern. For example, to add a child to a container, we need to invoke the *add* method on the *GtkContainer* class. The corresponding function name would be *gtkContainerAdd*. However, this introduces an inefficiency in that the user needs to remember the class to which a method belongs. To circumvent this problem, we introduce a syntax that is similar to that found in various object-oriented languages. The widget variable is given first, followed by the \$ operator, then the method name and its arguments. This syntax for calling *gtkContainerAdd* is demonstrated below as we add a button with the label “Hello World” to our window. The third statement calls *gtkWindowSetDefaultSize* to specify our desired size for the window when it is first shown. Each method belongs to a separate class, but the syntax frees the user from the need to remember the exact classes and also saves some typing as the \$ operator finds the most specific/appropriate method based on the class inheritance of the widget.

```
button <- gtkButton("Hello World")
window$add(button)
window$setDefaultSize(200,200)
```

Note that we use the lower case form of the first letter when using the \$ syntax, but the upper case form in the *classNameMethodName* function name. The \$ acts as a word separator and we use lower case at the beginning of new words.

### 2.2.2 Accessing Properties and Fields

Properties are self-describing elements that store the state of an aspect of a widget. Examples of properties include the title of a window, whether a checkbox is checked, and the length in characters of a text entry box. The R subset function / may be used to get the value of a widget property by name. Below we access the value of the *visible* property of our window. We find that the value is *FALSE*, since we specified it not to be shown at construction and have not made it visible since then.

```
> window["visible"]
[1] FALSE
```

Gtk+ properties may be set, given that they are writable, using the regular R assignment operator (*i-* or *=*). This is actually implemented via the */i-* method for Gtk+ widgets in RGtk2. The example below makes the window created above visible, using both property-setting methods, the second corresponding to a call to *gtkWidgetShow*, which is more conventional:

```
window["visible"] <- TRUE # or
window$show()           # the conventional way
```

For convenience, one might desire to set multiple properties with a single statement. This is possible using the *gObjectSet* method, which behaves similarly to the R *options* function, in that the argument name indicates the property to set to the argument value. In the single statement below, we set the window icon to the RGtk logo image and set the title to “Hello World 1.0”. The *imagefile* function retrieves an image from the RGtk2 installation. *gdkPixbuf* returns a list, where the first element is a *GdkPixbuf*, an image object, and the second is a description of an error encountered when reading the file or *NULL* if the operation was successful. Here we assume that there is no error.

```
image <- gdkPixbuf(filename=imagefile("rgtk-logo.gif"))[[1]]
window$set(icon = image, title = "Hello World 1.0")
```

In rare cases, it is necessary to access a field in the widget data structure. Fields are different from properties in several ways. Most importantly, it is never possible to set the value of a field. The user can retrieve the value of a field using the *//* function. For example, now that our window has been shown, it has been allocated a rectangle on the screen. This is stored in the *allocation* field of *GtkWidget*. It returns a list representing a *GtkAllocation* with elements *x*, *y*, *width* and *height*.

```
> window[["allocation"]]
$x
```

```
[1] 0

$y
[1] 0

$width
[1] 200

$height
[1] 200

attr("class")
[1] "GtkAllocation"
```

### 2.2.3 Handling Signals/Events

Once a GUI is displayed on the screen, the user is generally free to interact with it. Examples of user actions include clicking on buttons, dragging a slider and typing text into an entry box. In the CRAN mirrors example, possible user actions include selecting a mirror in the list, clicking the “OK” or “Cancel” buttons and pressing a keyboard shortcut, such as **Alt-O** for “OK”. An application may wish to respond in a certain way to one or more of such actions. The CRAN mirrors application, for example, should respond to an “OK” response by saving the chosen mirror in the session options.

So far, we have created and manipulated widgets by calling a list of procedures in a fixed order. This is convenient as long as the application is ignoring the user. Listening to the user would require a loop which continuously checks for user input. It is not desirable to implement such a loop for every application, so GTK+ provides one for all GUI applications to use within the same R session. When an application initializes the GTK+ event processing loop, there is an *inversion of control*. The application no longer has primary control of its flow; instead,

GTK+ asynchronously informs the application of events through the invocation of functions provided by the application to handle a specific type of event. These handlers are known as *callbacks*, because GTK+ is calling back into the application.

GTK+ widgets represent event types as signals. One or more callbacks can be connected to a signal for each widget instance. When the event corresponding to the signal occurs, the signal is emitted and the callbacks are executed in an order depending on how they were connected. In order to execute R code in response to a user action on a widget, we connect an R function to the appropriate signal on the widget. The *gSignalConnect* function performs this connection. The following code will make our “Hello World” example from above more interactive. The call to *gSignalConnect* will cause “Hello World!” to be printed upon emission of the “clicked” signal from the button in our window. The “clicked” signal is emitted when the user clicks the button with a pointer device or activates the button with a keyboard shortcut.

```
gSignalConnect(button, "clicked",
              function(widget) print("Hello world!"))
```

#### 2.2.4 Documentation

The RGtk2 documentation is available using the conventional R *help* command. It is derived from the documentation of GTK+ itself. To see the methods, properties, fields, and signals available for a particular class, the user should access the help topic matching the class name. For example, to read the documentation on *GtkWindow* we enter:

```
help(GtkWindow)
```

Similarly, the detailed help for a specific method is stored under the full name of the function. For example, to learn about the *add* method on *GtkContainer*, we enter:

```
help(gtkContainerAdd)
```

### 2.3 Widget Layout

In our “Hello World” example, we added only a single widget, a button, to the top-level window. In contrast, the CRAN mirrors window contains multiple widgets, which introduces

the problem of appropriately allocating the space in a window to each of its descendants in the container hierarchy. This problem is often called *layout management*. Laying out a GUI requires specifying the position and size of each widget below the top-level window. The simplest type of layout management is static; the position and size of each widget are fixed to specific values. This is possible with GTK+, but it often yields undesirable results. A GUI is interactive and changes in response to user input. The quality of a fixed layout tends to decrease with certain events, such as the user resizing the window, a widget changing its size requirement, or the application adding or removing widgets. For this reason, most layout management is dynamic.

In GTK+, containers are responsible for the layout of their children. The right panel in figure 1 shows how the nesting of layout containers results in the CRAN mirrors GUI shown in Figure 1. The example employs several important types of GTK+ layout containers.

First, there is the top-level *GtkWindow* that is derived from *GtkBin*, which in turn derives from *GtkContainer*. A *GtkBin* holds only a single child, and *GtkWindow* simply fills all of its allocated space with its child.

The most commonly used container for holding multiple children is the general *GtkBox* class, which stacks its children in a specified order and in a single direction, vertical or horizontal. The children of a *GtkBox* always fill the space allocated to the box in the direction orthogonal to that of the stacking, e.g. fill the available width when stacked vertically on top of each other.

The *GtkBox* class is abstract (or virtual), meaning that one cannot create instances of it. Instead, we instantiate one of its non-abstract subclasses. For example, in the CRAN mirror GUI, a vertical box, *GtkVBox*, stacks the label above the list, and a horizontal button box, *GtkHButtonBox*, arranges the two buttons. *GtkVBox* and its horizontal analog *GtkHBox* are general layout containers, while the button boxes *GtkVButtonBox* and *GtkHButtonBox* offer facilities specific to the layout of sets of buttons.

Here we will explain and demonstrate the use of *GtkHBox*, the general horizontal box layout container. *GtkVBox* can be used exactly the same way; only the direction of stacking

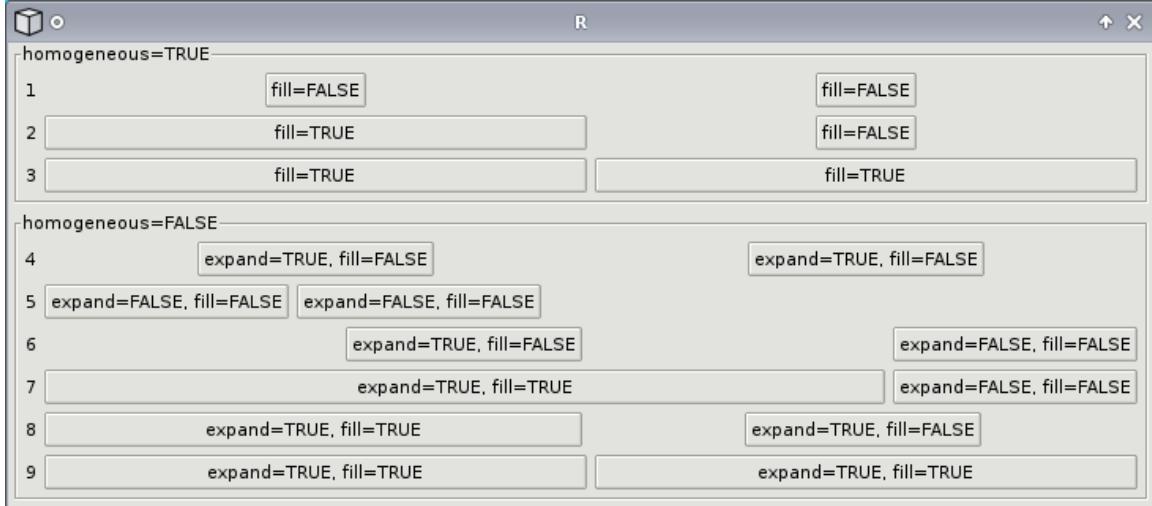


Figure 4 A screenshot demonstrating the effect of packing two buttons into *GtkHBox* instances using the *gtkBoxPackStart* method with different combinations of the *expand* and *fill* settings. The effect of the *homogeneous* spacing setting on the *GtkHBox* is also shown.

is different. Figure 4 illustrates a sampling of the possible layouts that are possible with a *GtkHBox*.

The code for some of these layouts is presented here. We begin by creating a *GtkHBox* widget. We pass *TRUE* for the first parameter, *homogeneous*. This means that the horizontal allocation of the box will be evenly distributed between the children. The second parameter directs the box to leave 5 pixels of space between each child.

```
box <- gtkHBox(TRUE, 5)
```

The equal distribution of available space is strictly enforced; the minimum size requirement of a homogeneous box is set such that the box always satisfies this assertion, as well as the minimum size requirements of its children.

The *gtkBoxPackStart* and *gtkBoxPackEnd* methods pack a widget into a box with left and right justification (top and bottom for a *GtkVBox*), respectively. For this explanation, we restrict ourselves to *gtkBoxPackStart*, since *gtkBoxPackEnd* works the same except for the justification. Below, we pack two buttons, *button\_a* and *button\_b* using left justification. First,

*button\_a* is packed against the left side of the box, and then we pack *button\_b* against the right side of *button\_a*. The space distribution is homogeneous, but the extra space for each widget is not filled. This results in the first row in Figure 4.

```
button_a <- gtkButton("Button A")
button_b <- gtkButton("Button B")
box$packStart(button_a, fill = FALSE)
box$packStart(button_b, fill = FALSE)
```

Making the space available to a child does not mean that the child will fill it. That depends on the minimum size requirement of the child, as well as the value of the *fill* parameter passed to *gtkBoxPackStart*. When a widget is packed with the *fill* parameter set to *TRUE*, the widget is sized to consume the available space. This results in rows 2 and 3 in Figure 4.

In many cases, it is desirable to give children unequal amounts of available space, as in rows 4-9 in Figure 4. This is evident in the CRAN mirrors dialog, where the mirror list is given more space than the “Please choose a mirror:” label. To create an inhomogeneously spaced *GtkHBox*, we pass *FALSE* as the first argument to the constructor, as in the following code:

```
box <- gtkHBox(FALSE, 5)
```

An inhomogeneous layout is freed of the restriction that all widgets must be given the same amount of available space; it only needs to ensure that each child has enough space to meet its minimum size requirement. After satisfying this constraint, a box is often left with extra space. The programmer may control the distribution of this extra space through the *expand* parameter to *gtkBoxPackStart*. When a widget is packed with *expand* set to *TRUE*, we will call the widget an “expanding” widget. All expanding widgets in a box are given an equal portion of the entirety of the extra space. If no widgets in a box are expanding, as in row 5 of Figure 4, the extra space is left undistributed. It is common to mix expanding and non-expanding widgets in the same box. For example, in the CRAN mirrors dialog, the box first ensures that the mirror list and the label above it are given enough space to satisfy their minimum requirement. Then, since the mirror list is expanding, all of the extra space is made

available to it, while the label is left only with its minimum requirement (i.e. enough space to show its text). Another example is given below, where *button\_a* is expanding, while *button\_b* is not.

```
box$packStart(button_a, expand = TRUE, fill = FALSE)
box$packStart(button_b, expand = FALSE, fill = FALSE)
```

The result is shown in row 6 of Figure 4. The figure contains several other permutations of the *homogeneous*, *expand* and *fill* settings.

GTK+ contains many types of layout containers besides boxes, including a grid layout (*GtkTable*), a user-adjustable split pane (*GtkHPaned* and *GtkVPaned*), and a tabbed notebook (*GtkNotebook*). More types of layout containers will be demonstrated later in the tutorial.

### 3 Basic GUI Construction

Thus far, we have reviewed the fundamentals of GTK+, working with GTK+ widgets from R, and widget layout management. In this section, we will build on this foundation to create some basic but potentially useful GUIs.

Constructing a GUI may be conceptually divided into two basic steps. First, one must create the individual widgets, specify their properties, and organize them into containers. This defines the physical aspect of the GUI: the appearance of the widgets and their spatial organization. The second step defines the behavior or the logical aspect of the interface. It involves registering handlers for signals that are emitted by the widgets, for example in response to a user pressing a button. The signal handlers encapsulate the logic beneath the interface. In this section, we will demonstrate these two steps and show how their integration results in functional GUIs.

#### 3.1 A Dialog with the User

A user interface is the conduit for a conversation between the machine and the user. This conversation may be broken down into a series of exchanges called *dialogs*. An application often needs to make a specific request for user input, such as the desired CRAN mirror. This

type of dialog is initiated by the machine posing a question to the user. The machine then waits for the user to respond. Usually, the application is unable to continue until receiving the user response, so the rest of the GUI is blocked until the dialog is concluded. This is called a *modal* dialog. A dialog is described as *non-modal* when the user can continue to perform other tasks even when the dialog is displayed.

GTK+ explicitly supports modal and non-modal requests for user input with a dialog widget, a top-level window that emits the *response* signal when the user has responded to the query. All dialogs in GTK+ are derived from the *GtkDialog* class. The CRAN mirrors GUI is an instance of *GtkDialog*. In the simpler example below, we will create a dialog that asks whether the user wants to upgrade the RGtk2 package installed on the system. Although we could build such a dialog using *GtkDialog* directly, *GtkMessageDialog*, an extension of *GtkDialog*, reduces the amount of necessary code for queries that can be expressed with a textual message and a set of buttons for the response. The dialog is constructed with a single function call:

```
main_application_window <- NULL # for purposes of this example
dialog <- gtkMessageDialog(main_application_window,
  "destroy-with-parent",
  "question", "yes-no", "Do you want to upgrade RGtk2?")
```

In the above invocation, the first parameter of the call to *gtkMessageDialog* indicates the parent window for the dialog. It is assumed that the main window of the application is stored as *main\_application\_window*. The second parameter indicates that the dialog should be destroyed when its parent, the main window, is destroyed. The next parameter indicates that this is a “question” dialog, which causes the dialog to display a question mark icon to the left of the text. The predefined set of buttons, in this case consisting of “Yes” and “No”, is specified by the next parameter. The final parameter specifies the text of the message. The resulting dialog is shown in Figure 5.

It is desirable for this dialog to be *modal*, meaning that user interaction is restricted to the dialog window until the user responds to the question. By invoking the *gtkDialogRun* function,



Figure 5 A screenshot of a message dialog requesting a “Yes” or “No” response from the user.

the dialog becomes modal and execution is blocked until the user gives a response, which is returned from the function. If the user answered “Yes”, our callback will install the latest version of the RGtk2 package. The call to *gtkWidgetDestroy* closes the dialog window and renders it unusable, i.e. if the object is used in subsequent computations, an error will be raised, because the dialog widget is no longer valid.

```
if (dialog$run() == GtkResponseType["yes"])
  install.packages("RGtk2")
dialog$destroy()
```

The reference to *GtkResponseType* above is one of the rare cases in which it is necessary to access an enumeration vector to retrieve the numeric value for a nickname. The reason for this is that *gtkDialogGetResponse* returns a plain numeric value to avoid an unnecessary restriction on the number of possible response types from a dialog. This allows programmers to introduce response types that do not exist within the *GtkResponseType* enumeration. In this case, it is known from the documentation of *GtkMessageDialog* that the value corresponding to the user clicking the “Yes” button will equal the “yes” value in *GtkResponseType*.

### 3.2 Giving the User More Options

Applications often need to ask questions for which a simple “Yes” or “No” answer does not suffice. As the number of possible responses to a query increases, enumerating every response with a button would place a burden on the user with a lengthy sequence of binary questions. It is easy to make a mistake when choosing one response from many and hard to go back to correct such errors. An interface should be forgiving and allow the user to confirm the choice before proceeding. This is how the CRAN mirrors dialog behaves: if the user accidentally chooses a mirror on the other side of the world, the user can correct the choice before clicking the “Okay” button and starting the installation process. This relates to the common need for a program to issue a set of queries to the user. Separating each query into its own dialog of buttons may unnecessarily force the user to answer the questions in a fixed, linear order and may not be very forgiving. It would also leave the user without a sense of context. If there were many actions and choices available to the user, a dialog-based interface would be tedious to use, requiring the user to click through dialog after dialog. Instead, a less assertive, non-linear interface is desired. In the examples below, we demonstrate widgets that present options in a passive way, meaning that there is usually no significant, immediate consequence to user interaction with the widget and the user has to conclude the interaction by clicking either the “Okay” or “Cancel” button.

The simplest user-level choice is binary and is usually represented in a passive way via a checkbox with a checked/unchecked or on/off state. In GTK+, the checkbox class is the *GtkCheckButton*. We may wish to extend our dialog confirming the upgrade of RGtk2 to include the option of also upgrading the underlying GTK+ C library. In the snippet below, we achieve this by adding a check button to the dialog. The area above the buttons in the *GtkDialog* is contained within a *GtkVBox*, which is stored as a property named *vbox* in the dialog object. Figure 6 shows our custom checkbox dialog and the following code is how to create it:

```
dialog <- gtkMessageDialog(main_application_window,
"destroy-with-parent",
```



Figure 6 A screenshot of a message dialog with a check box for requesting additional input on top of the original dialog in Figure 5.

```
"question", "yes-no", "Do you want to upgrade RGtk2?")  
check <- gtkCheckButton("Upgrade GTK+ system library")  
dialog[["vbox"]]\$add(check)
```

Let us now suppose that we would like to give the user the additional option of installing a development (experimental) version of GTK+. When an option has several choices, a check button is no longer adequate. A simple approach is to create a set of toggle buttons where only one button may be active at once. The buttons in this set are known as *radio buttons*, corresponding to the pre-programmed channel selection buttons on old-style radios. Below, we create a new dialog that asks the user to specify the version of GTK+ C libraries to install, if any. When each radio button is created, it needs to be given the existing collection of buttons already in the group. For creating the first button, *NULL* should be passed as the group. Each button is added to a vertical box.

```
dialog <- gtkMessageDialog(main_application_window,  
"destroy-with-parent",  
"question", "yes-no", "Do you want to upgrade RGtk2?")  
choices <- c("None", "Stable version", "Unstable version")  
radio_buttons <- NULL
```



Figure 7 A screenshot of a message dialog with a set of radio buttons on top of the base dialog shown in Figure 5.

```
vbox <- gtkVBox(FALSE, 0)
for (choice in choices) {
  button <- gtkRadioButton(radio_buttons, choice)
  vbox$add(button)
  radio_buttons <- c(radio_buttons, button)
}
```

A group of radio buttons are often graphically enclosed by a drawn border with a text label indicating the purpose of the buttons. This widget is a container called *GtkFrame* and is generally used for graphically grouping widgets that are logically related. The code below adds the box containing the radio buttons to a newly created frame. The final result is shown in Figure 7.

```
frame <- gtkFrame("Install GTK+ system library")
frame$add(vbox)
dialog[["vbox"]]$add(frame)
```

Now we would like to go a step further and allow the user to choose the exact version of

GTK+ to install, as RGtk2 is source compatible with any version from 2.8.0 onwards. As the number of options increases, however, radio buttons tend to consume too much space on the screen. In this case, a label displaying the current selection with a drop down menu allowing for selecting from a list of alternatives may be appropriate. This is known as a *GtkComboBox* in GTK+. The following snippet illustrates its use. Each call to *gtkComboBoxAppendText* adds a text item to the drop-down menu. The call to *gtkComboBoxSetActive* makes the first item (0 due to zero-based counting in C) the currently selected one. Figure 8 shows the result and below is the corresponding code:

```
dialog <- gtkMessageDialog(main_application_window,
  "destroy-with-parent",
  "question", "yes-no", "Do you want to upgrade RGtk2?")
choices <- c("None", "GTK+ 2.8.x", "GTK+ 2.10.x", "GTK+ 2.12.x")
combo <- gtkComboBoxNewText()
combo$show()
for (choice in choices)
  combo$appendText(choice)

combo$setActive(0) # select "None", the first choice.
frame <- gtkFrame("Install GTK+ system library")
frame$add(combo)
dialog[["vbox"]]$add(frame)
```

### 3.3 The CRAN Mirrors Dialog

Having demonstrated the creation some basic dialogs, we are now prepared to construct the CRAN mirror selection dialog, shown in Figure 1. Given the large number of CRAN mirrors, one strategy would be to borrow the combobox dialog created above; however, there may be a better alternative. Since there is no reasonable default CRAN mirror, the user always needs to pick a mirror. Packing the mirrors into a combo box would only force the user to make an

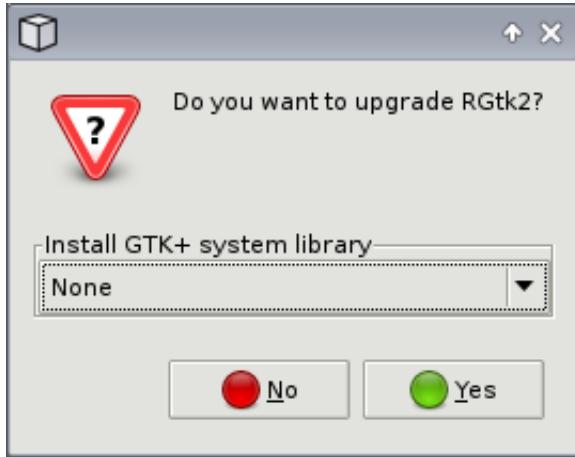


Figure 8 A screenshot of a message dialog with a combobox for selecting an option from a drop-down menu before responding to the dialog.

extra click. Instead, we want to display a reasonable number of CRAN mirrors immediately after the dialog is opened. It may not be possible to display every mirror at once on the screen, but, as seen in the screenshot, we can embed the list in a scrolled box, so that only one part of the list is visible at a given time.

We begin with the construction of the dialog window. For this dialog, we assume that there is no main application window (see Section 4) to serve as the parent. Instead, we pass *NULL* for the parent and 0 for the second argument rather than “*destroy-with-parent*”. We use the literal 0 here instead of a value name, because *GtkDialogFlags*, like all flag enumerations in GTK+, lacks a value for 0.

```
dialog <- gtkMessageDialog(NULL, 0, "question", "ok-cancel",
                           "Choose a mirror:", show = FALSE)
```

Next, we create a list for holding the mirror names using the *GtkTreeView* widget (so named because the rows in the list may be organized hierarchically, but we will not discuss this feature). The RGtk2 package provides a facility for creating a tabular data structure based on an R *data.frame*, called *RGtkDataFrame*. *RGtkDataFrame* is an extension of *GtkTreeModel*, which is the data structure viewed by *GtkTreeView*. Below, we create an *RGtkDataFrame* for

our list of CRAN mirrors and construct a *GtkTreeView* based on it.

```
mirrors <- read.csv(file.path(R.home("doc"), "CRAN_mirrors.csv"),
                     as.is = TRUE)

model <- rGtkDataFrame(mirrors)

view <- gtkTreeView(model)

view$getSelection()$setMode("browse")
```

The final line configures the *GtkTreeView* so that exactly one item is always selected in the view. This prevents the user from providing invalid input (i.e. a multiple or empty selection).

Initially, the tree view does not contain any columns. We need to create a *GtkTreeViewColumn* to list the mirror names, and we do so with the following code:

```
column <- gtkTreeViewColumn("Mirror", gtkCellRendererText(), text = 0)

view$appendColumn(column)
```

The first parameter to *gtkTreeViewColumn* specifies the title of the new column. Since we are displaying text (the names of the mirrors) in the column, we pass an instance of *GtkCellRendererText* which draws the values in our *data.frame* as text in the *GtkTreeView*. The parameter named *text* specifies that the first column of the *data.frame* contains the values to draw as text. Note that the index is zero-based (for historical reasons).

Given the large number of CRAN mirrors, the list would take up excessive space if not embedded into a scrolled window. *GtkScrolledWindow* is a container widget that provides a scrolled view of its child when the child requests more space than is available. We add the tree view to a *GtkScrolledWindow* instance that requests a minimum vertical size sufficient for showing several mirrors at once.

```
scrolled_window <- gtkScrolledWindow()

scrolled_window$setSizeRequest(-1, 150)

scrolled_window$add(view)
```

The size of *scrolled\_window* is set using *gtkWidgetGetSizeRequest*, which takes values in pixel units.

It only remains to add the scrolled window to the dialog, run the dialog, and set the selected CRAN mirror if the user confirms the selection. The selection of a tree view is stored in a separate *GtkTreeSelection* object retrieved by *gtkTreeViewGetSelection*. The *getSelectedRows* method returns a list containing the tree paths for the selected rows and the tree model. The list of tree paths is stored under the name *retval* as it is the actual return value from the C function. Finally, we retrieve the row index from the *GtkTreePath* for the first (and only) selected row and set its URL as the repository.

```
dialog[["vbox"]]$add(scrolled_window)

if (dialog$run() == GtkResponseType["ok"]) {

  selection <- view$getSelection()

  sel_paths <- selection$getSelectedRows()$retval

  sel_row <- sel_paths[[1]]$getIndices() [[1]]

  options(repos = mirrors[sel_row, "URL"])

}

dialog$destroy()
```

### 3.4 Embedded R Graphics

In a statistical graphical interface, it is often beneficial or necessary to display statistical graphics within the interface. As an example, we consider the contemporary problem of visualizing microarray data. The large number of genes leads to a significant amount of overplotting when, for example, plotting the expression levels from two chips in a scatterplot. One solution to the problem of overplotting is alpha blending. However, choosing the ideal alpha level may be time-consuming and tedious. Linking a slider widget to the alpha level of an R scatterplot may accelerate the search (See Figures 9 and 10).

As a preliminary step, we use a 2D mixture distribution of correlated variables to simulate expression values for two microarray chips.

```
n <- 5000

backbone <- rnorm(n)
```

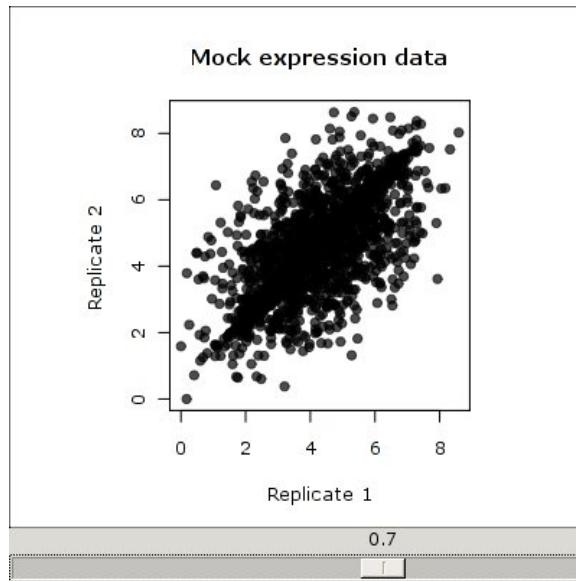


Figure 9 Scatterplot of two microarray replicates, with a slider widget underneath that controls the alpha level of the points. This screenshot shows the initial alpha of 0.7. This value does not lead to a clear display of the density at each location.

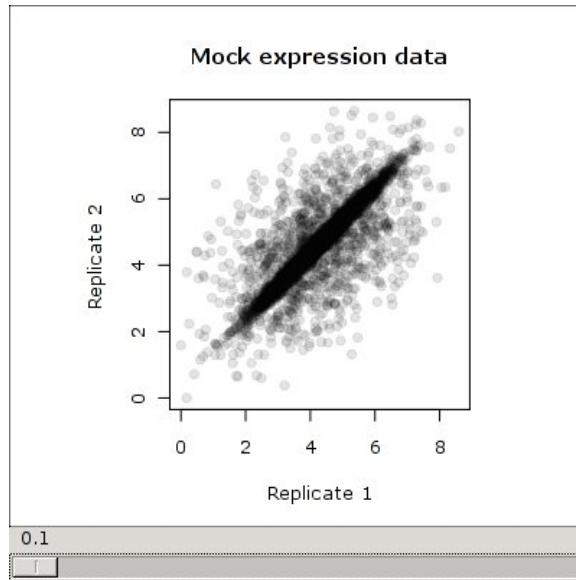


Figure 10 The same scatterplot from 9, except the alpha parameter has been set to 0.1.

```
ma_data <- cbind(backbone+c(rnorm(3*(n/4),sd=0.1), rt(n/4, 80)),
backbone+c(rnorm(3*(n/4),,0.1), rt(n/4, 80)))

ma_data <- apply(ma_data, 2, function(col) col - min(col))
```

The first step towards making our GUI is to create the window that will contain the graphics device and slider widgets.

```
win <- gtkWindow(show = FALSE)
```

One may embed R graphics within an RGtk2 GUI using the `cairoDevice` [Lawrence and Drake, 2007] or `gtkDevice` [Drake et al., 2005] packages. The `cairoDevice` package draws R graphics using Cairo [Cairo, 2007], a library for vector-based, antialiased graphics. When `cairoDevice` draws to the screen it is actually drawing to a GTK+ widget of type *GtkDrawingArea*. A *GtkDrawingArea* is an empty widget meant for drawing arbitrary graphics in an interface. Here we construct a drawing area in which the R graphics will be drawn:

```
graphics <- gtkDrawingArea()
```

Now that we have a widget for displaying R graphics, we need the slider that controls the alpha level. A slider is a widget, much like a scroll bar, for choosing a number at a certain precision from a certain range. Here, a horizontal slider, called *GtkHScale*, is created with a range from 0.1 to 1.0, with a step size of 0.1.

```
slider <- gtkHScale(min=0.1, max=1.00, step=0.1)
```

When the user moves the slider, the plot should be updated so that its alpha level reflects the slider value. This is achieved by connecting an R callback function to the “value-changed” signal of the slider. This callback function, *scale\_cb*, replots the microarray data, *ma\_data*, using an alpha level equal to the current value of the slider.

```
scale_cb <- function(range) { # 'range' is of type GtkRange
  par(pty = "s")
  plot(ma_data[,1], ma_data[,2],
       col = rgb(0, 0, 0, alpha = range$getValue()),
```

```

    xlab = "Replicate 1", ylab = "Replicate 2",
    main = "Mock expression data", pch = 19)

}

gSignalConnect(slider, "value-changed", scale_cb)

```

The next steps are to add the drawing area and the slider to the window and then to show the window on the screen. Although the window is a container, it inherits from *GtkBin*, meaning that it can hold only a single child widget. Thus, we will pack our widgets into a vertical stacking box container, *GtkVBox*, and add our box to the window. Here, we would like the graphics to take up all of the space not consumed by the slider, so the graphics device is packed to *expand* and *fill*, while the slider is not. (See section [2.3](#).)

```

vbox <- gtkVBox()

vbox$packStart(graphics, expand = TRUE, fill = TRUE, padding = 0)
vbox$packStart(slider, expand = FALSE, fill = FALSE, padding = 0)
win$add(vbox)

```

As a final step, we set the default size of the window and show it and all of its children.

```

win$setDefaultSize(400,400)
win$showAll()

```

Now that the window is visible on screen, we can instruct R to draw its graphics to the drawing area using the *asCairoDevice* function in the *cairoDevice* package:

```

require(cairoDevice)
asCairoDevice(graphics)

```

The call to *asCairoDevice* makes an R graphics device from this widget and makes this the currently active graphics device on which R graphics will be displayed.

Finally, the value of the slider is initialized to 0.7,

```
slider$setValue(0.7)
```

which in turn activates the callback, generating the initial plot. The initial state of the interface is shown in Figure 9. Figure 10 shows the plot after the user has moved the slider to set the value of alpha to 0.1.

## 4 A Sample Application

The interfaces presented thus far are each designed for a singular, focused task, such as choosing a CRAN mirror or viewing a scatterplot at different alpha levels. However, often an interface supports a larger collection of separate operations and the user is in control of initiating different tasks from the general interface. These interfaces for broader, more complex applications are typically based on what is called an *application window*, which often contains a menu bar, tool bar, application-specific area, and status bar in order from top to bottom. The menu bar and tool bar are widgets designed to facilitate the user selecting different *actions*, each of which represents an option or operation in the application. The status bar at the bottom commonly reports information about the activities or state of the application or information for the user as a text message and may be adjacent to a progress bar that displays the continuing progress of long running operations. This layout and design is a common convention which helps users navigate a new GUI.

The following example demonstrates how one might construct a reasonably complex application using RGtk2. We aim to build a viewer for one or more R *data.frames* that is capable of sorting and filtering the rows in each data frame. We also give it facilities to load and save a *data.frame* to and from a CSV file.

The resulting GUI is shown in Figure 11. Each data frame is displayed in a table, using a *GtkTreeView* widget. As we would like to support multiple spreadsheets at once, we embed each table in a tabbed notebook, *GtkNotebook*. Below each spreadsheet is a text entry (a *GtkEntry* widget), in which the user may enter an expression for filtering the table view. Below this is a status bar (a *GtkStatusbar* widget) that communicates the status of the application to the user, such as whether the loading of a dataset is complete. At the top are a menu bar (a *GtkMenubar* widget) and tool bar (a *GtkToolbar* widget) that allow the user to invoke various

actions, such as loading a new dataset or quitting the application.

## 4.1 Main window

We begin by creating the main window for the application and setting its default size, specified in number of screen pixels.

```
main_window <- gtkWindow(show = FALSE)  
main_window["title"] <- "RGtk2 Spreadsheet"  
main_window$setDefaultSize(600,600)
```

## 4.2 Menu bar and tool bar

### 4.2.1 Callbacks

Next, we implement the operations for the menu items and corresponding callbacks to load and save a data frame and to quit the “application”. Each of these functions is a callback which takes the widget associated with the action as its first argument and the top-level window as its second. The load and save operations leverage the *GtkFileChooserDialog* widget type, a dialog that contains a graphical file browser for specifying the path to a file. *GtkFileChooserDialog* has several modes corresponding to common file selection tasks. In this case, we use the “open” mode for the reading action and the “save” mode for the writing action. The “accept” response from the dialog indicates that the user has confirmed the file selection by clicking the “Open” or “Save” button.

**RGtk2 Spreadsheet**

File

Folder icon, Print icon, Save icon

rownames	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
Duster 360	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4
Merc 280C	17.8	6	167.6	123	3.92	3.44	18.9	1	0	4	4
Merc 450SE	16.4	8	275.8	180	3.07	4.07	17.4	0	0	3	3
Merc 450SL	17.3	8	275.8	180	3.07	3.73	17.6	0	0	3	3
Merc 450SLC	15.2	8	275.8	180	3.07	3.78	18	0	0	3	3

Filter expression: mpg > 20

mtcars chickwts

Dataset chickwts loaded.

Figure 11 Screenshot of a spreadsheet application constructed with RGtk2. The current sheet is from the *mtcars* dataset. The table is filtered by the expression `mpg > 20` and sorted by in decreasing order of the values of the *mpg* variable.

```

if (dialog$run() == GtkResponseType["accept"]) {
  df <- read.csv(dialog$getFilename())
  load_spreadsheet(df, basename(dialog$getFilename()))
}

dialog$destroy()

}

save_cb <- function(widget, window) {
  dialog <- gtkFileChooserDialog("Enter a name for the file", window,
                                "save",
                                "gtk-cancel",
                                GtkResponseType["cancel"],
                                "gtk-save",
                                GtkResponseType["accept"])
  if (dialog$run() == GtkResponseType["accept"])
    save_file(dialog$getFilename()) # not implemented
  dialog$destroy()
}

quit_cb <- function(widget, window)
  window$destroy() # quick and dirty

```

#### 4.2.2 User actions

We make these action callbacks available in both the menu bar and tool bar. We begin by creating the main window, defining the actions and bundling them into a *GtkActionGroup*. A *GtkActionGroup* is a container for *GtkAction* objects. The base *GtkAction* represents an operation that a user may request an application to perform. An action may be manifested in a GUI in multiple ways, such as an item in a menu or a button in a tool bar. The widgets are synchronized with the properties of the action. For example, if an action is disabled, the menu

items and tool bar buttons will also be disabled. Extensions of *GtkAction* exist for toggle and radio options, but those are not described here. For details, see *help(GtkToggleAction)* and *help(GtkRadioAction)*, respectively.

Each action is defined with a list, containing the action ID (for referring to the item later in the code), icon ID, label, keyboard shortcut, tooltip, and callback.

```
actions <- list(
  list("FileMenu", NULL, "_File"),
  list("Open", "gtk-open", "_Open File", "<control>O",
       "Select a CSV file to load as a spreadsheet", open_cb),
  list("Save", "gtk-save", "_Save", "<control>S",
       "Save the current spreadsheet to a CSV file", save_cb),
  list("Quit", "gtk-quit", "_Quit", "<control>Q",
       "Quit the application", quit_cb)
)
action_group <- gtkActionGroup("spreadsheetActions")
action_group$addActions(actions, main_window)
```

Specifying the action definitions as R lists is an example of high-level type conversion, where a native R structure is implicitly converted to a complex GTK+ object. In this case, an R list is being converted to a set of *GtkAction* objects. See Section 6.3.2 for a technical explanation and justification. The first action will serve as the basic menu container for the rest of the items and actions. Since it performs no function, it is not necessary to specify all of the fields, such as the action.

#### 4.2.3 Loading the actions into a *GtkUIManager*

We then create a *GtkUIManager* and provide it with our action group.

```
ui_manager <- gtkUIManager()
ui_manager$insertActionGroup(action_group, 0)
```

Next, we specify the layout of the menu bar and tool bar containing the actions defined above, by calling the *gtkUIManagerAddUi* method. Each piece of the user interface added to a *GtkUIManager* instance must be identified by a “merge id”. This allows removing (unmerging) the UI at a later time. The *path* parameter indicates where the UI element should be merged. Similar to a path in a file system or URL, each element name in the path is delimited by a forward slash (“/”). The *name* parameter identifies the element to the manager, and *action* is the ID of the action in the provided action group. If *action* is NULL, a separator widget is added. Finally, *type* indicates the type of UI element, such as a tool bar or menu bar. The default is “auto”, which asks the *GtkUIManager* to guess based on the path.

```
merge <- ui_manager$newMergeId()

ui_manager$addUi(merge_id = merge, path = "/", name = "menubar",
                 action = NULL, "menubar")

ui_manager$addUi(merge, "/menubar", "file", "FileMenu", "menu")
ui_manager$addUi(merge, "/menubar/file", "open", "Open", "menuitem")
ui_manager$addUi(merge, "/menubar/file", "save", "Save", "menuitem")
ui_manager$addUi(merge, "/menubar/file", "sep", NULL, "menuitem")
ui_manager$addUi(merge, "/menubar/file", "quit", "Quit", "menuitem")
ui_manager$addUi(merge, "/", "toolbar", NULL, "toolbar")
ui_manager$addUi(merge, "/toolbar", "open", "Open", "toolitem")
ui_manager$addUi(merge, "/toolbar", "save", "Save", "toolitem")
ui_manager$addUi(merge, "/toolbar", "quit", "Quit", "toolitem")
```

#### 4.2.4 Creating the widgets

The next step is to use the *GtkUIManager* to create the actual menu bar and tool bar widgets from the action definitions and layout.

```
menubar <- ui_manager$getWidget("/Menubar")
toolbar <- ui_manager$getWidget("/Toolbar")

# enable keyboard shortcuts
```

```
main_window$addAccelGroup(ui_manager$getAccelGroup())
```

### 4.3 Status bar

To report information from and about the application, we will use a *GtkStatusbar* widget. A status bar maintains a stack of text messages and displays the message on top of the stack. When a message is added to the status bar stack, it is immediately displayed, and when the message is removed, the previous one is displayed. Each message is associated with a *context*. A context ID may be created using the *gtkStatusbarGetContextId* function. A context ID is a number that is generated in a consistent way from any user-supplied string that serves as the human-readable context name. Here we create a status bar and push the message “Ready” onto the top of the stack within the context named “info”. Other contexts could be named, e.g., “warning” or “error”.

```
statusbar <- gtkStatusbar()
info <- statusbar$getContextId("info")
statusbar$push(info, "Ready")
```

In order to handle multiple spreadsheets simultaneously but display only one at a time, we will use a special type of container called *GtkNotebook*. This provides labels on the border of the notebook like a ring binder which the user can select to switch between the different widgets within the notebook. This is used in Excel to present several work sheets within a single window, and also within certain Web browsers to allow the user to view multiple Web pages within a single window. Below, we create the notebook and add it, along with the menu bar, tool bar and status bar, to the window, through a *GtkVBox*.

```
notebook <- gtkNotebook()
notebook$setTabPos("bottom") # tabs horizontally along the bottom
vbox <- gtkVBox(homogeneous = FALSE, spacing = 0)
vbox$packStart(menubar, expand = FALSE, fill = FALSE, padding = 0)
vbox$packStart(toolbar, FALSE, FALSE, 0)
```

```
vbox$packStart(notebook, TRUE, TRUE, 0)
vbox$packStart(statusbar, FALSE, FALSE, 0)
main_window$add(vbox)
main_window$show()
```

## 4.4 Spreadsheet

Next, we need to create the *GtkTreeView* that will display a given *data.frame* as a table. The data is passed to the *GtkTreeView* by attaching it to a *GtkTreeModel* data model.

### 4.4.1 Data model

The following function, *create\_tree\_model*, will create a *GtkTreeModel* object that obtains its data from an R *data.frame*, passed as an argument to the function.

```
create_tree_model <- function(df)

df <- cbind(rownames = rownames(df), df)

# add logical visibility column (all visible by default)
filter_df <- cbind(filter = TRUE, df)

model <- rGtkDataFrame(filter_df)

# create filter

filter_model <- gtkTreeModelFilterNew(model)

# show rows that have TRUE in first column

filter_model$setVisibleColumn(0)

# allow sorting of table

sort_model <- gtkTreeModelSort(filter_model)

sort_model

}
```

The function employs the *RGtkDataFrame* utility that allows the *GtkTreeView* to use an R *data.frame* as its data source. In order to support filtering and sorting of the displayed data, the *RGtkDataFrame* is proxied by a *GtkTreeModelFilter* model, which in turn is proxied by a

*GtkTreeModelSort* model. A proxy data model sits between a source data model and a client, such as a *GtkTreeView*. The data provided by a proxy model results from the modification of the data in the source model.

#### 4.4.2 Table view

The next function, *create\_tree\_view*, will create the *GtkTreeView* given the *GtkTreeModel* created by *create\_tree\_model* above. Each column of the *data.frame*, provided as the second argument, is displayed by a column in the tree view. We configure the tree view so that it shows grid lines (if the user has GTK+ 2.10.0 or higher) and supports sorting on a column when the user clicks on the column header.

```
create_tree_view <- function(model, df)
{
  tree_view <- gtkTreeView(model)
  sapply(seq_len(ncol(df)),
         function(j) {
           renderer <- gtkCellRendererText()
           column <- gtkTreeViewColumn(colnames(df)[j], renderer,
                                         text = j)
           column$setSortColumnId(j)
           column$setCellDataFunc(renderer,
                                 function(column, renderer, model, iter)
{
  i <- model$getPath(iter)$getIndices()[[1]] + 1
  renderer["text"] <- as.character(df[i,j])
})
tree_view$appendColumn(column)
}
)
tree_view$setHeadersClickable(TRUE) # sort by clicking column header
```

```

if (is.null(gtkCheckVersion(2,10,0))) # check GTK+ version >= 2.10.x
  tree_view$setGridLines("both")
}

```

The call to *setCellDataFunc* (above) attaches a callback that formats the text values as R does by default (GTK+ takes a simpler approach that gives each number 6 significant figures). Note that this callback is called each time a cell is rendered, so it could negatively impact performance, especially when scrolling. For large spreadsheets, we recommend using a dedicated spreadsheet application.

#### 4.4.3 Filter text entry

Next, we define a function that creates the text box for the user to enter a filter expression. This uses the *GtkEntry* widget. Whenever the *GtkEntry* is “activated,” e.g. by the user pressing the ENTER key, we update the filter by the result of the R expression.

```

create_entry <- function(model)
{
  entry <- gtkEntry() # for filter expression
  gSignalConnect(entry, "activate",
    function(entry)
      # update column used by filter
      # according to logical expression
      model[, "filter"] <- eval(parse(text=entry$text),
        df))
}

```

#### 4.5 Loading a spreadsheet

Finally, we define the function that uses the others to load a *data.frame* into the GUI. This function creates the necessary widgets and packs them into a notebook page. To limit its visible size, the data grid/table is added to a *GtkScrolledWindow*.

```

load_spreadsheet <- function(df, name)
{
  model <- create_tree_model(df)
  tree_view <- create_tree_view(model, df)
  entry <- create_entry(model)

  # pack widgets
  hbox <- gtkHBox(FALSE, 5)
  hbox$packStart(gtkLabel("Filter expression:"), FALSE, FALSE, 0)
  hbox$packStart(entry, TRUE, TRUE, 0)
  vbox <- gtkVBox(FALSE, 5)
  scrolled_window <- gtkScrolledWindow()
  scrolled_window$add(tree_view) # support scrolling for the table
  vbox$packStart(scrolled_window, TRUE, TRUE, 0)
  vbox$packStart(hbox, FALSE, FALSE, 0)

  # add page to notebook
  if (missing(name))
    name <- deparse(substitute(df))
  notebook$appendPage(vbox, gtkLabel(name))

  # update statusbar
  statusbar$push(info, paste("Dataset", name, "loaded."))
}

}

```

The function concludes by updating the status bar to indicate that the dataset has been successfully loaded.

An example of using the above function to add a spreadsheet is given below:

```
load_spreadsheet(mtcars)
```

This application is obviously missing many important features. For example, there is no easy way to return to the complete *data.frame* after subsetting, and it is not possible to edit the cells. The main purpose of the example is to introduce the process of building an application window.

## 5 Advanced Features

This section describes features of RGtk2 that are beyond the construction of basic and intermediate GUIs. It is meant for readers interested in advanced and specialized RGtk2 features such as the ability to extend GTK+ classes and interface with low-level and third-party libraries that are integrated with GTK+. Much of this functionality is applicable outside of GUI construction.

First, we describe the additional libraries (other than Gtk+) bound by RGtk2 that are meant to support the construction of advanced, graphically-intensive interfaces. The focus then shifts to the low-level support for the GObject object-oriented programming library. The RGtk2 user is able to manipulate objects in external GObject-based applications (i.e. top-level GUIs running within the same R session) that are bound to R by code outside of the RGtk2 package. RGtk2 also supports defining new GObject classes in R.

### 5.1 Additional Library Support

The GTK+ 2.0 library incorporates several other libraries: Cairo, GDK, GdkPixbuf, Pango and ATK and the RGtk2 package provides R-level bindings for each of these libraries, in addition to GTK+ itself.

**Cairo** Cairo is a 2D vector graphics library with which GTK+ widgets are drawn. It is possible to use Cairo directly to draw custom graphics within a *GtkDrawingArea*. The library is also useful outside of GUI construction, in that one can draw vector graphics to off-screen surfaces in common formats such as PNG, SVG, PS, and PDF files.

**GDK** The GIMP Drawing Kit, GDK, is the low-level hardware access and drawing layer for GTK+. It is most useful for raster-based (non-vector) drawing of graphical primitives like lines, rectangles and circles and for handling raw mouse and keyboard events. It also provides access to windowing system resources, such as screens in a multi-headed environment. Although the drawing functions of GDK overlap somewhat with Cairo, Cairo is for drawing vectors, while GDK is for direct drawing of pixels. Another reason for the redundancy is that GDK predates Cairo, and thus the GDK drawing routines are present for backwards compatibility.

**GdkPixbuf** GdkPixbuf is an image manipulation library based on GDK. Its features include rendering, scaling, and compositing of images. GdkPixbuf can read and write several image formats, including JPEG, PNG, and GIF. Like Cairo, GdkPixbuf could be used independently of a GUI for working with arbitrary graphics in R.

**Pango** Pango provides facilities for rendering and formatting text with rich capabilities for handling international characters. It also provides cross-platform access to the font configuration of a system. Pango is most often used directly for embedding text in graphics when drawing to a *GtkDrawingArea* or an off-screen destination, e.g. image.

**ATK** The Accessibility ToolKit (ATK) supports accessibility technologies to make GUIs amenable to users with “disabilities”. It allows accessibility devices to interact with GTK+ GUIs. ATK is not likely to be very useful from R. Its binding is included for the sake of completion, since ATK types are present in the GTK+ API.

**Libglade** Libglade constructs GTK+ GUIs from XML descriptions. The XML descriptions are output from Glade, which is a GUI tool for interactively designing other GUIs. As of GTK+ 2.12.0, which includes native support for constructing widgets from XML descriptions, Libglade is essentially obsolete. The bindings are still included for backwards compatibility.

## 5.2 A GObject Primer

GTK+, as well as the libraries described in the previous section, except for Cairo, are based on the GObject library for object-oriented programming in C. GObject forms the basis of many other open-source projects, including the GNOME [Krause, 2007] and XFCE [Fourdan, 2000] desktops and the GStreamer multimedia framework [Walthinsen, 2001].

RGtk2 interfaces with parts of GObject and permits the R programmer to create new GObject classes in R. Understanding this functionality depends on a familiarity with the concepts underlying GObject. This section introduces those concepts.

GObject is organized as a collection of modules. The fundamental modules are *GType*, *GSignal*, and the base *GObject* class. Each of these modules is described in further detail below. For further details, please see the GObject documentation [GObject, 2007].

### 5.2.1 GType

*GType* is at the core of GObject. Its basic functionality is to manage the definition, registration and introspection of types at run-time. The main commonality between all GTypes, as they are called, is that they define a method for copying their values. This allows generic memory management for every value with a GType. Those GTypes that directly define a copy mechanism, instead of inheriting one, are known as *fundamental* types.

The set of fundamental types includes many of the built-in C data types. For example, “primitive” types like integers, doubles, and strings (character pointers) are all fundamental types. Arbitrary C structures are adapted to the GType framework by providing a copy function and free function for the structure. Such types are said to be “boxed” types and inherit from the type *GBoxed*. For example, RGtk2 registers a boxed type for the R *SEXP* structure, which is used to represent all R objects.

The GType module also supports the definition of object types (the main purpose of GObject). Like all GTypes, object types must be or inherit from a fundamental type. GObject provides a fundamental type, *GObject*, that may be extended to define a new type of object. Every type derived from *GObject* has a C structure representing its class. Inheritance of class

structures is accomplished through the standard C idiom for object-oriented programming: prefixing a structure with the structure of the parent class, so that fields are aligned. The use of the structure prefixing idiom restricts *GObject* to single inheritance. The class structure contains class-wide fields, including function pointers called *virtual functions* that may be overridden by changing the value of the corresponding field in the class structure during initialization. This is the primary mechanism in *GObject* for changing class behavior through inheritance. Each object type also has a registered structure with instance-level members (i.e. fields). The instance structure of a type inherits from the parent instance structure using the same idiom as the class structures. An instance of an object type is manifested as a value of the corresponding instance structure. In order to link an instance to its class, each instance structure holds a reference to the shared value of the class structure for the type.

Like many object-oriented languages, *GObject* supports the definition and implementation of *interfaces*. An interface specifies a set of methods that represent a role performed by one or more classes, where the role is shared independently of the class hierarchy. If a class plays a role represented by an interface, it may formally declare the contract by registering itself as an implementation of the interface. As a result, the type is required to provide values (implementations) for the methods declared by the interface. Any object type, such as *GObject*, may implement multiple interfaces. Like a *GObject*-derived type, an interface has a class structure that declares its virtual functions (i.e. methods). Every interface class structure may only be prefixed by *GTypeInterface*, so there is no inheritance between interfaces in *GObject*. This is a significant difference from many object-oriented languages. However, an interface can be made to *require* the implementation of one or more other interfaces by any type that implements it. Unlike the *GObject* type, *GTypeInterface* is non-instanciable, so there is no instance-level structure and it is not possible to create instances of interfaces directly.

Two other fundamental types are *GEnum* and *GFlags*, both of which are registered with a class structure. The *GEnumClass* structure stores metadata about a particular enumeration, such as the names and nicknames of its values. *GFlags* is similar as it represents an enumeration where the values are intended to be combined bitwise (via AND and OR operations) to

represent the presence of one or more settings.

### 5.2.2 GSignal

One of the defining characteristics of GObject is its emphasis on *signals*, which were introduced earlier in this paper in the context of notification of user events in a GTK+ GUI. Any instance of a GType can have registered signals. Each *signal* is defined by its name and the types of its arguments and return value. A class inherits signals from its parents.

### 5.2.3 The GObject Base Class

*GObject* is the basic/fundamental classed and instantiable type provided by the GObject library. The key feature provided by the *GObject* class, from the perspective of the RGtk2 user, are *properties*. Properties may be thought of as introspectable and encapsulated public fields. Like instance fields of a *GObject*-derived type, properties are inherited. They support automated validation of their values at runtime, and a change in a property value emits the *notify* signal from its instance, allowing objects to respond to changes in the state of other objects. It is possible to control whether a property is readable, writeable, and more. Depending on the options specified in the declaration of a property, declaration, one may be able to or even restricted to set a property at construction time, using the generic *GObject* constructor, *gObject()*.

A property is defined by a *GParamSpec* structure that specifies a name, nickname, description, value GType, and other options. There are subclasses of *GParamSpec* for particular GTypes that permit specification of further constraints. For example, *GParamSpecInt* is specific to integers and can be configured to restrict its valid range of integer values between a minimum and maximum. Many *GParamSpec* subclasses also permit default values.

## 5.3 Interfacing With External GObject-based Applications

Many of the RGtk2 functions developed for the creation of GUIs using GTK+ are applicable to other libraries and applications based on GObject. There are several such packages of

interest to staticians, including Gnumeric, a spreadsheet application, and GGobi, software for multivariate interactive graphics. The rggobi package [Temple Lang, 2001b] provides a high-level interface to GGobi from R. Although it is somewhat hidden, rggobi objects are *externalptrs* that reference the underlying GGobi objects, which extend *GObject*. RGtk2 uses the same R representation, so many RGtk2 functions can operate on rggobi objects directly without additional interface code.

As an example, we consider the problem of displaying an R plot in response to a user “identifying” a point in a GGobi plot with the mouse. When a GGobi point is identified, the main GGobi context emits the “identify-point” signal. If we connect an R function to this signal, using *gSignalConnect*, the function will be executed whenever a point is identified. The following code displays data within a GGobi window and draws a fit of the simple linear model in a separate R graphics window. When the user identifies a point in the GGobi plot, the corresponding point is highlighted in the R display, providing simple linking.

```
library(rggobi)
attach(mtcars)
gg <- ggobi(mtcars)
model <- lm(mpg ~ hp)
plot(hp, mpg)
abline(model)
gSignalConnect(gg, "identify-point",
               function(gg, plot, id, dataset) {
                  plot(hp, mpg)
                  points(hp[id+1], mpg[id+1], pch=19)
                  abline(model)
               })
)
```

The GGobi instance is initialized with the *mtcars* dataset. A linear model is fit with *lm* and the line is drawn on an R plot. The important step is connecting a handler to the “identify-point” signal. The handler regenerates the R plot, and, for the identified point, replaces the

empty circle glyph with a filled circle. In this way, we have created a simple integration of the interactive graphics of GGobi with an R graphic that displays a linear model fit, which GGobi cannot display. More interesting integration uses the same basic tools.

Since the GGobi GUI is based on GTK+, it is possible to embed GGobi displays into RGtk2 GUIs, but that interface is still in flux and will not be detailed here.

## 5.4 Defining GObject Classes

All of the above examples utilize objects that are implemented in C. RGtk2 supports the definition of *GObject*-derived classes from within R. The *gClass* function in R registers a class, given the name of the new class, the name of the parent class, and the class definition. The class definition is a series of arguments that specify the new fields, new methods, methods that override inherited methods, signals, properties, and initialization function for the class. The name of a parameter specifies its role in the definition.

### 5.4.1 Example of defining a class

The example below illustrates the definition of a new *GObject*-derived class by revisiting the example in Section 3.4 involving the embedded plotting of microarray data. The slider in that example controls the alpha level of the points in the scatterplot in a linear fashion. Given the large amount of overplotting, the alpha level does not have a strong visual effect until it approaches its lower limit. One may desire greater control in this region, without limiting the range of the slider.

A possible solution would be to map the slider value to an alpha value using a non-linear function. All that is required is to change the slider callback so that it computes the alpha value as a non-linear function of the slider value. However, the label on the slider would be inaccurate; it would still report the original value. Overriding how the label is computed is possible by connecting a handler to the “format-value” signal on the *GtkScale* class. Let us assume, however, that we would like to create a reusable type of slider that mapped its value using a specified R expression.

Below is our invocation of *gClass* that defines *RTransformedHScale*, an extension of *GtkHScale*, the horizontal slider.

```
tform_scale_type <- gClass("RTransformedHScale", "GtkHScale",
  .props = list(
    gParamSpec(type = "R", name = "expr", nick = "e",
               blurb = "Transformation of scale value",
               default.value = expression(x))
  ),
  .public = list(
    getExpr = function(self) self["expr"],
    getTransformedValue =
      function(self)
        self$transformValue(self$value)
  ),
  .private = list(
    transformValue = function(self, x) eval(self$expr, list(x = x))
  ),
  GtkScale = list(
    format_value = function(self, x)
      as.character(self$transformValue(x))
  )
)
```

The third argument to *gClass*, *.props*, is a list containing property definitions. Each property is defined by a *GParamSpec* structure created using the *gParamSpec* function. *RGtkTransformedHScale* defines a single property named “expr” for holding the R *expression* that performs the transformation, e.g. *x^3*. Definitions of properties may refer to any GType by name. The names of primitive R types, like *integer* and *character* are mapped to the corresponding (scalar) GType, if available. It is also possible to specify the *RGtkSexp* type, as we

have done for *RGtkTransformedHScale* using the shorthand alias *R*. The Values of type *RGtkSexp* are left as native R objects instead of being converted to a C type, allowing the storage of R types that do not have a conventional C analog, like expressions, data frames, fitted models and S4 objects. For *RGtkSexp* properties, it is possible to specify the underlying R type for validation purposes. In our example, that type is inferred from the default value, which is of mode *expression*. The “any” type allows an *RGtkSexp* property to hold any R type. Normally, the class defining a property is responsible for handling the getting and setting of it. In order to override the management of a property defined by a parent class, the name of the property should be included in a character vector passed as an argument named *.prop\_overrides* to the *gClass* function.

Methods and fields may be encapsulated at the public, protected or private level. Public members may be accessed by any code, while protected members are restricted to methods belonging to the same class or a subclass. Access to private members is the most restricted as they are only available to methods in the same class. *gClass* has a separate parameter for each level of encapsulation. The values should be lists and are named according to their level of encapsulation: *.public*, *.protected* or *.private*. The functions for the methods and the initial assignments for the fields should be passed in the relevant parameter. The name of a member in a list serves as its identifier. In our example above, we define two public methods, *getExpr* and *getTransformedValue*, for retrieving the transformation expression and the transformed value, respectively. There is one private method, *transformValue* that is a utility for evaluating the expression on the current value.

Any virtual function defined by an inherited class or registered interface may be overridden. Like methods, virtual functions are implemented as R functions. In the *RGtkTransformedHScale* example, we override the *format\_value* virtual function in the *GtkScale* class to display the transformed value in the label above the slider. We first define the R function that implements the new behavior. Next, since the *gClass* function requires all overrides of methods from a particular class to be grouped together in a list, we create a list for *GtkScale*. We then add our R function to the list as an element named “*format\_value*”. This informs *gClass* that

we are overriding the *format\_value* method.

Any public or protected method defined in R may be overridden in R as if it were a virtual function. This is useful when the new class extends a class that itself is defined in R. Methods external to R may only be overridden if they are virtual functions.

A function implementing a virtual function may delegate to the function that it overrides from an ancestor class. This is achieved by calling the *parentHandler* function and passing it the name of the method and the arguments to forward to the method. For example, in the override of *format\_value* in the *RGtkTransformedHScale* class, we could call *parentHandler("format\_value", self, x)* to delegate to the implementation of *format\_value* in *GtkScale*.

Two elements of the class definition that are not in the example above are the list of signal definitions and the initialization function. The signal definition list is passed as a parameter named *.signals* and contains lists that each define a signal for the class. Each list includes the name, return type, and parameter types of the signal. The types may be specified in the same format as used for property definitions. The initialization function, passed as the *.initialize* parameter, is invoked whenever an instance of the class is created, before any properties are set. It takes the newly created instance of the class as its only parameter.

The return value from the call to *gClass* is the identifier of the new *GType*, and this can be used in calls to create instances of this type.

The next step in our example is to create an instance of *RGtkTransformedHScale* and to register a handler on the “value-changed” signal that will draw the plot using the transformed value as the alpha setting.

```
adj <- gtkAdjustment(0.5, 0.15, 1.00, 0.05, 0.5, 0)
s <- gObject(tform_scale_type, adjustment = adj, expr =
expression(x^3))
gSignalConnect(s, "value-changed", function(scale) {
  plot(ma_data, col = rgb(0,0,0,scale$getTransformedValue()),
  xlab = "Replicate 1", ylab = "Replicate 2",
```

```

    main = "Expression levels of WT at time 0", pch = 19)
})

```

Instances of any GObject class may be created using the *gObject* function. The value of the “expr” property is set to the R expression  $x^3$  when the object is created. The signal handler now calls the new *getTransformedValue* method, instead of *getValue* as in the original version. This final block of code completes the example:

```

win <- gtkWindow(show = FALSE)

da <- gtkDrawingArea()

vbox <- gtkVBox()

vbox$packStart(da)

vbox$packStart(s, FALSE)

win$add(vbox)

win$setDefaultSize(400,400)

require(cairoDevice)

asCairoDevice(da)

win$showAll()

par(pty = "s")

s$value(0.7)

```

More precise details on defining GObject classes are available in the R help page for the *gClass* function.

## 6 Language binding design and generation

### 6.1 Goals and Scope

There are two primary concerns for the design of RGtk2: consistency and efficiency of use. In terms of consistency, the API should be consistent with R first and GTK+ second.

RGtk2 aims to provide a complete and consistent interface to the GTK+ API, except where that would conflict with R conventions. This is based on the assumption that the GTK+ API has been designed to be used as a whole. We purposefully avoid any attempt to limit the bindings to what we might consider the most useful subset of GTK+. Only functionality that would introduce foreign concepts to R, such as memory management, return-by-reference parameters, and type casting, is excluded from the RGtk2 interface. It should not be obvious to the user that GTK+ is implemented in a foreign language. As a consequence of consistency with GTK+, RGtk2 provides a fairly low-level interface, which likely detracts from its ease of use. To rectify this, GTK+ aims to increase the usability of its API. Towards this end, it provides facilities like the *RGtkDataFrame* utility and the custom syntax for calling methods and accessing properties.

In addition to GTK+, RGtk2 also provides bindings for Cairo, GDK, GdkPixbuf, Pango, ATK, and Libglade. All of these libraries were designed with language bindings in mind, and, except for Cairo, they are all based on the GObject framework. The API for Cairo is sufficiently simple that its independence from GObject is of little consequence. As a result, there are no significant binding issues that are particular to a single library, so the discussion of GTK+ suffices for all of the bindings.

With the exception of properties and signals, which are bound at runtime using introspection, the RGtk2 bindings, including functions, methods, fields, virtual functions, callbacks and enumerations, are based on programmatically generated code connecting R and the C routines and data structures. This section continues by detailing the code generation system and the type conversion routines utilized by the generated code. It concludes by introducing the system for autogenerating the R documentation for the package. The explanations assume the reader has a working knowledge of the GObject system (see section 5.2).

## 6.2 Automatic Binding Generation

Given the broad scope of the project, it was decided that developing a system for automatically generating the interface would be more time efficient than manual implementation.

```
(define-object Widget
  (in-module "Gtk")
  (parent "GtkObject")
  (c-name "GtkWidget")
  (gtype-id "GTK_TYPE_WIDGET")
  (fields
    '(("GtkStyle*" "style")
      ("GtkRequisition" "requisition")
      ("GtkAllocation" "allocation")
      ("GdkWindow*" "window")
      ('("GtkWidget*" "parent"))
    )
  )
)
```

Figure 12 An example of the *defs* format for specifying the API of GObject-based libraries. This particular expression describes the *GtkWidget* class.

Autogeneration also enhances the maintainability of the project, since improved code can be uniformly and programmatically generated across for new versions of each library. Additionally, this allows us and other users to programmatically generate interfaces to other libraries. This section describes the design of the code generation system, beginning with the input format and then explaining how each component of the bindings is generated.

### 6.2.1 The *defs* format

The GTK+ API and other GObject-based API's are often described by a Scheme-based format called *defs*. A *defs* file describes the types and functions of an API. The autogeneration system for the RGtk2 bindings takes *defs* files as its input. This section briefly describes the *defs* format and how it is leveraged by RGtk2. It concludes with a discussion of alternative API description methods.

The *defs* format supports six different kinds of types: objects, interfaces, boxed types, enumerations, flags and pointers. Each of these correspond to a fundamental GType (see section 5.2.1). Every type of definition has a field identifying the module in which it is contained (usually the name of the library or API), its C symbol and its GType, with the exception of

raw pointer types, which lack a specific GType. The objects, boxes, and pointers may contain a list of field definitions, each consisting of the type and name of a field. The type names are formatted as they are in C except for some special syntax for indicating arrays and specifying the type of the elements in a list. Object definitions have a field for the parent type, while definitions of boxed types specify the copy and free functions of the type. Each enumeration and flag definition contains a list of their allowed values. As an example, the *defs* representation of the *GtkWidget* object is given in Figure 12.

In addition to types, the *defs* format supports definition of four kinds of invocable or callable elements: functions, methods, virtual functions and callbacks. All callable definitions contain the C symbol, a return type, whether the caller owns the returned memory and a list of parameter definitions.

Each parameter definition contains a type, name, parameter direction (in or out), optional default value and optional deprecation message. Parameter direction refers to whether a parameter is passed as input (*in*) to the function or is part of the return value (*out*), which is known as “return by reference” in C. An example is retrieving the dimensions of a *GdkDrawable*, the rectangular target of GDK drawing operations. The method *gdkDrawableGetSize* has two integer out parameters, *width* and *height*. A few parameters in the bound API’s are sent in both directions, but these so-called *inout* parameters are so rare that we handle them manually. Parameter types are formatted like field types.

There are slight differences in the way the different types of callables are defined in the *defs*. Functions may be marked as constructors, i.e. for creating objects of a specified type. Methods and virtual functions belong to an object or interface type. This distinguishes them from plain functions and callback functions, which are independent of a class. The name of the type declaring the method or virtual function is specified in the definition. Below is an example of the *getSize* method on *GtkWindow*:

```
(define-method get-size
  (of-object "GtkWindow")
  (c-name "gtk_window_get_size")
```

```

  (return-type "none")

  (parameters
    ('("gint*" "width" (out))
     ('("gint*" "height" (out))
      )
    )
)

```

The Python binding to GTK+, PyGTK [Chapman and Kelley, 2000], provides Python classes for the generation and parsing of *defs* files. The generation scripts scan C header files for information about an API. The autogenerated *defs* file is then manually annotated with information that is not derivable from header files, such as that regarding memory ownership. PyGTK maintains a set of reference *defs* files for every library bound by RGtk2 except Cairo, for which a *defs* description was created as part of this work.

RGtk2 leverages this information as input to its binding generation system. The system is implemented in R and calls the PyGTK *defs* parsing code via the RSPython [Temple Lang, 2005c] package. The resulting descriptions are converted to R and from these the interface code is generated, consisting of both R and C binding code. In the great majority of cases, the information provided by a *defs* file is sufficient for autogeneration of bindings. However, there is a small number of functions that require manual implementation, such as those with variadic arguments or complicated memory ownership policies.

There are some alternatives to the *defs* format. The GTK# project [Bernstein Niel, 2004], which binds GTK+ to the .NET platform, has defined the XML-based *GAPI* format [Project, 2008]. *GAPI* contains essentially the same information as *defs* files, but the *GAPI* tools allow the raw API description, which is normally derived automatically from the header files of the library, to be stored separately from the manual annotations. The raw definitions and annotations are merged when generating the code for an interface. This facilitates maintenance of the interface definitions. The *defs* tools from PyGTK do not support this, although filtering using regular expressions and storing the changes as *patch* or difference files works fairly well. *GAPI* came long after the introduction of RGtk, and it was decided that there were not enough

advantages over *defs* to justify a switch. A second XML-based format, *GIDL* [GIDL, 2005], has recently been developed as a unifying standard for representing GObject-based API’s. Although no official tools for generating *GIDL* yet exist, it holds promise for being accepted as a standard, as it has the backing of GTK+ developers. The use of XML as input to our code generation system would substitute the dependency on the RSPython package with the XML package and this might prove simpler.

### 6.2.2 The Generated Bindings

This section introduces each component of the bindings output by the code generation system. The components include a set of wrappers for each callable type, an accessor function for each field, and code for creating an R vector for each enumeration flags type.

**Function and Method Wrappers** Functions and methods are mapped to R functions of the same name, transformed to camelBack form, i.e. words concatenated and the first letter in upper case, except for the first word. Although an object-oriented syntax for methods is supported, its use is not mandatory; every API call is possible through an R function. This results in an interface that is familiar to the R programmer. Each function and method definition in the *defs* input is converted to two wrapper functions, one in R and the other in C. The R wrapper is responsible for coercion of the parameters to the R types that correspond to the C types of the parameters of the underlying C function. This includes checking the “class” attribute of the *externalptr* objects for the expected type. It is considered simpler, safer and more maintainable to perform the coercion in R than in C. The R wrapper will optionally emit a warning if the function is deprecated. It then calls the C wrapper for the function, which converts the parameters from R types to C types and invokes the API function. The return value, if any, is converted from C to R. If there are any *out* parameters, these are also converted to R types and bundled with the return value in a list. This avoids the foreign concept of return-by-reference in R. The result is then returned to the R wrapper. If the function is a widget constructor, an extra optional parameter (*show*) is added to the generated R function and this controls whether the newly created widget will immediately be made visible. Finally,

the result is returned to the user.

The following is an example of this process for the function *gtkWidgetCreatePangoLayout*, which is a commonly used function for drawing text on a widget, such as a *GtkDrawingArea*. First, we present the autogenerated R wrapper, from the RGtk2 source code, reformatted to wrap long lines.

```
gtkWidgetCreatePangoLayout <-
function(object, text)
{
  checkPtrType(object, "GtkWidget")
  text <- as.character(text)

  w <- .RGtkCall("S_gtk_widget_create_pango_layout", object,
  text, PACKAGE = "RGtk2")

  return(w)
}
```

The wrapper ensures that the object is of type *GtkWidget* and coerces the text to display to a character vector. It then invokes the C wrapper with the validated arguments and returns the result. Below is the source code listing of the *S\_gtk\_widget\_create\_pango\_layout* function.

```
USER_OBJECT_
S_gtk_widget_create_pango_layout(SEXP s_object, SEXP s_text)
{
  SEXP _result = R_NilValue;
  GtkWidget* object = GTK_WIDGET(getPtrValue(s_object));
  const gchar* text = ((const gchar*)asCString(s_text));

  PangoLayout* ans;
```

```

ans = gtk_widget_create_pango_layout(object, text);

_result = toRPointerWithFinalizer(ans, "PangoLayout",
(RPointerFinalizer) g_object_unref);

return(_result);
}

```

The R types are converted to C types and passed to the actual GTK+ function. The result, a *PangoLayout* object, is converted to an R *externalptr* type and returned.

**Constructors** There are often several constructor routines for a given GTK+ class, e.g. for *GtkButton* there is *gtkButtonNew*, *gtkButtonNewWithLabel*, *gtkButtonNewFromStock*. While bindings for each of these are available, we also want the R programmer to be able to use a single general purpose constructor function, e.g. *gtkButton()*. Depending on which arguments are provided to *gtkButton()*, the code decides which of the low-level constructors is to be invoked.

For each object class, the high-level constructor function (e.g. *gtkButton()*), is programmatically generated. Its parameter list matches the union of all of the parameter lists for each constructor of the class. The function body delegates to one of the constructors based on which parameters are provided by the user. As with *GtkButton*, the name of the constructor is the name of the class with the first character in lower case.

As an example, the code for the programmatically generated *gtkButton()* function is given below. The *GtkButton* class has three constructors which correspond to the functions *gtkButtonNewFromStock(stock.id)*, *gtkButtonNewWithLabel(label)* and the basic *gtkButtonNew*, which takes no arguments. From these, we generate the following code:

```

gtkButton <- function (label, stock.id, show = TRUE)
{

```

```

if (!missing(stock.id)) {
    gtkButtonNewFromStock(stock.id, show)
}
else {
    if (!missing(label)) {
        gtkButtonNewWithLabel(label, show)
    }
    else {
        gtkButtonNew(show)
    }
}
}

```

This then allows the R programmer to use calls such as

```

gtkButton()
gtkButton('my label')
gtkButton(stock.id = '...')


```

**Callback Wrappers** Callbacks are functions that are passed to and returned from functions and methods in the API of a library. After a callback is registered, the library may invoke it to perform a particular task, and, in so doing, it *calls back* into client code. Thus, callbacks are one means for a client to customize part of the functionality of a library.

An example of a function that registers a callback is *gtkTreeViewColumnSetCellDataFunc*, which is used by the *create\_tree\_view* function in section 4.4.1 to customize the rendering of the numeric values as text in the spreadsheet. There are two integral components of the bindings that enable support for callback functions. First, there is the wrapper for each callback registration function, like *gtkTreeViewColumnSetCellDataFunc*. Then, for each callback function, there is a C implementation of the callback that delegates to the R function registered as the callback by the user.

In general, callback registration functions take two parameters: the callback function and a “user-data” structure that is conventionally passed as the final argument to the callback function. Similar to connecting a handler to a signal, the user needs to provide an R function as the callback function. The “user-data” parameter is optional; if given, it may be any R object.

When wrapping a callback registration function, special handling is required for its parameters. As the underlying C registration function requires a C function for its callback parameter, we pass an autogenerated C function, which delegates to the user-provided R callback function (see below). In order to provide the R function to the C wrapper, we place the R function, as well as the R “user-data” (if any) in a structure and pass that structure to the underlying C registration function as the “user-data”.

The generated code for the C wrapper that delegates to the user-provided R callback is similar to that of an ordinary function wrapper, except the flow of control is in the opposite direction. When invoked, the function retrieves the R function and “user-data” object from the special structure passed as the “user-data” to the wrapper. The wrapper then converts its parameters to their R equivalents, calls the user-provided R function, and returns the result after converting it to its C equivalent.

**Virtual Function Wrappers** Virtual functions, like *format\_value* in the *RTransformedHScale* class defined in section 5.4.1, are bound to R in order to support the extension of GObject classes. Wrappers for virtual functions are generated for both directions, from R to C, like the function wrappers, and from C to R, like callbacks.

Although virtual functions are not public like methods, they are bound in the forward direction for calling the overriden implementation of a virtual function in a parent class. The generated code in this case is very similar to that for wrapping ordinary functions and methods.

The reverse wrapper, from C to R, is needed to support the overriding of virtual functions through inheritance. The R functions implementing the virtual functions are stored within the GObject class structure. When a virtual function is invoked, the code searches for a corresponding R function. It is not guaranteed that one exists within the class structure, as

a class need not override every virtual function it inherits. If one is found, the R function is invoked in the manner described above for callbacks functions. If no overriding function is found, the code delegates to the implementation of the method in the parent class.

**Field Accessors** Fields, which are typically considered read-only in GObject API's, may be accessed in R using the extraction function `[[` as in `obj[[name]]`. The usage of `[[` is the same as when extracting a named element from an R list, which should be familiar to every R programmer. This mechanism is based on an R wrapper function named according to the scheme `classNameGetFieldName`, e.g. `gtkWidgetgetStyle` for retrieving the `style` field from an instance of `GtkWidget`. This function works much the same as the function bindings introduced above, except the C wrapper accesses a field of a C structure rather than invoking a function, and converts the value from C to R.

**Enumeration and Flag Definitions** Although the function wrappers accept the string representations of enumerations and flags, as that is likely familiar to R programmers, there are some cases, such as in the example in Section 3.1 involving `GtkResponseType` and when performing bitwise operations on flags, that the numeric values of enumerated types are required. The code generator outputs definitions of R numeric vectors with the names corresponding to the string representation of each value.

## 6.3 Type Conversion

### 6.3.1 Overview

Most of the work on RGtk2 outside of autogeneration deals with type conversion. Conversion of strings and primitive C types, such as `int` and `double`, is relatively obvious and simple. Pointers to C structures are converted in two different ways, generally referred to as “high-level” and “low-level” type conversion. High-level conversion is the translation between a C structure and a native R object, such as a list. The class attribute on the object normally corresponds to the type of the original C structure. The alternative is low-level conversion to and from R `externalptrs`. For consistency, the method of conversion is the same for a partic-

ular structure type in both directions, to and from C. Collections, such as arrays and linked lists, are converted by iterating over the data structures, converting each element and storing the result into an R list. This section continues with further details on the two methods for converting C structures, and this is followed by explanations of array and error conversion.

### 6.3.2 High-level Conversion

High-level structure conversion either produces or consumes a native R object instead of a low-level *externalptr*. The advantage of a native R value is more obvious integration with R. In particular, reference semantics are avoided. However, due to performance considerations, information hiding, library design, and other constraints, high-level conversion is only feasible in certain cases. One rare case is where a complex C type has a clear analog in R. An example of this is the *GString* structure, which is a convenience wrapper around an array of characters. This is naturally mapped to an R *character* vector of length 1, i.e. a single string and not a vector of characters. The more common second case is the conversion between C structures and R lists, where each field of the structure is represented by an element in the list, in the same order. The names of the list elements match the names of the structure fields.

Structures qualify for the second case if they are meant to be initialized directly in C and therefore lack a constructor. Although a new function could be introduced as a constructor, this would introduce an unnecessary inconsistency between R and C. experience, if the underlying API requires that a structure be initialized directly, it is feasible to perform high-level conversion on the structure. An example of this type of high-level conversion may be found in the spreadsheet example in Section 4. The actions for the menu and tool bar are specified as lists; no external references are created.

### 6.3.3 Low-level Conversion

The use of low-level *externalptr* objects for the underlying C structures is likely unfamiliar to most R programmers, but, in general, it is difficult to avoid. The primary reason is that the C libraries depend on the treatment of many structures as references. For reasons connected

to run-time “safety” and method dispatch, the type of the pointer, as well as the entire class hierarchy in the case of an object, is stored as a character vector in the *class* attribute of the R object. This is used, for example, when validating inputs in function wrappers, as well as for determining the function to call when the user employs the object-\$-method syntax.

An important consideration when handling references is memory management, which needs to be hidden from the R user. The base policy is that memory is preserved until it is no longer referenced by R. This relies on the R garbage collector and the reference counting of *GObject*. Boxed structures are copied using their copy function and registered for finalization using their free function. Instances derived (directly or indirectly) from the *GObject* class are managed using a reference counting scheme. The reference count is incremented when a reference is obtained and decremented when the reference is finalized. In cases where memory ownership is transferred implicitly, such as when an object is constructed, it is not necessary to claim ownership by copying or increasing an reference count.

There are two cases where the above mechanism is insufficient: C structures without GTypes and objects derived from *GtkObject*, which serves as the base class *G GtkWidget*, as well as several other GTK+ classes. When a structure lacks a GType, RGtk2 does not know how to manage its memory. Thus, the structure is passed to R without copying it or otherwise transferring the ownership of the memory to R, in the hope that the memory is not freed externally. Thankfully, these types of structures are rare. Most of them are converted to high-level R structures, which avoids holding a reference.

The second exception is *GtkObject*, which extends *GObject* to support explicit destruction via the *gtkObjectDestroy* function. When that function is invoked, the “destroy” signal is emitted. All parties that hold a reference to the object are required to respond to the signal by releasing their reference. This functionality is useful for destroying widgets when they are no longer needed, even if other parties hold references to them. However, it also means that the R references to the object will become invalid even though they are still visible to the R session. When a reference to a *GtkObject* is obtained, the RGtk2 package transparently connects a handler to its “destroy” signal. Besides releasing the reference, the signal handler modifies the

*class* attribute of the *externalptr* to a sentinel value indicating that the reference is invalid. If the programmer attempts to use an invalidated reference, an error will be thrown. This silent modification of the class attribute may surprise the R programmer, but it avoids fatal errors that may corrupt the R session (e.g. segmentation faults).

### 6.3.4 Arrays

C arrays are converted to R lists, with each element converted individually. The primary complication is that C arrays do not track their length. Unless an array is terminated by a sentinel value, there is usually no way to determine the length from the array itself. This requires C functions to accept and return array length parameters along with arrays. Array length parameters need to be hidden from the R programmer, since R vectors have an inherent length. The code generator uses heuristics to identify array length parameters and does not require the R programmer to provide them. For example, if an array parameter is followed by an integer parameter, the generator will assume the integer parameter specifies the length of the array. A specific example of this is the function *gtkActionGroupAddActions*, which is called in section 4.2.2 as part of the spreadsheet application example. The function wraps the C function with the signature *gtk\_action\_group\_add\_actions(GtkActionGroup \*action\_group, GtkActionEntry \*entries, guint n\_entries, gpointer user\_data)*. The generator guesses that the unsigned integer *n\_entries* parameter represents the length of the array of *GtkActionEntry* structures passed as *entries*. For input parameters, the wrapper passes the length of the input R list as the array parameter. For returned arrays, a similar heuristic finds the returned length and uses it when converting the array to an R list. For example, the C function *gtk\_recent\_chooser\_get\_uris(GtkRecentChooser \*chooser, gsize \*length)* returns an array of strings, containing the URI's in a recent file chooser widget. The generator identifies the *length* return-by-reference (out) parameter as representing the length of the returned array. This heuristic is slightly less reliable compared to the one for input arrays, since there is no array parameter to search around for a length parameter. In cases where these heuristics fail, we manually implement the function wrapper (relying on the code generator for a head start).

### 6.3.5 Errors

Certain errors that occur in GLib-based libraries are described by a returned *GError* structure. In R, the user is often alerted to a problem via a condition emitted by the *stop()* or *warning()* functions. The user may pass a value of *TRUE* or *FALSE* as the value of the *.errwarn* parameter to any wrapper that might raise a *GError*. If *.errwarn* is *TRUE*, a warning is raised. Alternatively, if *.errwarn* is *FALSE*, no warning will be emitted and the user can inspect a returned list structure containing the fields of the *GError*, which often holds more information compared to the warning string. In the future, a new type of R-level condition may be added for a *GError*, but the system currently emits only warnings.

## 6.4 Autogeneration of the Documentation

The final design consideration is the documentation of the bindings, which is also accomplished by auto-generation. A relatively easy approach would be to generate a single documentation file with an alias for all of the functions and data structures of a particular library. That file could contain a reference to the library's C documentation on the web. However, referring the user to C documentation would have several disadvantages. First, most R programmers are likely not familiar with C. Second, there would be a number of significant inconsistencies in the API. This might confuse even an experienced C programmer. For example, RGtk2 hides function parameters that specify the lengths of arrays, since these are always known in R. The existence of these in the C documentation would confuse the R user. Other inconsistencies would be return-by-reference parameters and the names of data types. Also, the C documentation would omit concepts such as high-level structure conversion.

Fortunately, all of the bound libraries rely on the gtk-doc utility that produces documentation as Docbook XML. The XML representation may be parsed into R using the XML package [Temple Lang, 2001a]. From within R, it is possible to introspect the bindings and access the API descriptions stored in the *defs* files. By combining this information with the original documentation, the documentation generator is able to output R help files that are consistent with the RGtk2 API. Embedded C examples are replaced with their R equivalent by looking up an

R translation by the name of the example. The translation is done manually. The generator attempts to filter out irrelevant statements, such as those regarding memory management, though many C-specific phrases still exist in the output. Thus, the documentation of RGtk2 is still very much a work in progress.

## 7 Technical Language Binding and GUI Issues

### 7.1 Fully Programmatic Binding Generation

The strategy of autogenerated the bindings saves a significant amount of time and facilitates maintenance, but it is not without its problems. The *defs* files as generated from the header files do not contain all of the information necessary to correctly generate bindings to many of the C functions. This requires human annotation of the *defs* files. The two most common types of required annotation are the direction of parameters (in, out or inout) and the transfer of memory ownership. There is no way to determine this information from the header files.

One way the machine might programmatically determine information about return-by-reference parameters, memory management and other aspects would be to inspect the C source code of the library in addition to or instead of the header files. The RGCCTranslationUnit package [Temple Lang, 2006a] provides a framework and some tools to support such inspection.

Another solution would be to require the authors of the API to include the missing information as specially formatted comments in the source code. The comments could even be part of the inline documentation, as it would be beneficial to state such information in a standard way in the documentation, as well. This method does not avoid human annotation and there is the potential for the code and the documentation to become unsynchronized, but the benefit is that the annotations are centrally maintained by an authoritative source.

A variation on the above idea would be to support registration of functions, with all information necessary for binding, during class initialization, just as signals and properties are currently. This would render the entire API of a library introspectable at runtime; compiled bindings would no longer be necessary. However, runtime introspection of functions would

have a high performance cost due to the need to lookup the information each time it is needed and the consumption of a large amount of memory. One way around this would be to use the information for generating a compiled interface but not to load the information during normal use of the library. Still, the previous solution of storing the information in comments would have the advantage of being accessible without linking to the library.

A more radical solution would be to write libraries in an entirely different language, which compiled down to GObject-based C code. The design of the language would ensure that all information necessary for binding would be known to the compiler. Such a language already exists, named Vala [Billeter, 2007]. Vala is an object-oriented language with a C# syntax and features like assisted memory management, lambda expressions and exceptions. The Vala compiler provides an API for inspecting the parsed language, from which binding information like memory management and function parameter directions may be obtained. Of course, this solution would require an existing library to be completely reimplemented in Vala, so it may only be feasible in the future, if and when Vala becomes more widespread. Another drawback is that it introduces an additional compilation stage and thus complicates the build process.

## 7.2 RGtk2 as a Base for Other GObject Bindings

Implementations of most programming languages are still written in C. This suggests that libraries implemented in C are likely accessible to more languages than those implemented in Java, for example. GObject is designed with language bindings in mind. Given this incentive, it is likely that the number of GObject-based libraries will continue to grow.

RGtk2 has been designed to serve as a base for other R packages binding to GObject-derived libraries. The mechanism introduced by R 2.4 for sharing C interfaces between packages allows RGtk2 to export all of its C-level utilities for interacting with GObject, including type conversion routines, wrappers for the GObject API, and functions for extending GObject classes. This support has already been used by an experimental version of rggobi [Lawrence et al., 2007b]. If this functionality proves to be of general use, it should probably be split out of RGtk2 as a base binding to GObject. In conjunction with this, the binding generation system

should be revised and made public, as was done for the original RGtk package.

### 7.3 Event Loop Issues

All user interfaces need to respond to user input. GTK+ provides an *event loop* that checks for user input and executes application callbacks when necessary. GTK+ applications written in C usually execute the GTK+ event loop after initialization. The loop takes is started and continues processing events until the GUI is terminated. The interactive R session is a user interface, and it has its own event loop. When using RGtk2 from an interactive session, there are two event loops, R and GTK+, trying to process the user input at the same time.

By default, RGtk2 attempts to reconcile the two loops by delegating to the GTK+ event loop when the R event loop is idle. In general, both interfaces operate as expected under this configuration. However, as the GTK+ event loop is not iterated continuously, certain operations, in particular timer tasks, are not executed reliably. While it is not expected that many RGtk2 users will rely on timers, several GTK+ widgets use timers for animation purposes. These widgets tend not to be as responsive as the others without reliable iteration of the GTK+ event loop. One solution to this problem is to invoke the R function *gtkMain*, which transfers control to the GTK+ event loop and blocks the R console for the lifetime of that GUI. If the user is willing to sacrifice access to the regular R console, this is a viable method to enhance the responsiveness of the GTK+ GUI. Of course, an alternative command line interface can be provided by implementing it within a Gtk+-based GUI and so the user would have both. Indeed, we feel that R should not have its own event loop but rather be treated as library from other front ends. Another possible solution would be a multithreaded model, with synchronized access to the R evaluator. There has been some work towards a solution to this problem, such as the REventLoop package [Lang, 2003], but this remains an area for further research.

## 8 Comparison of RGtk2 to other R GUI toolkit bindings

There are many different ways to construct a GUI from R. All of them, at some level, depend on a binding to an external widget toolkit. Direct bindings exist for Tcl/Tk [Ousterhout, 1994, Welch, 2003] and wxWidgets [Smart et al., 2005], in addition to GTK+. Other toolkits are indirectly accessible across interfaces to DCOM [Microsoft Corporation, 2007] and Java [Sun Microsystems, 2007]. This section outlines the alternatives to RGtk2 for constructing GUIs in R, considering the features of both the R binding and the underlying toolkit.

The great majority of R GUIs rely on the tcltk package [Dalgaard, 2001, 2002] that binds R to tcl/tk [Ousterhout, 1994, Welch, 2003], a mature light-weight cross-platform widget library. Applications of tcltk range from limmaGUI [Smyth, 2005], a task-specific GUI for microarray preprocessing, to the more general R Commander [Fox, 2007]. The tcltk package is bundled with the core distribution of R. This means that developers can usually count on its availability. This is not the case for RGtk2, which requires the user to install RGtk2, GTK+, and all of the libraries on which GTK+ depends. The small footprint of tcl/tk likely delivers better performance in terms of speed and memory than GTK+ in many circumstances. tcl/tk also offers some features that base GTK+ currently lacks, the canvas widget being one example.

Unfortunately, tcl/tk development is slow and the library is beginning to show its age. It lacks many of the widgets present in GTK+ and other modern toolkits, such as tree tables, progress bars, and autocompleting text fields. tcl/tk widgets are often less modern or sophisticated than their GTK+ counterparts. For example, a GTK+ menu is able to be torn off as an independent window and the GTK+ file chooser supports the storage of shortcuts. tcl/tk also lacks theme support, so it is not able to emulate native look and feels. tcl/tk is not object-oriented, and it is not possible to override the fundamental behavior of widgets. While one can build so-called “megawidgets” on top of existing Tk widgets, this is not the same as creating new *GtkWidget*-derived classes with RGtk2. Moreover, the design goals of the tcltk package differ from those of RGtk2, in that tcltk aims to expose the functionality of the Tcl engine to the R programmer, while RGtk2 is a binding to a collection of specialized C libraries.

The Windows-specific tcltk2 package [Grosjean, 2006] is an attempt to overcome some of

the limitations of the `tcltk` package by binding the `Tile` extension [Tile, 2007] of `tcl/tk`. `Tile` adds support for themes, allowing emulation of native widgets and prettier GUIs, as well as new widgets like a tree table and progress bar. However, `Tile` still lags behind `GTK+`. For example, the `GTK+` tree table allows the embedding of images, check boxes, and combo boxes, while the `Tile` one does not.

`wxWidgets` [Smart et al., 2005] differs from `Tcl/Tk` and `GTK+` in that it provides a common API with platform-specific implementations based on the native widgets of each platform, and so preserves the look and feel of each platform, without resorting to emulation. In contrast, `Tcl/Tk` and `GTK+` provide exactly the same widgets on all platforms, leaving the look and feel to theme engines. `GTK+` serves as the “native” Linux implementation of `wxWidgets`. The first binding from R to `wxWidgets` is the now defunct `wxPython` package that leverages `RSPython` to access the Python binding to `wxWidgets`. `RwxWidgets` [Temple Lang, 2007] is a more recent binding that directly binds to the C++ classes of `wxWidgets`.

`wxWidgets` also lacks integrated 2D vector graphics. The `RwxWidgets` package does not yet bind to a parser of external GUI descriptions like `Libglade` for `GTK+`. However, `wxWidgets` does provide some features that do not exist yet in base `GTK+`, such as HTML display and a dockable window framework.

The `RDCOM` [Temple Lang, 2005a] and `R-(D)COM` [Baier and Neuwirth, 2007] packages provide an interface between R and DCOM [Microsoft Corporation, 2007]. This permits manipulation of existing GUIs, such as that of Microsoft Office or programmatically placing ActiveX controls on an Excel spreadsheet and, with the `RDCOMEvents` package [Temple Lang, 2005b], connecting R functions to their events. The `R-(D)COM` package has been used to create the educational R GUI `simpleR` [Maier, 2006]. A major drawback to the use of DCOM, however, is its dependence on Microsoft Windows. However, given the prevalence of Microsoft Windows, this is a significant benefit for those seeking to develop rich GUIs for that platform and integrating tools such as Excel, Word and Internet Explorer.

Java toolkits, including `Swing` and `SWT`, are also accessible from R through `R-Java` interfaces such `rJava` [Urbanek, 2006]. The features of `Swing` and `SWT` are comparable to those

of GTK+, and one could use rJava to develop Java-based GUIs. This would be facilitated by a high-level interface for GUI development built on top of the low-level interface provided by rJava.

Such an interface is delivered by the gWidgets package [Verzani, 2007a]. gWidgets provides a simplified, common-denominator-style API for GUI programming that, similar in spirit but not as complete as the approach of wxWidgets, is implemented by multiple toolkit backends. gWidgets is written in R, so its backends rely on bindings to the external toolkits. So far, there are three backends for gWidgets: gWidgetsRGtk2, based on RGtk2; gWidgetsJava, based on rJava and Swing; and gWidgetsTcltk for tcl/tk. A defining characteristic of gWidgets is the design of its API, which aims for simplicity and consistency with R conventions. The goal is to accelerate the construction of simple GUIs by those inexperienced with GUI programming. For this purpose, using gWidgets is likely a better course than direct use of RGtk2; however, the simplified interface hides functionality that more complex applications might find useful.

One benefit of RGtk2 (and RwxWidgets on Linux) is the capability to integrate with other GUIs based on GTK+. Such software includes GGobi, Mozilla Firefox (on some platforms), and Gnumeric. Widgets from these tools could be embedded in RGtk2-based GUIs. The rggobi package enables this for GGobi, a software tool for multivariate graphics.

## 9 Impact and Future Work

RGtk2 aims to provide a consistent and efficient interface to GTK+ for constructing GUIs in R. The design of the API prioritizes usability from the perspective of the R programmer. The package has been adopted by several projects, including: gWidgets [Verzani, 2007a], a simple interface for GUI construction in R; Rattle [Williams, 2006], a data mining GUI based on Libglade; and playwith [Andrews, 2007], a package for interactive R graphics. Future plans for RGtk2 include more fully automating the code generation process and keeping pace with frequent GTK+ releases.

**Supplemental information**

More information, including download instructions, are available at the RGtk2 website  
<http://www.ggobi.org/rgtk2>.

**Acknowledgements**

Michael Lawrence's work was supported in part by National Science Foundation Arabidopsis 2010 grants DBI-0209809 and DBI-052067. We also thank Dr. Dianne Cook for providing helpful feedback on the software and this paper.

## THE PLUMBING OF INTERACTIVE GRAPHICS

A paper To appear in Computational Statistics

Hadley Wickham, Michael Lawrence, Dianne Cook, Andreas Buja, Heike Hofmann, Deborah F. Swayne

### 1 Introduction

What is a pipeline, and why do we need one for interactive graphics? This conceptual paper attempts to answer these questions, building on previous work by Buja et al. [1988b] and Sutherland et al. [2000]. A pipeline controls the transformation from data to graphical objects on our screens, and we argue that the pipeline must be present, in some form, in all graphics software. The pipeline is made explicit in descendants of DataViewer [Buja et al., 1986].

The essence of a pipeline is data moving through a series of transformations. To allow updates from plot interaction data also must be able to move backwards along the pipeline, using the inverses of the original transformations. The metaphor of a pipe breaks down somewhat here, as pipelines typically flow in one direction, but also because there are multiple ways we can control the flow of data along the pipe. In the second part of this paper, we will discuss some possible means of control, and propose one design which we think is particularly appropriate for the needs of interactive graphics.

Finally, we will conclude with an discussion of how these ideas are being implemented in the next version of GGobi [Swayne et al., 2003a], a tool for highly interactive and dynamic data visualization.

## 2 Related work

Apart from the details of DataViewer [Buja et al., 1986] and its descendants, XGobi [Swayne et al., 1991], Orca [Sutherland et al., 2000], and GGobi [Swayne et al., 2003a], little has been published about the flow of data within interactive graphics software. In this section we summarize related work, drawing on academic publications, conversations with software authors, and the source code, where possible.

The early work of Fisherkeller et al. [1975], McDonald [1982] and Becker and Cleveland [1987] started the field of statistical graphics, but they published the uses of their new tools, not the details of their data structures. We might surmise, due to the speed of computation at the time, that their software was written very specifically for the task at hand: focusing on adequate speed for interaction, not generality.

Quail [Hurley, 1993, Hurley and Oldford, 1999], written in lisp, provided a powerful and flexible linking system. This system worked by connecting plots directly, not mediated through the data. mmvis [Ardis et al., 2000], each user action is tied to a *component*, which is responsible for updating the appropriate views. DAVIS [Huh and Song, 2002], another java application, experiments with tours.

XmdvTool [Ward, 1994] uses parallel coordinates, scatterplot matrices, glyphs and dimensional stacking to explore complex data simplified using hierarchical clustering. Recent work has given some insight into the data processing done behind the scenes: Doshi et al. [2007] describes a caching mechanism that takes advantage of the periods when the user is looking at and processing on screen graphics to precompute expensive operations that the user may wish to do next.

The Augsburg impressionist software (Manet, Unwin et al. [1996]; Cassatt, Winkler [2000]; Mondrian, Theus [2003]; Klimt, Urbanek [2004]; Gauguin, Gribov [2007]) has focused on new types of graphics and improved models of interaction, not on the data pipeline. While there are no published accounts of the data model that underlies these applications, personal communication with some the authors and inspection of the code has revealed some details. In particular, little data manipulation takes place, apart from the computation of multidimensional contin-

gency tables necessary for categorical graphics. Wilhelm [2005], another Augsburg graduate, discusses general issues regarding interactive graphics, but does not deal with computational considerations.

It is even harder to find out how commercial software works. Conversations with experienced users and inspection of documentation indicates that Datadesk [Velleman, 1992] has a powerful and speedy pipeline, but the structure is unknown. SPSS [SPSS Inc, 2007] has implemented interactive extensions to the grammar of graphics, and uses the fixed pipeline outlined in [Wilkinson, 2005].

Outside of statistical graphics, there has also been work on these topics in the information visualization community. The prefuse framework [Heer et al., 2005] is a Java framework for designing web-based interactions, based on the “information visualization reference model” Heer and Agrawala [2006]. It is not very well documented, and supports very little data manipulation. Improvise [Weaver, 2006, 2007] provides a more flexible framework than outlined in this paper. The general model is a directed graph, and it the responsibility of the user to prevent cycles from forming. Even less is known about commercial products which grew out of the work from this community, for example, Spotfire [Jog and Shneiderman, 1994] and Tableau [Stolte et al., 2002].

### 3 What is a pipeline?

With the pipeline we want to make explicit the process of taking raw data, visualizing it and then updating the data based on interactions with that visualization. A pipeline takes the raw data that we want to display and after a series of steps produces explicit instructions on how to render the end product. In particular, from a vector of data coordinates, measured in any system of units, it generates a vector of screen coordinates, in pixels. This pipeline has to be present in every graphics program—all software must perform these transformations—although it may not be explicit. We argue that making it explicit has important advantages: it helps with computation and implementation, and it makes it possible to compare different interactive systems on a deeper and more objective level.

For this paper, we will consider the original data matrix to be augmented with extra columns which describe additional details about the observations. For example, extra columns give the color, shape and size of the glyph that represents that point. The data matrix is also augmented with column level information such as the range of each variable. This augmentation is largely a computational and notational convenience.

A pipeline is composed of pipes, or stages, each of which encapsulates a task. At a high level, a pipeline stage is simply a function and its inverse, with a set of parameters that controls its operation. At a more detailed level, a pipeline stage is an actor which (1) responds to events, (2) generates and forwards events, (3) receives data, (4) applies some function to the data (selecting rows or columns, imputing values, applying a linear transformation, etc), and (5) passes the data on. Depending on what event triggers the change these changes either cascade up or down the pipeline.

A stage needs the inverse function because the pipeline also coordinates the flow of information in the other direction: from plot to data. For example, when a user interacts with a point on screen, we must be able to map backwards along the pipeline to figure out how to adjust that point in the original data. One type of interaction that is particularly useful is brushing, or selection. Brushing changes the color (or size, or shape) of an observation. Because the change flows back to affect the original data set, it also flows downstream to all other plots so that we get linked brushing. Coordinating this flow of data is not simple, and the next section discusses some of the possible approaches for coordinating the pipeline.

## 4 Coordinating the pipeline

The simplest pipeline looks something like Figure 1. We have our augmented data, pipeline stages to transform and standardize the data, and then the final plot. Changes can occur for two reasons: a change to the data, or a change to the parameters of a stage. For example:

- The original data may change if we have streaming data, or if we are creating an animation from within another program.

- Changing which variables are transformed, or the type of transformation, is a change to the parameters of the stage and requires updating everything downstream.
- Similarly, changing the type of standardization also requires downstream updates.
- Finally, interaction with the plot, such as brushing, requires updates to the original data.

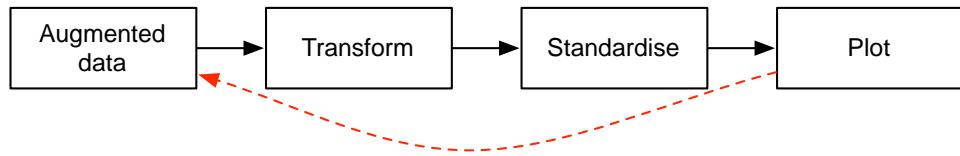


Figure 1 Simple pipeline. Dashed red line indicates data flowing from the plot to the original data

The possibilities for change in this very simple example are already quite large, but we can make a simple rule which ensures everything remains consistent: changes to the parameters of a stage update all stages downstream, and changes from interaction with the plot flow upstream. Changes need to flow up one stage at a time so that any alterations can be reversed before updating the original data. This shown in figure 2

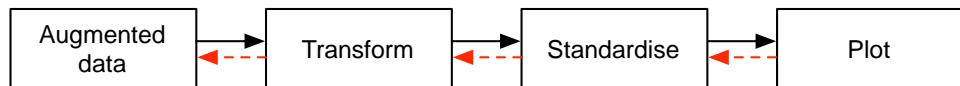


Figure 2 Simple pipeline with reverse pipeline. Dashed red line indicates data flowing from the plot to the original data

Unfortunately life is not that simple, and in any non-trivial application, we will have multiple plots, as in Figure 3. In this case, creating a consistent system of updates is much more complicated. We describe three possibilities below: plots update other plots, a central commander orchestrates the system, or updates follow the pipeline. When a plot is the current target of interaction, we call it the *active plot*.

Plot-to-plot communication, figure 4, relies on the active plot knowing how to update the other plots. At the minimum, it must know what other plots exist, and then tell each of them what has changed. This design is simple, but has considerable drawbacks: each plot must

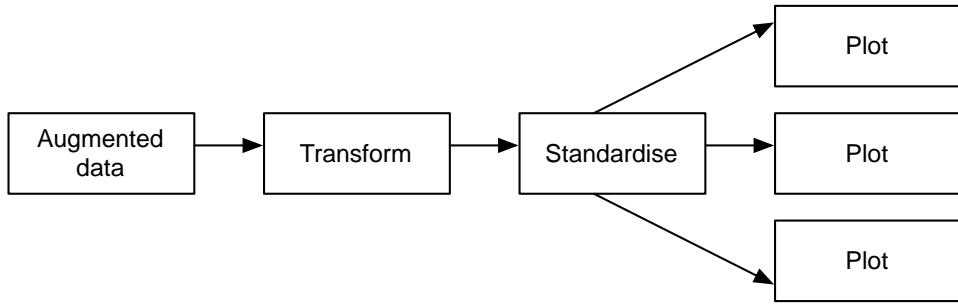


Figure 3 Pipeline with multiple plots.

know a lot about the other plots, and more importantly the original data is never updated. This makes this design increasingly complicated as the number of plot types increases, and we can never inspect the original data to see what changes have been made. Interactions between changing the plot and changing the parameters of the pipeline stages will be complicated.

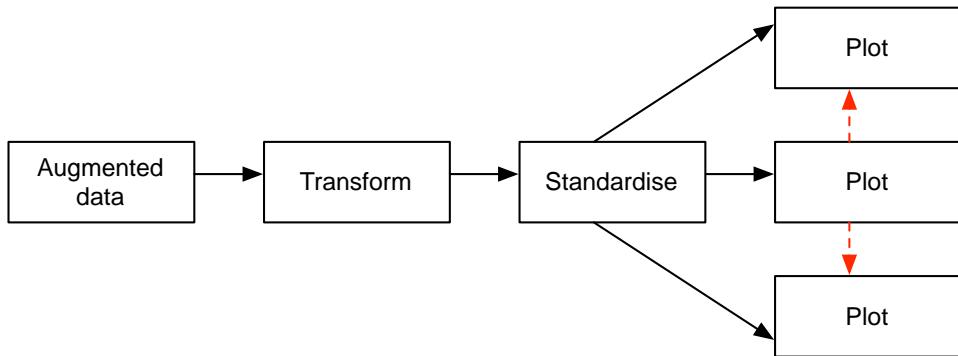


Figure 4 Pipeline with plot-to-plot updates.

A better design uses a central commander to take the burden off the plot objects, as shown in figure 5. This is a common object oriented design pattern [Gamma et al., 1995], and used in many software packages [Apache Software Foundation, 2007]. It is useful because it is centralized, with a single object encapsulating all the control paths. The plumbing exists in one place and a single component “commands” the pipeline stages like a puppeteer: in essence, the commander *is* the pipeline. However, this design suffers a similar problem to the plot-to-plot design: the commander object needs to know how to coordinate every component, and as the number of components increases there may be a corresponding increase in complexity.

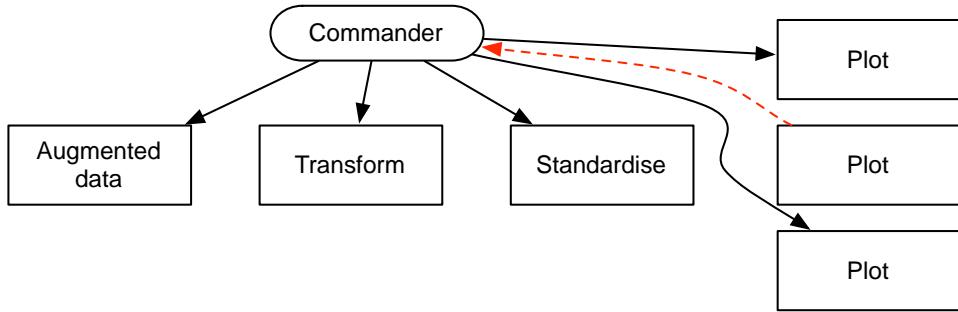


Figure 5 Pipeline with central commander to control flow of updates..

In the final design, and the one that we think is most appropriate, events follow the same path as the pipeline. There is a two stage approach where first the data changes flows right back to the original data, and then the events flow back out to the plots. There are some complications in this design to ensure that we don't update things twice or get caught in infinite loops, but these are resolvable. This design is demonstrated in figure 6.

We favor our design because it is self-contained. The pipeline emerges as the stages are connected, just as in a real plumbing system. Because the operation of each stage is tightly constrained to be orthogonal to all others, we can treat new stages as black boxes and only need worry about how they are connected together. Just as plumbers have adopted standard pipe size and set of connectors, we have adopted a common interface for pipeline stages.

## 5 Pipes vs plumbing

It is important to reinforce the difference between pipes and the pipeline. Pipes, or stages, are the simple orthogonal components that when plumbed together form a pipeline. Some graphics software provides the pipes, while other software provides the plumbing. Software that provides only pipes is generally used as a component of other programs: it does not offer a data analysis environment. Software that offers a pipeline usually provides a tailored environment for data analysis, but if you want to extend or modify the underlying process you will need to call a plumber (a developer).

Table 1 provides a list of some interactive graphics software with an explicit notion of a

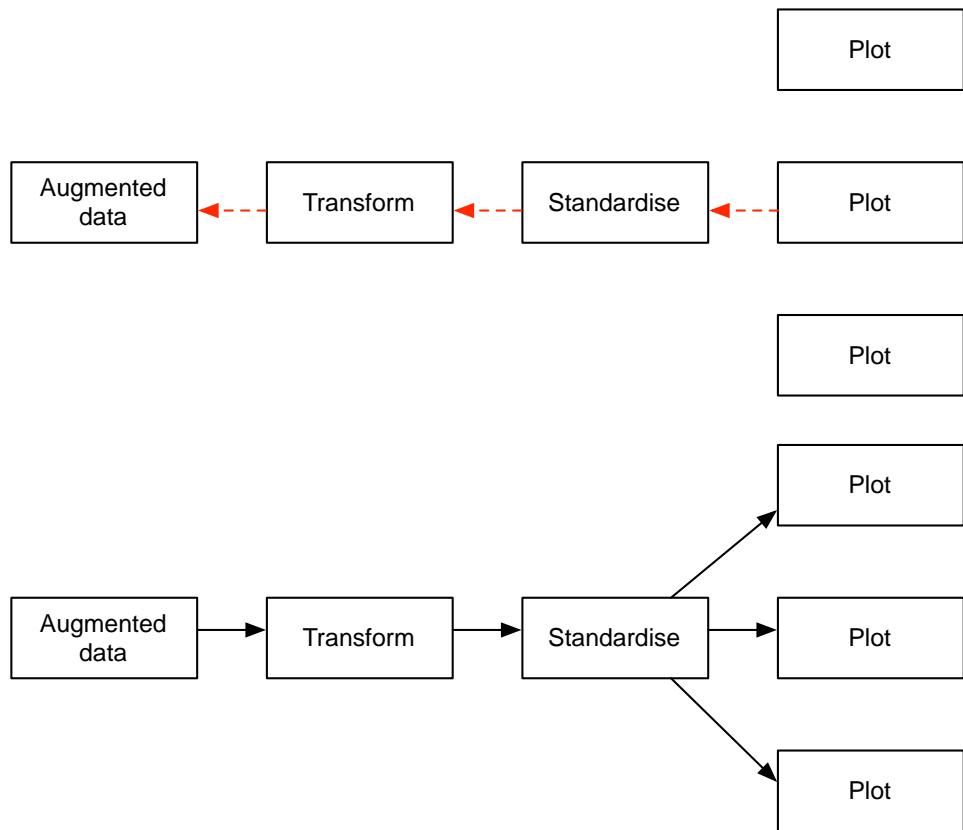


Figure 6 Pipeline with two-stage event model, where events follow the pipeline.

pipeline. All the interactive graphics software with an explicit pipeline that we are aware of comes from DataViewer [Buja et al., 1986] ancestry.

Package	Type	Year released	Reference
DataViewer	pipeline	1986	Buja et al. [1986]
XGobi	pipeline	1992	Swayne et al. [1991]
Orca	pipes	1999	Sutherland et al. [2000]
GGobi	pipeline	2000	Swayne et al. [2003a]
GGobi 3	pipes, and pipeline	2007	

Table 1 Interactive graphics software with an explicit pipeline.

## 6 Motivation and implementation

Over the past two years, we have implemented a prototype of the communication and pipeline model in GGobi. Previously, although the pipeline was an important part of GGobi, the code for the pipeline was not cleanly separate from the rest of the implementation, and changing the plumbing of the pipeline was near impossible.

The motivation for this change is simple: we want to make GGobi more flexible and customizable. Figure 7 provides an example of what we would like to be able to do with GGobi. Currently, the subset stage occurs immediately after the raw data. This means that every plot must display the same data. We would like to be able to shift this subset stage to a place much later in the pipeline, so that each plot can display different subsets of the original data. This will enable us to add features such as conditioning and shingles.

This is one advantage of the new pipeline: it makes it easy to recode the plumbing of GGobi because concerns are cleanly decoupled, and so makes it easy for developers to change how GGobi works. However, the new pipeline is more adaptable yet: it is possible to rearrange the pipeline at run-time. If this were supported by a user interface to GGobi, the user could generate custom types of interactive visualizations. We have only just begun to explore the possibilities of this, but one simple consequence is that it becomes very easy to experiment with interactive graphics from R or some other environment capable of embedding GGobi.

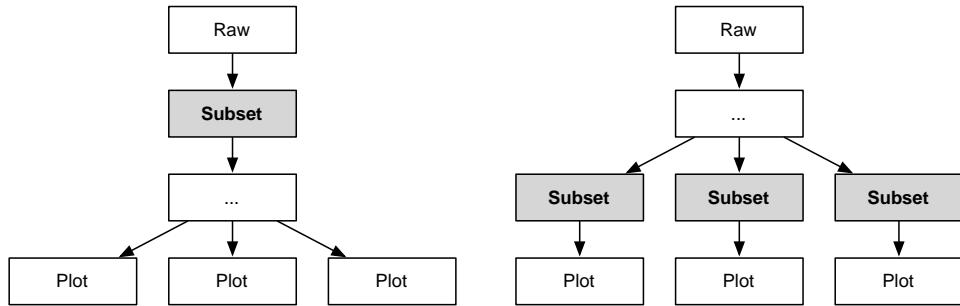


Figure 7 What we want to be able to do with GGobi. Currently subsetting occurs as the first step in the pipeline, so that all plots show the same data, but moving it to a later point will allow different plots to show different subsets of the data.

Two other important aims are adding area plots (eg. mosaic plots) to GGobi and creating more complex displays. The original pipeline was created for scatterplots, and area plots require very different treatment. We need a better mechanism for aggregation; currently it's handled as a sort of diversion from the pipeline. One more complex display we would like to experiment with is a 2D tour with the corresponding 1D tours displayed on the margins. This is not possible in the current framework.

The biggest concern with the new implementation is speed: will the new, more, general framework decrease the overall speed of the software? To date, there have been some slowdowns, but we are hopeful that the developer time saved with a cleaner, more flexible design, can be spent profiling and improving performance so that the net effect is positive. There are also some things that we do now that could be done much more cleanly and efficiently: subsetting simply for computational efficiency when working with large data is rather messy now, and we still store the full data longer than necessary.

## 7 Conclusions

This paper revisits the data pipeline for interactive graphics in light of recent work with GGobi. The major difficulty for any interactive graphics software is co-ordinating changes and ensuring that plots are updating in a timely and consistent manner. We discussed three possible designs for controlling the flow of updates, and suggest that the most appropriate

design occurs when updates follow the same path as the pipeline. Implementation of these ideas in GGobi are still at an early stage, but development so far shows promise, and we have many ideas on how to use the new power and flexibility.

## AN INTRODUCTION TO RGGOBI

A paper published in R News

Hadley Wickham, Michael Lawrence, Duncan Temple Lang, Deborah F Swayne

### 1 Introduction

The rggobi [Temple Lang, 2001b] package provides a command-line interface to GGobi, an interactive and dynamic graphics package. Rggobi complements GGobi’s graphical user interface by enabling fluid transitions between analysis and exploration and by automating common tasks. It builds on the first version of rggobi to provide a more robust and user-friendly interface. In this article, we show how to use GGobi and offer some examples of the insights that can be gained by using a combination of analysis and visualization.

This article assumes some familiarity with GGobi. A great deal of documentation, both introductory and advanced, is available on the GGobi web site, <http://www.ggobi.org>; newcomers to GGobi might find the demos at [ggobi.org/docs](http://ggobi.org/docs) especially helpful. The software is there as well, both source and executables for several platforms. Once you have installed GGobi, you can install rggobi and its dependencies using `install.packages("rggobi", dep=T)`.

This article introduces the three main components of rggobi, with examples of their use in common tasks:

- Getting data into and out of GGobi.
- Modifying observation-level attributes (“automatic brushing”).
- Basic plot control.

We will also discuss some advanced techniques such as creating animations with GGobi, the use of edges, and analyzing longitudinal data. Finally, a case study shows how to use rggobi to create a visualization for a statistical algorithm: manova.

## 2 Data

Getting data from R into GGobi is easy: `g <- ggobi(mtcars)`. This creates a `GGobi` object called `g`. Getting data out isn't much harder: Just index that `GGobi` object by position (`g[[1]]`) or by name (`g[["mtcars"]]` or `g$mtcars`). These return `GGobiData` objects which are linked to the data in GGobi. They act just like regular data frames, except that changes are synchronized with the data in the corresponding GGobi. You can get a static copy of the data using `as.data.frame`.

Once you have your data in GGobi, it's easy to do something that was hard before: find multivariate outliers. It is customary to look at uni- or bivariate plots to look for uni- or bivariate outliers, but higher-dimensional outliers may go unnoticed. Looking for these outliers is easy to do with the tour. Open your data with GGobi, change to the tour view, and select all the variables. Watch the tour and look for points that are far away or move differently from the others—these are outliers.

Adding more data sets to an open GGobi is also easy: `g$mtcars2 <- mtcars` will add another data set named “`mtcars2`”. You can load any file type that GGobi recognizes by passing the path to that file. In conjunction with `ggobi_find_file`, which locates files in the GGobi installation directory, this makes it easy to load GGobi sample data. This example loads the olive oils data set included with GGobi:

```
ggobi(ggobi_find_file("data", "olive.csv"))
```

## 3 Modifying observation-level attributes, or “automatic brushing”

Brushing is typically thought of as an operation performed in a graphical user interface. In GGobi, it refers to changing the color and symbol (glyph type and size) of points. It is

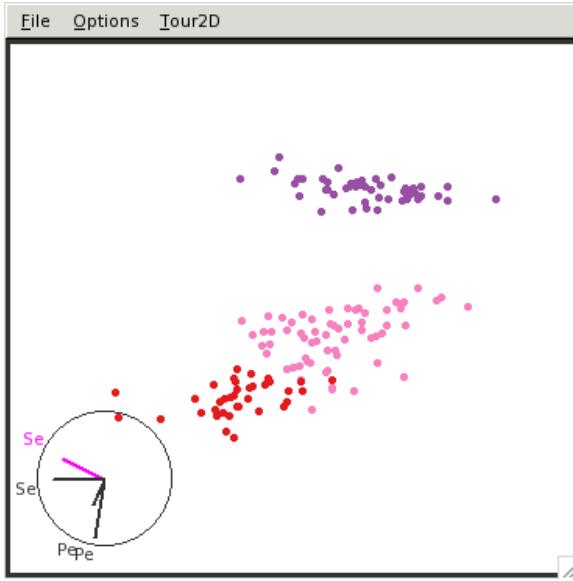
typically performed in a linked environment, in which changes propagate to every plot in which the brushed observations are displayed. in GGobi, brushing includes shadowing, where points sit in the background and have less visual impact, and exclusion, where points are completely excluded from the plot. Using rggobi, brushing can be performed from the command line; we think of this as “automatic brushing.” The following functions are available:

- change glyph color with `glyph_color` (or `glyph_color`)
- change glyph size with `glyph_size`
- change glyph type with `glyph_type`
- shadow and unshadow points with `shadowed`
- exclude and include points with `excluded`

Each of these functions can be used to get or set the current values for the specified `GGobiData`. The “getters” are useful for retrieving information that you have created while brushing in GGobi, and the “setters” can be used to change the appearance of points based on model information, or to create animations. They can also be used to store, and then later recreate, the results of a complicated sequence of brushing steps.

This example demonstrates the use of `glyph_color` to show the results of clustering the infamous Iris data using hierarchical clustering. Using GGobi allows us to investigate the clustering in the original dimensions of the data. The graphic shows a single projection from the grand tour.

```
g <- ggobi(iris)
clustering <- hclust(dist(iris[,1:4]),
method="average")
glyph_color(g[1]) <- cuttree(clustering, 3)
```



Another function, `selected`, returns a logical vector indicating whether each point is currently enclosed by the brush. This could be used to further explore interesting or unusual points.

## 4 Displays

A `GGobiDisplay` represents a window containing one or more related plots. With `rggobi` you can create new displays, change the projection of an existing plot, set the mode which determines the interactions available in a display, or select a different set of variables to plot.

To retrieve a list of displays, use the `displays` function. To create a new display, use the `display` method of a `GGobiData` object. You can specify the plot type (the default is a bivariate scatterplot, called “XY Plot”) and variables to include. For example:

```
g <- ggobi(mtcars)

display(g[1], vars=list(X=4, Y=5))

display(g[1], vars=list(X="drat", Y="hp"))

display(g[1], "Parallel Coordinates Display")

display(g[1], "2D Tour")
```

The following display types are available in GGobi (all are described in the manual, available from [ggobi.org/docs](http://ggobi.org/docs)):

Name	Variables
1D Plot	1 X
XY Plot	1 X, 1 Y
1D Tour	$n$ X
Rotation	1 X, 1 Y, 1 Z
2D Tour	$n$ X
2x1D Tour	$n$ X, $n$ Y
Scatterplot Matrix	$n$ X
Parallel Coordinates Display	$n$ X
Time Series	1 X, $n$ Y
Barchart	1 X

After creating a plot you can get and set the displayed variables using the `variable` and `variable<-` methods. Because of the range of plot types in GGobi, variables should be specified as a list of one or more named vectors. All displays require an X vector, and some require Y and even Z vectors, as specified in the above table.

```
g <- ggobi(mtcars)
d <- display(g[1],
  "Parallel Coordinates Display")
variables(d)
variables(d) <- list(X=8:6)
variables(d) <- list(X=8:1)
variables(d)
```

A function which saves the contents of a GGobi display to a file on disk, is called `ggobi_display_save_pict`. This is what we used to create the images in this document. This creates an exact (raster) copy of the GGobi display. If you want to create publication quality graphics from GGobi, have a

look at the `DescribeDisplay` plugin and package at <http://www.ggobi.org/describe-display>. These create R versions of your GGobi plots.

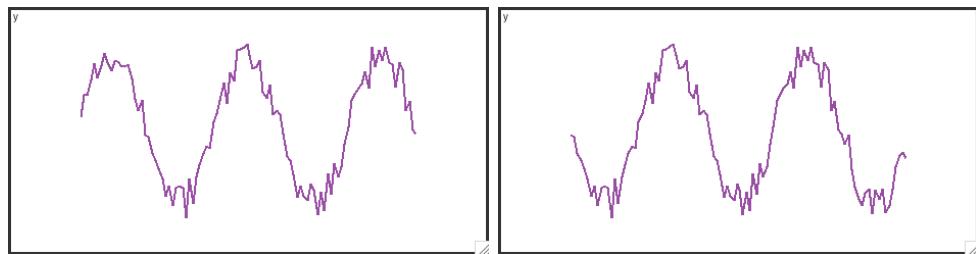
To support the construction of custom interactive graphics applications, rggobi enables the embedding of GGobi displays in graphical user interfaces (GUIs) based on the RGtk2 package. If `embed = TRUE` is passed to the `display` method, the display is not immediately shown on the screen but is returned to R as a `G GtkWidget` object suitable for use with RGtk2. Multiple displays of different types may be combined with other widgets to form a cohesive GUI designed for a particular data analysis task.

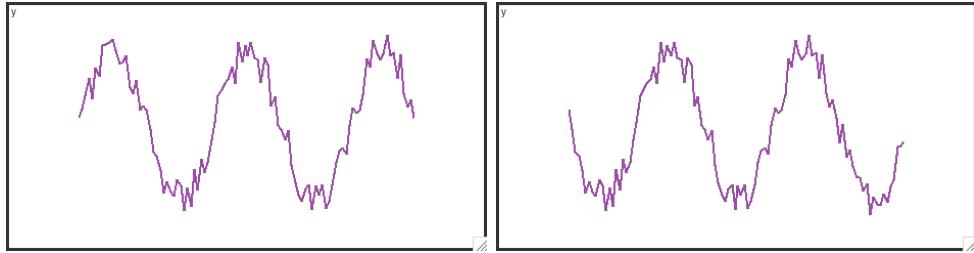
## 5 Animation

Any changes that you make to the `GGobiData` objects are updated in GGobi immediately, so you can easily create animations. This example scrolls through a long time series:

```
df <- data.frame(
  x=1:2000,
  y=sin(1:2000 * pi/20) + runif(2000, max=0.5)
)
g <- ggobi_longitudinal(df[1:100, ])

df_g <- g[1]
for(i in 1:1901) {
  df_g[, 2] <- df[i:(i + 99), 2]
}
```





## 6 Edge data

In GGobi, an edge data set is treated as a special type of dataset in which a record describes an edge – which may still be associated with an n-tuple of data. They can be used to represent many different types of data, such as distances between observations, social relationships, or biological pathways.

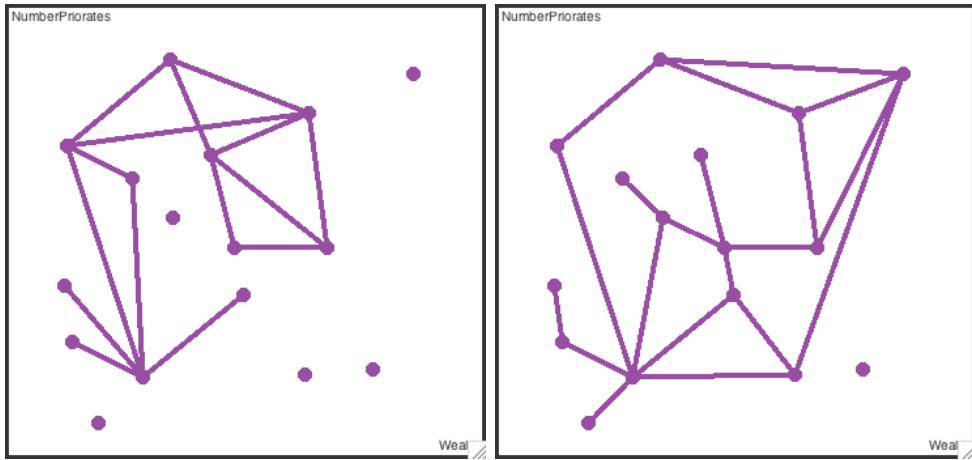
In this example we explore marital and business relationships between Florentine families in the 15th century. The data comes from the **SNAdata** (social networking analysis) package [Scholtens, 2007], in the format provided by the **graph** package [Gentleman et al., 2007].

```
library(graph)
library(SNAdata)

data(business, marital, florentineAttrs)

g <- ggobi(florentineAttrs)
edges(g) <- business
edges(g) <- marital
```

This example has two sets of edges because some pairs of families have marital relationships but not business relationships, and vice versa. We can use the edges menu in GGobi to change between the two edge sets and compare the relationship patterns they reveal.



How is this stored in GGobi? An edge dataset records the names of the source and destination observations for each edge. You can convert a regular dataset into an edge dataset with the `edges` function. This takes a matrix with two columns, source and destination names, with a row for each edge observation. Typically, you will need to add a new data frame with number of rows equal to the number of edges you want to add.

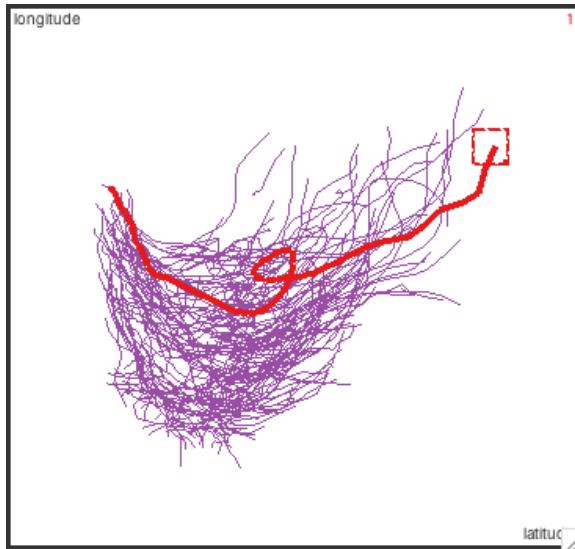
## 6.1 Longitudinal data

A special case of data with edges is time series or longitudinal data, in which observations adjacent in time are connected with a line. Rggobi provides a convenient function for creating edge sets for longitudinal data, `ggobi_longitudinal`, that links observations in sequential time order.

This example uses the `stormtracks` data included in rggobi. The first argument is the dataset, the second is the variable specifying the time component, and the third is the variable that distinguishes the observations.

```
ggobi_longitudinal(stormtracks, seasday, id)
```

For regular time series data (already in order, with no grouping variables), just use `ggobi_longitudinal` with no other arguments.



## 7 Case study

This case study explores using rggobi to add model information to data; here will add confidence ellipsoids around the means so we can perform a graphical manova.

The first (and most complicated) step is to generate the confidence ellipsoids. The `ellipse` function does this. First we generate random points on the surface of sphere by drawing `npoints` from a random normal distribution and standardizing each dimension. This sphere is then skewed to match the desired variance-covariance matrix, and its size adjusted to give the appropriate `cl`-level confidence ellipsoid. Finally, the ellipsoid is translated to match the column locations.

```
conf.ellipse <- function(data, npoints=1000,
  cl=0.95, mean=colMeans(data), cov=var(data),
  n=nrow(data)
) {
  norm.vec <- function(x) x / sqrt(sum(x^2))

  p <- length(mean)
  ev <- eigen(cov)
```

```

normsamp <- matrix(rnorm(npoints*p), ncol=p)
sphere <- t(apply(normsamp, 1, norm.vec))

ellipse <- sphere %*% diag(sqrt(ev$values))
%*% t(ev$vectors)

conf.region <- ellipse * sqrt(p * (n-1) *
qf(cl, p, n-p) / (n * (n-p)))
if (!missing(data))
  colnames(ellipse) <- colnames(data)

conf.region + rep(mean, each=npoints)
}

```

This function can be with a data matrix, or with the sufficient statistics (mean, covariance matrix, and number of points). We can look at the output with ggobi:

```

ggobi(conf.ellipse(mean=c(0,0), cov=diag(2)))

cv <- matrix(c(1,0.15,0.25,1), ncol=2)
ggobi(conf.ellipse(mean=c(1,2), cov=cv))

mean <- c(0,0,1,2)
ggobi(conf.ellipse(mean=mean, cov=diag(4)))

```

In the next step, we will need to take the original data and supplement it with the generated ellipsoid:

```

manovaci <- function(data, cl=0.95) {
  dm <- data.matrix(data)
  ellipse <- as.data.frame(

```

```

conf.ellipse(dm, n=1000, cl=cl)
)

both <- rbind(data, ellipse)
both$SIM <- factor(
  rep(c(FALSE, TRUE), c(nrow(data), 1000))
)

both
}

ggobi(manovaci(matrix(rnorm(30), ncol=3)))

```

Finally, we create a method to break a dataset into groups based on a categorical variable and compute the mean confidence ellipsoid for each group. We then use the automatic brushing functions to make the ellipsoid distinct and to paint each group a different color. Here we use 68% confidence ellipsoids so that non-overlapping ellipsoids are likely to have significantly different means.

```

ggobi_manova <- function(data, catvar, cl=0.68) {
  each <- split(data, catvar)
  cis <- lapply(each, manovaci, cl=cl)

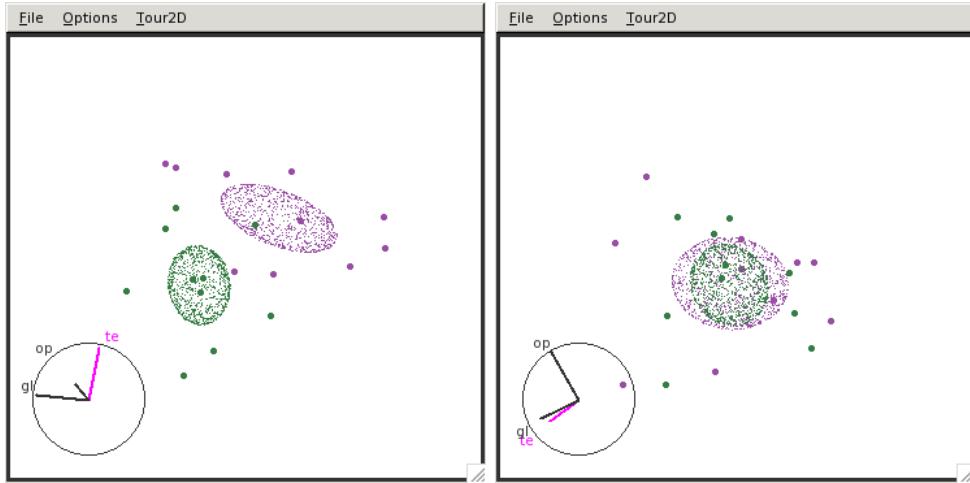
  df <- as.data.frame(do.call(rbind, cis))
  df$var <- factor(rep(
    names(cis), sapply(cis, nrow)
  ))

  g <- ggobi(df)
  glyph_type(g[1]) <- c(6,1)[df$SIM]
  glyph_color(g[1]) <- df$var
}

```

```
invisible(g)
}
```

These images show a graphical manova. You can see that in some projections the means overlap, but in others they do not.



## 8 Conclusion

GGobi is designed for data exploration, and its integration with R through rggobi allows a seamless workflow between analysis and exploration. Much of the potential of rggobi has yet to be realized, but some ideas are demonstrated in the `classifly` package [Wickham, 2007a], which visualizes high-dimensional classification boundaries. We are also keen to hear about your work—if you develop a package using rggobi please let us know so we can highlight your work on the GGobi homepage.

We are currently working on the infrastructure behind GGobi and rggobi to allow greater control from within R. The next generation of rggobi will offer a direct low-level binding to the public interface of every GGobi module. This will coexist with the high-level interface presented in this paper. We are also working on consistently generating events in GGobi so that you will be able respond to events of interest from your R code. Together with the RGtk2 package [Lawrence and Temple Lang, 2007], this should allow the development of custom interactive graphics applications for specific tasks, written purely with high-level R code.

### Acknowledgments

This work was supported by National Science Foundation grant DMS0706949. The ellipse example is taken, with permission, from Professor Di Cook's notes for multivariate analysis.

## EXTENDING THE GGOBI PIPELINE FROM R

A paper to appear in Computational Statistics

Michael Lawrence, Hadley Wickham, Dianne Cook, Heike Hofmann, Deborah F. Swayne

### **Abstract**

GGobi is a software tool for multivariate interactive graphics. At the core of GGobi is a data pipeline that incrementally transforms data through a series of stages into a plot and maps user interaction back to the data. The GGobi pipeline is extensible and mutable at run-time. The rggobi package, an interface from the R language to GGobi, supports the extension of the GGobi pipeline from R. Out of recognition of the utility of R for working with data, rggobi is designed to support rapid prototyping of customized interactive data visualizations based on GGobi. The large size of the GGobi API has motivated the use of the RGtk2 code generation system to create the low-level interface between R and GGobi. The software is demonstrated through an application to interactive network visualization.

### **1 Introduction**

The primary purpose of computer software is produce an output in response to some input. In the realm of computer graphics, this involves transforming graphical information, such as pixels, vectors or other primitives, into something visible on an output device, often a computer monitor. Statistical graphics software also performs these functions, with the addition of an initial step that converts a dataset, containing variables describing a set of observations, into graphical information. This conversion involves a sequence of steps that incrementally

transform the data into a visual representation. The composition of this sequence determines the nature of the statistical visualization.

This sequence is often formally referred to as a data pipeline. When applied to statistical graphics, a data pipeline transforms a dataset, through a series of stages, into a plot [Buja et al., 1988a]. For interactive graphics, the pipeline needs to be dynamic. The user can manipulate the visualization, for example, through interaction with the data in the plot or by modifying the parameters of a pipeline stage. The pipeline must react to these changes and efficiently update the visualization by only executing stages that may have modified input. By operating in reverse, a pipeline is able to map user interaction back to the original data. A branching pipeline allows multiple plots to share pipeline resources. Factoring the common stages into a single trunk minimizes update operations by not repeating them for every plot. This optimization is possible as long as the common stages precede any plot-specific stages. A pipeline design based on a sequence of independent, modular stages facilitates the invention of new types of visualizations and generally makes the software more maintainable. For these reasons, the pipeline is critical for dynamic visualizations like the grand tour, as well as for maintaining the flow of an exploratory analysis.

The data pipeline forms the core of the ancestral interactive graphics packages DataViewer [Buja et al., 1988a] and XGobi [Swayne et al., 1991]. The Orca project [Sutherland et al., 2000] extends the DataViewer / XGobi pipeline to support multiple linked views and data typing. The Orca pipeline implementation is object-oriented and follows many established software design patterns. However, Orca was slow and lacked some of the necessary basic tools for it to be very useful in practice.

The successor to XGobi, GGobi, is an open-source tool for interactive multivariate graphics [Swayne et al., 2003a]. It is capable of several types of displays, such as scatterplots, barcharts and parallel coordinate plots. Each display supports multiple interaction modes, including linked brushing and identification. GGobi also includes a small collection of tools that support exploratory data analysis, like principal component analysis and missing value imputation. It is extensible via plugins. The data pipeline pattern has always been part of the core GGobi

design. The big step from the XGobi pipeline is that the GGobi pipeline supports multiple views. Careful attention was paid to where the pipeline needed to branch into multiple copies of the data to use space efficiently.

We have recently refactored the GGobi pipeline so that it is more object-oriented, like Orca's, and more dynamic, so that a new stage object may be inserted into the pipeline at any point. The new pipeline is mutable at run-time and new types of stages may be defined by extending existing stage classes. This improves the general maintainability of GGobi, and the increased flexibility facilitates experimentation with interactive visualizations, including those specifically tailored to a particular dataset or problem domain.

The rggobi package [Temple Lang, 2001b] is an interface between the R language [R Development Core Team, 2005] and GGobi. There are two major motivations for developing rggobi. First, GGobi is controlled via a graphical user interface (GUI). The GUI is convenient for casual exploration of a dataset, but it becomes cumbersome and restrictive during more sophisticated analyses. Due in part to its data manipulation features and high-level, interactive nature, R has proved to be an effective language for scripting GGobi. The rggobi interface is consistent with R language conventions in order to make it more accessible to R programmers. With rggobi, it is possible to automate tasks that would be tedious or impossible to perform from the GGobi GUI. The second motivation for rggobi is that there are very few numerical methods implemented within GGobi, because the designers did not wish to reinvent the wheel by implementing methods available elsewhere. R fills this void, as it provides access to an ever increasing number of modern numerical, as well as graphical, methods.

The existing rggobi interface to GGobi supports common high-level tasks, such as loading a dataset and coloring points. The design of the interface aims for consistency with R conventions and familiarity to the R user. For example, the GGobi dataset object may be accessed as if it were an R *data.frame*. The subset methods, `$`, `[` and `[[` all behave in the expected way. Replacement methods are used to set properties on objects. For setting the glyph colors, for example, the name of the generic method is `glyph_color<-`. This high-level interface blends with the rest of R and is accessible to the majority of R programmers. However, its control

over GGobi is limited compared to what is possible through the GGobi API.

The current work, described in this paper, augments rggobi with a low-level interface to GGobi that binds directly to the public methods of the underlying GGobi objects and allows the embedding of R within GGobi through the extension of GGobi classes. This functionality relies on improvements in the RGtk2 package [Lawrence and Temple Lang, 2007], a binding from R to GTK+, an open-source library for constructing graphical user interfaces [Krause, 2007]. Through the low-level interface, loading a dataset is not nearly as simple as through the high-level interface. It is necessary to obtain an instance of a dataset factory, invoke the factory on an input source to create the dataset object, and add the dataset to the GGobi session. The advantage of the low-level interface is that it allows, for example, the R programmer to implement a parser for a custom input format and register it for use when opening files from the GGobi GUI. The low-level and high-level interfaces to GGobi are designed to be compatible, so the user may easily switch between levels of granularity during an rggobi session.

The recent work on rggobi and RGtk2 is described in the next section. This is followed by a demonstration, complete with code listings, that applies the new rggobi features to interactive network layout. The paper concludes with a discussion of future directions for rggobi.

## 2 Low-level interface to GGobi

The low-level interface to GGobi permits the R programmer to rearrange the pipeline and define new types of pipeline stages. This is based on its general ability to dispatch methods on GGobi objects and to extend GGobi classes from R.

The objects in GGobi are based on the GObject library, a library for object-oriented programming in C. GObject provides a run-time introspectable type system on top of conventional C structures. Types can have an associated class structure, and classes can inherit from a single parent class. The GObject class is the canonical base class. GObject-derived classes support the registration of introspectable properties, which might be described as encapsulated public instance fields. Another important feature of GObject is support for signals. Client code can register handler functions against signals, and those functions will be invoked when a signal is

emitted. A signal might be emitted, for example, in response to a user action in a GUI. The GObject library forms the basis for the GTK+ library that GGobi uses for constructing its GUI [Krause, 2007]. The R package RGtk2 binds GTK+, as well as much of GObject, to R, so rggobi depends on RGtk2 for interacting with the GGobi GObjects.

Although the redesign of GGobi is far from complete, there are already more than a dozen formal GGobi classes. Their combined number of public methods is nearly 300. The GGobi API is currently highly unstable, and many more classes will be created as the redesign progresses. Thus, the creation and maintenance of manual bindings between R and the GGobi API would require substantial effort and time. In order to avoid this cost, we have adopted an automated approach to binding generation.

Since GGobi and GTK+ are both based on GObject, we are able to use the existing RGtk2 binding generation system to automatically create the low-level GGobi bindings. Briefly, the GGobi header files are scraped using scripts from the PyGTK [Chapman and Kelley, 2000] project to create a high-level description of the API. That description is fed into the RGtk2 binding generator, which outputs a set of C and R source files that interface GGobi with R. For binding the methods of the GGobi classes, the system generates a pair of wrapper functions, one in R and one in C, for each function in the GGobi API. The R wrapper coerces the arguments to the correct type and invokes the C wrapper. The C wrapper converts the R objects to the data structures specified by the GGobi API and invokes the GGobi function on those arguments. The return value is converted to the corresponding R object and returned to R.

The R documentation for the methods and data types of the GGobi API is also autogenerated. The GGobi API is documented by comments in the source code that are converted by the gtk-doc utility [GTK-Doc, 2007] to an XML representation. This information is parsed into R using the XML package [Temple Lang, 2001a] and converted to the Rd format. The conversion process transforms the syntax and, as much as feasible, the semantics of the documentation from C to R.

Rggobi also leverages RGtk2 to support the extension of GGobi classes from R. To extend

a GGobi class, the R programmer must specify the name of the class, the name of the parent class and a class definition. The class definition is a list of R functions that are named according to the GGobi method that they override. To support the overriding of GGobi methods with R functions, the RGtk2 code generator outputs wrappers in C that convert the GGobi data structures to R objects and invoke the R function. The return value from R is converted to a GGobi structure before returning it to GGobi. This is essentially the reverse of the GGobi API function wrappers.

It is possible to define new fields and methods, as well as other aspects of GObjects, in the class definition list. RGtk2 permits encapsulation of the fields and methods at the public, protected and private level. The semantics are roughly equivalent to those of Java. In short, this is achieved by storing fields and methods in R environments within the instance and class GObject C structures. There is an environment for each level of encapsulation. These environments are exposed, as appropriate, as attributes on the R object corresponding to the GObject-derived instance. The public environment is always exposed, and when a method is invoked, the private environment of its class and the protected environments of its ancestor classes are exposed. The GObject property, field and method accessor functions, `[`, `[[` and `$`, respectively, search the exposed environments when looking up a symbol. Information on using this feature is available from the RGtk2 documentation.

This ability to use R to extend GGobi classes is especially useful for creating custom visualizations in GGobi by extending the data pipeline. There is one instance of the pipeline for every dataset loaded into GGobi. Each data pipeline is a set of stage objects that are linked together in sequence. The low-level interface to GGobi allows the R programmer to change the composition and order of the pipeline. This includes the ability to insert stages, including custom stages defined in R, at any point in the pipeline. Taken to the extreme, the entire GGobi pipeline, the core of GGobi, could be redefined in R at run-time.

However, replacing or augmenting native code with code written in a high-level interpreted language like R has a negative impact on performance. It is important that the R programmer only override what is necessary. The modular, orthogonal design of the GGobi pipeline facil-

itates this. A pipeline stage can be inserted or replaced without affecting the performance of the other stages. Many GGobi pipeline stages cache their results, so they only need to query their parent when there are upstream changes in the data. This further minimizes the performance impact of R pipeline stages by reducing the frequency of their execution. The overall performance of the pipeline may be decreased by inserting an R stage, but the visualization will be more responsive than if the entire pipeline had been ported to R.

The next section gives a simple example of customizing the data pipeline to create a new type of GGobi visualization.

### 3 Application to interactive network layout

It is often useful to express the relationships between the records in a dataset as a network. GGobi can display networks by drawing lines between data points in a scatterplot. There is a GGobi plugin that generates static layouts of networks using the graphviz library [Gansner and North, 2000]. While the GGobi scatterplot has many interactive features, the network layout algorithms provided by the plugin are not interactive. Another plugin, ggviz, uses multidimensional scaling to draw a graph incrementally [Swayne et al., 2003b]. The plugin reoptimizes the layout in real time when the user moves a point in the scatterplot. Although the ggviz algorithm is interactive, it does not support user-defined constraints.

We are interested in interactive network layout, because we are developing software that combines the analysis of biological experimental data with biochemical networks. Networks are useful for experimental data analysis, because they describe the relationships between chemical measurements at the system level. This helps biologists form high-level hypotheses and integrates datasets describing different parts of the system. Biochemical pathways are often quite large, and automatic layout algorithms have trouble drawing networks in a style familiar to the biologists performing the analysis. The ultimate goal is to design an algorithm that overcomes these problems by optimizing the layout in a region of interest specified by the user.

A simple prototype of this algorithm is an effective demonstration of the new capabilities

of rggobi. This example integrates GGobi with an incremental graph layout algorithm that is available from R. The layout algorithm is known as IPSep-CoLa [Dwyer and Marriott, 2006] and is linked to R by the rcola package [Lawrence, 2007a] (the algorithm is implemented in the C++ adaptograms library [Dwyer, 2007]). This algorithm performs force-directed layout using techniques from multi-dimensional scaling, subject to separation constraints that enforce an ordering and gap between pairs of nodes.

A new class, extending *GGobiStage*, was written in R and has two functions: (1) It provides two new variables to the data matrix, specifying the X and Y coordinates of the layout, and (2) it adds constraints to layout algorithm so that, in plots with two colors, the points of the first color (red) are always to the left of the points of the other color (pink). This is illustrated by the screenshots in Figure 1. Whenever the user brushes (colors) a record in GGobi, the pipeline stage catches the event and adds a constraint to the layout algorithm to maintain the separation between the points of different colors. As IPSep-CoLa minimizes the layout stress, it passes the current positions to the stage, which in turn passes them down the pipeline, updating the plot and producing an animation.

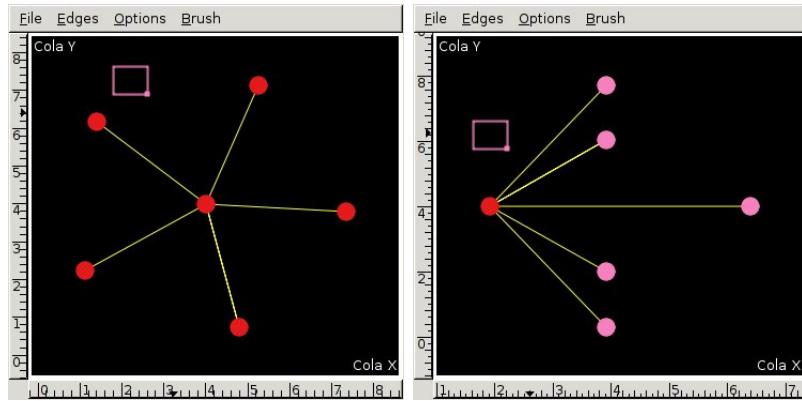


Figure 1 The GGobi scatterplots before (left) and after (right) brushing five points pink in a small network. Note that the pink points are now constrained to lie to right of the red points. This transition is animated on the screen.

We will now step through the actual source code for this example, which is also available on the GGobi beta website [Lawrence et al., 2007b]. The complete documentation for RGtk2 and

`rggobi` is available from the R help system. In particular, the code below involves code using the `$` function as a shortcut for invoking methods on GObjects. `help(RGtkObject)` explains this syntax and `help(GObject)` gives a general introduction to working with GObjects in RGtk2.

To begin our demonstration, we need to create a layout object using `rcola` and generate an initial layout of our Bioconductor graph object `g`, which represents the first step in glycolysis pathway.

```
layout <- new("ConstrainedMajorizationLayout", g)
layout <- run(layout) # compute layout
```

We then add a boundary constraint to the layout named “`_color`” that the stage will bind to the “`_color`” attribute in the pipeline. The boundary constraint will force separation along the X axis depending on the colors of the points.

```
layout <- addConstraintsX(layout,
  new("BoundaryConstraint", 0, name = "_color"))
```

Next, we start GGobi.

```
gg <- ggobi() # start GGobi session
```

Now that GGobi is loaded, we will load two datasets. First, we load the dataset containing the nodes of the network. There are two variables in the dataset, each describing the position of each node in one dimension.

```
# load dataset with initial node positions
positions <- rects(layout) # get intial node positions
d <- ggobi_set_data_frame(as.data.frame(positions[,1:2]),
  name = "layout")[[1]]
```

The second dataset loaded into GGobi consists of the network edges. Every dataset loaded by `rggobi` needs to have at least one variable. We use the edge weights as the variable for the edge dataset, even though the weights are meaningless in this example.

```
# load dataset for edges
weights <- unlist(edgeWeights(graph(layout)))
names(weights) <- NULL
gg["edges"] <- weights
```

The next step tells rggobi to use the graph object to specify the edges on the edge dataset. Every dataset in GGobi can hold an edge for each of its rows. An edge is specified by the record identifiers for its source and destination. In this case, the identifiers for the edges in the edge dataset match those of the records in the node dataset.

```
e <- gg["edges"]
edges(e) <- g
```

Finally, we display the edges in the plot of the nodes. This effectively draws the graph.

```
disp <- displays(gg)[[1]]
edges(disp) <- e # add edges to plot
```

We have successfully displayed a static layout of the graph in GGobi. All of the code presented so far is possible with the original version of rggobi. We now transition to the low-level interface and create our custom stage that we will add to the pipeline. The `gClass` function creates a new GObject class with a name, parent class, and class definition.

```
# get a low-level reference to GGobiData
# (transition to low-level interface)
d <- ggobi_ref(d)

# create new class derived from GGobiStage
gClass("RColaStageLayout", "GGobiStage", list(
```

The class definition is specified in the code block below. It begins by defining a single GObject property that holds the S4 layout object from the rcola package. The property

definition is a *GParamSpec* structure created with the `gParamSpec` function. All property definitions belong in the `.props` element of the class definition list.

```
.props = list(
  gParamSpec( # define layout property
    "R", # type: R (native R object)
    "layout", # name of the property
    "l", # nickname
    "The rcola layout",, # descriptive blurb
    default.value = new("ConstrainedMajorizationLayout")
  )),
)
```

The class overrides two methods from its parent *GGobiStage*. First, it overrides `get_raw_value()` in order to provide the layout coordinates for the two variables (Cola X and Cola Y) that it defines.

```
# Overrides for methods in GGobiStage
GGobiStage = list(
  get_raw_value = function(self, i, j)
  { # provide layout coordinates
    layout <- self["layout"] # get layout property
    p_cols <- self["parent"]$getNcols()
    if (j >= p_cols)
      rects(layout)[i+1, j+1-p_cols]
    else parentHandler("getRawValue", self, i, j)
  },
)
```

The layout must be updated every time a point is brushed to a different color, so the class also overrides `process_incoming()`, allowing it to catch changes to the color from its parent *stage* (the stage preceding this one in the pipeline, not to be confused with parent *class*).

```

process_incoming = function(self, msg)
{
  # override processing of message from parent stage
  layout <- self[["layout"]]

  # get all changed/added col indices from message
  cols <- msg$getChangedCols()

  if (msg$getNAddedCols())

    cols <- c(cols, 1:msg$getNAddedCols() +
      self$getNcols() - 1)

  msg$setNcols(msg$getNcols() + 2) # pad message

  # ask super class to do the boring work
  self$parentClass()$processIncoming(self, msg)

  # update the constraints based on changes
  col_names <- sapply(cols, self$getColName)

  # update_constraints() is omitted for brevity
  layout@constraintsX <- update_constraints(
    layout@constraintsX, self, col_names)
  layout@constraintsY <- update_constraints(
    layout@constraintsY, self, col_names)

  # recalculate layout and store
  self[["layout"]] <- run(layout)
}

),

```

The final part of the class definition list overrides the `set_property()` method on the base `GObject` class. This method is invoked whenever a property is set on the object. For simplicity, we assume here that the only property set is the parent stage. When the parent stage is set, our stage adds two columns on to the data matrix and sends the changes down the pipeline. This makes it possible to plot the two layout variables, Cola X and Cola Y, defined by our stage.

```

GObject = list( # called when parent set
  set_property = function(self, id, value, pspec)
  { # implicitly add our X and Y columns
    p_cols <- self["parent"]$getNcols()
    self$colsAdded(2) # 2 columns have been added
    self$flushChangesHere() # allocate columns
    self$setColName(p_cols, "Cola X") # set col names
    self$setColName(p_cols+1, "Cola Y")
    # we have overridden set_property(), so
    # we need to store the property ourselves
    assignProp(self, pspec, value)
  }
)
))

```

We have now defined our custom GGobi pipeline stage. This is far from the canonical way to implement a pipeline stage, but it has the benefit of simplicity for this demonstration. It remains to create an instance of the class and to insert the instance into the pipeline. In this case, we insert the new stage right before the subset stage, which is normally the first stage after the raw data. This means that the subset stage could use the variables generated by our stage as part of the subsetting process.

```

child <- d$find("ggobi-main-subset") # find subset stage by id
# create the stage instance
# using the parent of the subset stage as its parent
stage <- gObject("RColaStageLayout", parent=child$getParent(),
  layout = layout, name = "rcola-layout")
# set our stage as the parent of the subset stage
child["parent"] <- stage

```

Now that our stage is live, we register a callback against the rcola layout object so that we can update the drawing as the IPSep-CoLa algorithm minimizes the stress.

```
# register rcola observer callback
layout <- stage[["layout"]]
# stage_observer omitted for brevity
observer(layout) <- stage_observer
stage[["layout"]] <- layout
```

Finally, we display the layout in GGobi and activate its brush.

```
# show our layout and activate brush
variables(disp) <- list(X = 18, Y = 19)
imode(disp) <- "Brush"
```

The user may now brush a node in the network to require it to be positioned to the right of the other points. This constraint is based on the Boolean condition of whether a point is brushed. Examples of more complex constraints include aligning a set of nodes in the X or Y dimension and positioning nodes on the perimeter of a two dimensional shape, like a circle. These constraints may improve the overall clarity of the visualization and help the analyst find interesting structures in the network topology.

As we aim to use this network visualization as part of a larger analysis of biological experimental data, we are interested in deriving constraints from variables describing the nodes. For example, categorical variables could be mapped to a set of boundary constraints, each identical in form to the left/right constraint demonstrated in the example, that segregates the nodes in one dimension according to their value for the variable. Similarly, constraints could order a set of nodes along a dimension according to their order on a quantitative variable. The mapping of data to constraints integrates the data into the layout of the network. Besides position, the visual attributes of the nodes are unaffected and may be used to communicate other information. Moreover, there is no requirement for complex visual annotation schemes that may complicate the drawing.

Another extension to this example would be to link the brush to a custom attribute, instead of color, that indicates the constraint applied to each node. A custom RGtk2-based GUI could be constructed for control of the “constraint brush”. This outlines how one might use rggobi and RGtk2 to incrementally construct a full-fledged application from an initial prototype.

#### 4 Conclusion

Rggobi now provides a fundamental low-level interface to GGobi that, with some work, allows prototyping of interactive visualizations. This complements the existing high-level interface that allows the user to access the data structures in GGobi and set plot parameters such as color.

The rggobi package may not be the only means for customizing interactive visualizations from R. It should also be possible to extend the Java-based iplots package [Urbanek and Theus, 2003] using recently added functionality in the rJava package [Urbanek, 2006] for implementing Java classes in R. However, it is not clear how feasible this would be, given that the iplots software does not appear to be explicitly based on a modular pipeline design.

Future work will focus on convenience for the user. This includes providing high-level wrappers for common tasks such as reordering stages and creating utility stage classes that might, for example, augment the pipeline with user-provided variables, like the layout coordinates in our example. Eventually, this work will evolve into a language for interactive and dynamic graphics.

# RSBML: AN R PACKAGE FOR IMPORTING SBML DOCUMENTS USING LIBSBML

A paper to be submitted to a Bioinformatics journal

Michael Lawrence

## Abstract

Rsbml is an R package that leverages libsbml to read, write, and semantically validate documents formatted according to the Systems Biology Markup Language (SBML) specification, including its layout extension. SBML documents may be imported to R either as an S4 object conforming to the SBML object model or as a Bioconductor graph object. Rsbml is an R package available from [CRAN](#) and [Bioconductor](#). A quick-start vignette is included with the package.

## 1 Introduction

There is a need to integrate biological network models with experimental data analysis. Networks relate measurements from biological experiments to biological systems. This supports the integration of datasets and helps biologists for system-level hypotheses. Experimental data is also useful for validating network simulations. The R platform [R Development Core Team, 2005] has strong support for experimental data analysis. We aim to enable R to access biological network models.

The Systems Biology Markup Language (SBML) is a widely accepted, XML-based standard for representing models of biochemical reaction networks [Hucka et al., 2003]. The fundamental

components of an SBML model are species and reactions. A species is a participant in the model and is normally bound to a reaction, as either a reactant, product, or modifier, by a species reference. Each species belongs to one of a set of defined compartments. The SBML standard also allows for the specification of quantitative behavior on top of the qualitative foundation. Quantitative components include unit definitions, rate laws, and kinetic parameters. Mathematical expressions are formatted as MathML [MathML, 2006]. Application-specific data may be encoded in RDF and embedded into annotation elements.

The most mature library for parsing SBML is libsbml [Bornstein, 2006], which is easy to access from various languages, since it is written in C. Libsbml is able to parse the entire SBML specification and validates the semantic and syntactic correctness of models. It also converts MathML elements into textual representations. Although not part of the SBML specification, libsbml parses descriptions of model layouts, as specified by the SBML layout extension from the European Media Lab [Gauges et al., 2006].

There is an existing package for reading SBML into R, named SBMLR, which does not take advantage of libsbml and its validation features [Radivoyevitch, 2004]. Rather, it uses the XML R package and parses the SBML documents directly. Unfortunately, the parser is buggy and incomplete and fails to parse many well-formed SBML documents. It also lacks support for the SBML layout extension. SBMLR does not offer the semantic validation features that libsbml gives to rsbml, and SBMLR is also quite slow, since it is implemented in R. In response to this, rsbml was designed to be a fast, complete, and reliable importer, exporter and validator of SBML data.

## 2 Features

The primary purpose of rsbml is to convert a libsbml parse result into an S4 object representation of an SBML model. The S4 classes are direct mapped from elements in the SBML specification, including the layout extension. Most of the SBML data types have obvious equivalents in R. SBML strings map to R character vectors, doubles to R numeric vectors, and integers to integer vectors. MathML elements are converted by libsbml into string expressions,

which are simply parsed into R expressions. rsbml does not attempt to parse the application-specific data contained within annotation elements. Instead, the RDF annotation is stored as a string, which may be passed to a custom parser or the Bioconductor Rredland RDF parser. Rredland converts the RDF into an R data frame, which is a simple and convenient basis for further processing.

A key feature of rsbml is the efficient construction of Bioconductor [Gentleman et al., 2005] graph objects from SBML. The graph object incorporates only the underlying graph structure of the SBML model. The species and reactions map to graph nodes, while the species references become the edges in the graph. The graph objects are fully compatible with the other graph-related Bioconductor packages, such as RBGL (a link to the C++ Boost graph library [Siek et al., 2002]), Rgraphviz (for laying out graphs using graphviz [Gansner and North, 2000]) and gaggle (a link from R to the gaggle framework [Shannon et al., 2006], which also includes Cytoscape [Shannon et al., 2003]). One may also use rggobi to convert a graph object into a GGobi edge set, facilitating the display of the network in a GGobi scatterplot.

### 3 Demonstration

The following demonstrates the reading, manipulating and writing of an SBML document using rsbml. Most users will begin an rsbml session by importing an SBML file into R. In the example below, we load an SBML file describing the glycolysis pathway. It is also possible to parse an R character vector instead of an external file.

```
> library(rsbml)
> file <- system.file("sbml", "GlycolysisLayout.xml", package = "rsbml")
> doc <- rsbml_read(file)
```

If errors are encountered, the function aborts and emits warnings describing the specific problem(s) with the document. Otherwise, the result is an opaque object referring to a low-level libsbml data structure. From here, the user currently has two options for accessing the data: as an S4 object conforming to the SBML document object model or as a Bioconductor graph object.

The following converts the opaque libsbml parse result to an S4 object:

```
> dom <- rsbml_dom(doc)
```

The result contains all of the information from the SBML document. Methods exist for getting and setting every element and attribute of the SBML specification (up to L2V3). The following demonstrates how one would retrieve all of the species IDs from the SBML model:

```
> s <- names(species(model(dom)))
> head(s, 1)
[1] "Glucose"
```

All SBML models have an implicit graphical structure. The following extracts the network into a Bioconductor graph object:

```
> g <- rsbml_graph(doc)
> n <- nodes(g)
> head(n, 1)
[1] "Glucose"
```

At this point, the graph can be passed to other packages, such as RBGL, Rgraphviz, etc.

The SBML specification provides many complex rules that ensure an SBML model is internally consistent. The following is an example of checking a document against those rules.

```
> rsbml_check(doc)
[1] TRUE
```

After creating/manipulating SBML objects in R, the result may be translated back to XML in two different ways: directly to a file or to an R character vector. This following code block demonstrates both methods:

```
> rsbml_write(doc, "my_glycolysis.xml")
```

NULL

```
> xml <- rsbml_xml(doc)
```

#### 4 Future

The scope of rsbml is limited to the importing of SBML models into R. Other R packages may utilize rsbml to visualize and analyze biochemical models stored as SBML, but rsbml does not and will not include any such functionality. Rather, future development will be in response to changes in the SBML standard, with the intent on maintaining full and robust support for all SBML documents.

# AN EXTENSIBLE SOFTWARE SYSTEM FOR INTERACTIVE NETWORK VISUALIZATIONS IN THE CONTEXT OF DATA ANALYSIS

A paper to be submitted to a visualization journal

Michael Lawrence

## Abstract

Visualization is a common feature of network analysis programs. However, many network visualizations do not incrementally adapt to the changing focus of an analysis. We propose a network visualization system based on an incremental, constraint-based layout algorithm. The system is implemented in the R statistical language, which facilitates integration of the network visualization with the analysis of supplemental data. The design of the system is extensible and independent of any particular network viewer. When driving multiple network views, user interaction is linked between the views. A GUI is provided for controlling the layout algorithm.

## 1 Introduction

We have designed an interactive, extensible network visualization system that is integrated with the R platform for statistical computing [R Development Core Team, 2005]. The system is implemented as an R package named *rcola* that provides:

- Integration with the data analysis features provided by the R platform

- An incremental layout algorithm that is adaptable to particular network analysis problems
- Pluggable and coordinated network view components that draw networks according to the output of the layout algorithm
- An extensible GUI for controlling the layout algorithm

A network model represents the connections between a set of entities. In network terminology, the connections are called *edges* and the entities they connect are called *nodes*. Network models are applied to the study of communication, social interaction, transportation, biology and many other areas. An example of a communication network is the Internet, where the nodes of the network are computers and the edges are the lines, such as telephone wires and fiber optic cables, that connect the computers. Analyzing the network model of the Internet might help optimize traffic by revealing bottlenecks.

Given the broad applicability of network analysis, there is a significant amount of research towards helping the data analyst understand network models. As networks are graphs, methods from the field of graph theory are often applied to the analysis of networks. Graph theoretical techniques help answer queries such as the shortest path between two nodes or whether a network contains any cycles. Information from graph theory is useful; however, obtaining it requires asking very specific questions about the network. Even if the analyst asked the right questions, it might be difficult to recognize complex, and perhaps unexpected, patterns from the results of graph theoretical queries.

Network visualizations assist the analyst in the recognition of such patterns by leveraging the perceptual powers of the human brain. They help the analyst detect patterns in the network that may not be obvious from the output of mathematical algorithms.

Visual properties of node and edge glyphs in a network drawing are often initialized to uniform values. For example, all nodes may be assigned the same initial shape and color. This is not a viable approach, however, for initializing the node positions. If all nodes were given the same position, the network structure would be virtually invisible due to the overlap. The

node positions should be assigned in a way that helps answer the questions of the analyst. For all but the smallest networks, the manual selection of node positions would be an excessively tedious task. Thus, many algorithms have been proposed for automatically positioning nodes according to certain aesthetic criteria and they are reviewed later in this paper.

Through the course of an analysis, the analyst will likely ask a number of different questions. Each question may be best answered by a particular set of visualizations. Thus, the network view needs to adapt to the current focus of the analysis. This requires interaction between the analyst and the view. Common features of interactive network views include:

- Panning and zooming to focus on a particular region of the network drawing
- Querying of nodes and edges for extra information
- Adjustment of node and edge visual properties, such as color and size

Like the network view, the layout of the network should adapt to the changing focus of an analysis. While manual adjustment of node positions is often possible, the process is tedious and somewhat defeats the purpose of automatic layout. The analyst could recompute the layout using a different algorithm, but this has at least two drawbacks. First, many layout methods optimize a layout globally, without regard for the original node positions. This often results in disorienting changes in the layout that disturb the “mental map” of the user [Eades et al., 1991]. Second, most algorithms apply the same rules to every node in the network. The analyst may desire to layout a particular subset of nodes according to a specific set of criteria without significantly affecting the rest of the layout. The layout algorithm in rcola is capable of this.

Our software is implemented in the R language, because networks are rarely analyzed in the absence of supplemental data describing the nodes and edges. To facilitate the overall analysis of a system, a network visualization should be integrated with general data analysis features. The R platform provides a wide array of methods for data analysis, including numerical modeling and statistical graphics. Statistics calculated on the data may be graphically represented in the drawing of the network, and interaction with the network view may be linked to plots

of supplemental data. For example, moving the mouse over a node in the network view might highlight the corresponding point in a scatterplot from a dataset of node attributes.

The questions asked by a data analyst depend on the type of data being analyzed. The effectiveness of a graph drawing may be judged by its ability to answer the questions of the analyst. Accordingly, many graph layout algorithms are designed specifically for certain types of networks. However, it is infeasible to develop a layout algorithm from scratch for every type of network encountered. For this reason, the layout algorithm in rcola has been designed to be extensible.

Independent of the layout algorithm, the effectiveness of the network view also depends on its applicability to a particular data analysis task. For example, one network view may be integrated with data plots, while another may render nodes and edges in a domain-specific way. Analysts frequently use several different network viewers during an analysis. An interactive layout system should easily integrate with third party network views. In addition, if multiple views are in use simultaneously, the system should coordinate the views, such as through linked selection of nodes and edges. The rcola package has been designed to meet these requirements.

Finally, we have included a GUI for controlling the layout algorithm in rcola. There are two primary reasons for this. First, a GUI complements an interactive visualization. With a GUI, the user does not have to transition to the command line interface (CLI) of R in order to modify the network layout after interacting with the network drawing. The second reason is that many potential users of rcola are likely not experienced with the R language and lack the motivation to learn it. The GUI makes rcola accessible to such users.

The rest of this paper describes the rcola system in further detail. The next section reviews graph drawing techniques, including the algorithm employed by rcola. We then discuss the requirements of interactive graph layout. This is followed by a technical description of the rcola software. Finally, we conclude with a short discussion of open problems and future work.

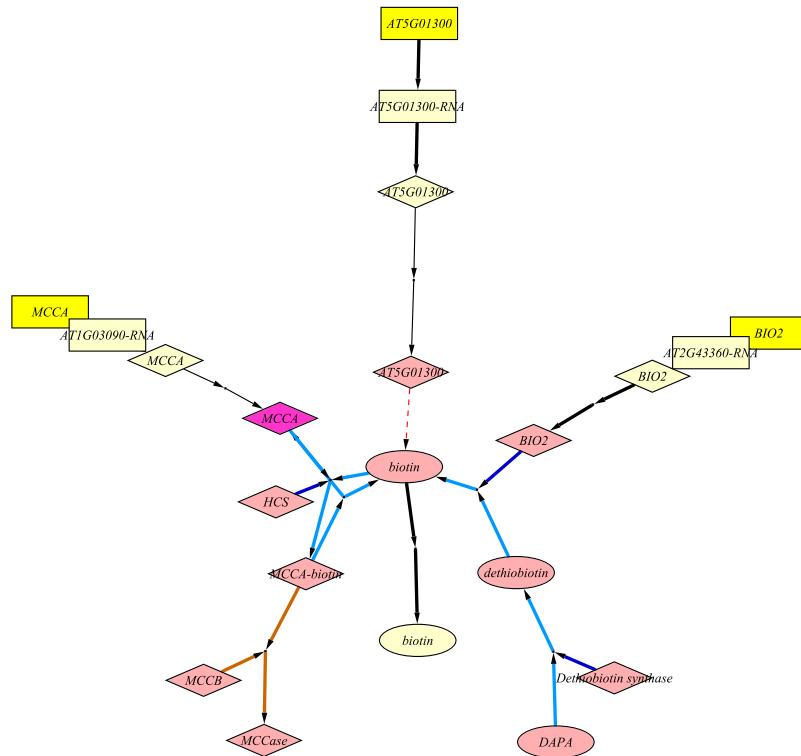


Figure 1 Cytoscape spring-embedded layout of a few steps of biotin synthesis and the production of MCCase, from the MetNetDB [Wurtele et al., 2003] Acetyl-CoA-biotin pathway. The nucleic acids are boxes (genes bright yellow). The enzymes are diamonds and the metabolites are circles. The colors indicate the compartment: yellow for nucleus, off-white for cytosol, and magenta for mitochondrion. The edge colors are also meaningful: cyan for enzymatic reaction, dark blue for catalysis, orange for forming a complex, dashed red for negative regulation (degradation), and black for transcription, translation, and translocation. The force-directed layout produces visually pleasing symmetry, but there is little sense of flow through the network. Node and edge overlaps further confuse the drawing.

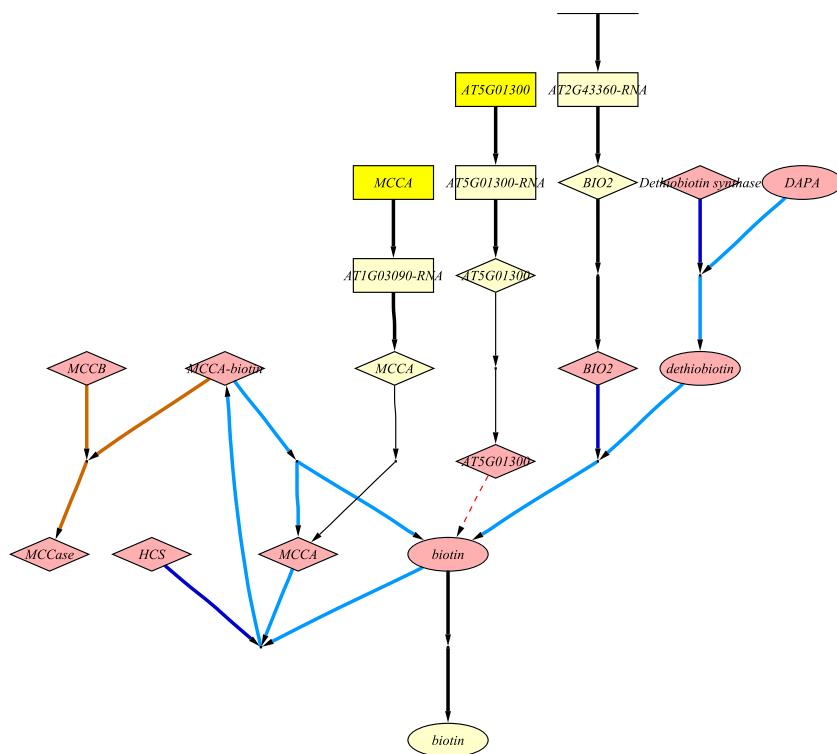


Figure 2 Dot layout (in Cytoscape) of the same network shown in 1. The flow through the network is clear relative to Figure 1, but it is difficult to perceive the cycle involving biotin, MCCA, and MC-CA-biotin.

## 2 Existing graph layout techniques

There are several classes of general layout algorithms, which are traditionally classified according their appropriateness for directed or undirected graphs. Popular undirected graph layout algorithms include force-directed placement and orthogonal layout. For directed graphs, hierarchical layout is dominant. Figures 1 and 2 show layouts of a biochemical network using a force-directed and hierarchical algorithm, respectively.

### 2.1 Orthogonal layout

Orthogonal layout constrains nodes to lay on an orthogonal grid and restricts edges so that they flow parallel and perpendicular to the grid lines [Sugiyama, 2002]. This quality makes orthogonal layout algorithms attractive for the drawing of circuit diagrams, for example, but it also generally restricts the applicability of orthogonal layout to a relatively small number of graph classes.

### 2.2 Force-directed layout

Force-directed placement is more generally applicable [Eades, 1984]. Conceptually, it treats a graph as a physical system of rings (nodes) connected by springs (edges). Some force-directed algorithms explicitly model the network as a physical system. In the specific method proposed by Eades, there are hidden springs between non-adjacent nodes that repel in order to enforce separation between nodes. The forces are simulated until the system reaches a stable state. If all edges are modeled identically, the edge lengths and node distribution tend to be uniform, which is generally an attractive quality for graph drawings [Sugiyama, 2002] (see Figure 1). An important feature of the force-based model is that the algorithm proceeds incrementally. This is ideal for user interaction, since the simulation can be restarted from a user-modified state and the progress of the algorithm is easily communicated to the user through animation of the graph.

There have been many extensions to the basic force-directed method. Kamada and Kawai [1989] abstract the mechanical system to a single energy function that is minimized as the

distance between each pair of nodes approaches a goal distance, which is canonically defined as the length of the shortest path between them. This method avoids explicit simulation of physical forces, which is slow and especially prone to local minima. Fruchterman and Reingold [1991] extend the Eades method so that the springs try to fill a drawing area by enforcing inter-node distances proportional to the ratio of the area to the number of nodes. Several extensions have added gravity to the force system, such as the GEM layout [Frick et al., 1994]. The force-directed approach has been applied to directed graphs by adding magnetism to the force model Sugiyama and Misue [1995], which enables control over the orientation of “magnetic” edges. By pulling the heads of the magnetic edges in a single direction, for example, the flow through the network may become more obvious. They also note that many additional types of forces could act as layout constraints. However, this strategy would likely suffer from the tendency of a complex physical system to become trapped in local minima.

### 2.3 Hierarchical layout

Hierarchical layout is often applied to acyclic directed graphics. When a graph is laid out hierarchically, its nodes are organized into horizontally, or vertically, aligned layers such that the directed edges tend to point from one layer to the next in the sequence [Sugiyama, 2002] (see Figure 2). Each layer is meant to represent a level in a hierarchy or stage in a process. The definitive hierarchical layout algorithm is due to Sugiyama et al. [1981]. The dot module of graphviz [Gansner et al., 1993] implements an optimized and refined version of the algorithm. The Sugiyama algorithm proceeds through four stages.

1. Make the directed graph acyclic. Any cycles are detected and broken by reversing the feedback edge.
2. Assign the nodes to layers such that the edges flow in a uniform direction, nodes are evenly distributed, and edge lengths are minimized.
3. Order the nodes within the layers to minimize edge crossings.
4. Position the nodes to fill the space and make edges straight.

The algorithm often performs well for directed acyclic graphs, especially those representing a hierarchical structure. However, if a graph has cycles, the algorithm tends to draw them asymmetrically by forcing directed edges to point in the same direction. It also has a tendency to produce excessively long edges in large graphs.

## 2.4 Constraint-based layout

The algorithms discussed so far are applicable to a wide range of graph types. However, their generality is also a drawback, in that they do not consider the specific semantics of a particular type of graph or network. This may decrease their effectiveness at answering questions specific to a problem domain. It is infeasible to develop a special algorithm from scratch for every type of graph. An alternative approach is to apply domain-specific constraints on top of a general layout algorithm.

There is a large body of literature on constraint-based graph drawing. Constraint-based algorithms are often based on force-directed placement, since it provides a simple mechanism for balancing constraints: energy minimization. The first attempts at including constraints in force-directed layouts use explicit forces to model constraints. For example, the magnetic force has been applied to control the orientation of “magnetized” edges [Sugiyama and Misue, 1995]. Taking inspiration from this idea, one might imagine a layout that applies multiple types of forces to different subsets of nodes and edges, achieving an effect similar to compound layout but within a single layout. However, as the number of constraints grows, the system of forces quickly gains in complexity, making optimization difficult. For example, the GLIDE system [Ryall et al., 1997] enforces a number of constraints by varying the spring constant of each edge depending on the relative locations of the nodes incident to the edge. The system is likely prone to local minima and not scalable to large graphs.

Other constraint-based approaches avoid explicit force simulation. The object-based constrained layout system COOL [Kamada and Kawai, 1991] lays out graphs in two separate stages. First, it uses the algorithm of Kamada and Kawai to produce an unconstrained layout. The second stage adjusts the initial layout with constraints. Unfortunately, the separation

into two stages is undesirable, because the constraints are not considered throughout the entire process. The result of the first stage, which is oblivious to constraints, may be such that it hinders the performance of the second stage. In order to address this limitation, He and Marriott [1998] devise a set of layout algorithms with fully integrated general linear constraints using a quadratic programming optimization method called active sets, but the method has not been shown to be scalable to graphs with more than about 20 nodes.

## 2.5 The IPSep-CoLa algorithm

Another quadratic programming technique, known as energy majorization, has been recently borrowed from the field of multidimensional scaling [Gansner et al., 2005]. It demonstrates better efficiency in the layout of large graphs compared to Kamada and Kawai. By extending the energy majorization technique to include constraints for the orthogonal ordering of nodes, an algorithm called Dig-CoLa [Dwyer et al., 2006] produces drawings of directed graphs in which the nodes are positioned across a sequence of horizontal layers, in the spirit of the Sugiyama hierarchical layout. The goal of Dig-CoLa is similar to that of Kamps et al. [1996] who augmented the objective function of Eades [1984] with an “angle force” for the orthogonal ordering of nodes. Dig-CoLa contrasts with this approach in that its constraints are guaranteed, instead of being mere forces. Also, Dig-CoLa is likely more efficient for large graphs.

The layout algorithm employed by rcola is a generalization of Dig-CoLa known as IPSep-CoLa [Dwyer and Marriott, 2006]. It supports node separation constraints specified in either the X or Y dimension. A separation constraint orthogonally orders a pair of nodes and enforces a gap between them. The constraint is expressed mathematically in Equation 1 for two nodes  $a$  and  $b$  with a minimal gap  $g$  in the X dimension. The relational operator in the equation is  $\leq$  but  $=$  is also possible.

$$a_X + g \leq b_X \quad (1)$$

This additional flexibility allows IPSep-CoLa to, for example, avoid node overlaps and to group nodes into bounding rectangles. Through the combination of separation constraints, it is pos-

sible to construct high-level domain-specific constraints. Thus, IPSep-CoLa is easily adaptable to specific data analysis tasks. As a force-directed algorithm, IPSep-CoLa is incremental, which, as discussed in the next section, is an important requirement for integration with an interactive network view.

### 3 Interactive graph layout

Ultimately, it is the user who decides whether a layout is effective for the problem at hand. The user should be able to choose, on the fly, the most appropriate layout for the current task. This is especially important in the context of exploratory data analysis, as the questions asked by the user are constantly changing.

There are several tools described in the literature that allow the user to interactively constrain a graph layout. One of the earliest is the ExtenDible Graph Editor (EDGE) [Newberry and Tichy, 1990], which has been extended to handle constraints [Böhringer and Paulisch, 1990]. EDGE allows the user to specify different types of constraints, including absolute (fixed) position, relative (top, bottom, left, right) position to other nodes and clusters. It relies on a modified version of the Sugiyama layout. The Graph Layout Interactive Diagram Editor (GLIDE) [Ryall et al., 1997] also supports user defined constraints. The constraints in GLIDE are based on the Visual Organization Features [Kosak et al., 1994]. The layout algorithm of GLIDE is based on a complex system of springs that is likely prone to local minima. The more recent GDHints system [do Nascimento, 2001] allows the user to offer “hints” to a layout algorithm in order to achieve a more optimal layout, according to predefined aesthetic criteria. GDHints differs from the previous two tools in that the latter help the user to sculpt a graph drawing towards a user-defined goal, while GDHints restricts the user to helping the algorithm towards its global optimum. The first two tools, EDGE and GLIDE, are closest in their goals to this project, except they are not designed to be used in conjunction with data analysis.

An important problem in interactive layout is the update scheme. Transitions between layouts must attempt to preserve the mental map of the user [Eades et al., 1991]. The mental

map is modeled according to three criteria: orthogonal ordering, proximity, and topology. This means that an adjustment to the layout should avoid changing the relative ordering of nodes, increasing the distance between proximal nodes, and moving nodes to different logical regions of the diagram. An algorithm wishing to preserve the mental map should keep changes along these axes to a minimum.

A variety of distance metrics have been proposed to measure the preservation of the mental map according to these criteria. A user survey has been conducted to evaluate the various layout distance metrics with regard to their ability to predict the amount of difference perceived by the user [Bridgeman and Tamassia, 2002]. The authors find that the top performing metrics are those based on the relative positions of nodes. Specifically, maintaining the orthogonal ordering and the identity of the nearest neighbor of each node seem to be the most effective methods for preserving the mental map.

Layout constraints are useful in controlling the adjustments to a layout. This is demonstrated by EDGE, which relaxes the constraints on the nodes in the region of the change. The separation constraints supported by the IPSep-CoLa algorithm may be applied to the preservation of the mental map. In particular, separation constraints can preserve the orthogonal ordering of the nodes, which, as noted above, is likely an effective means of mental map preservation.

Animating the necessary changes in the layout further helps to maintain the mental map. Changes that must occur should be communicated via an interpolated animation. The incremental nature of force directed layouts allows them to produce animated transitions without violating any constraints. Thus, as a force-directed layout algorithm, IPSep-CoLa should be capable of the incremental state transitions that help preserve the mental map.

## 4 The rcola package

We have developed the rcola package with four goals:

1. Integrate with the data analysis features provided by the R platform
2. Incorporate an incremental and adaptable network layout algorithm

3. Support pluggable and coordinated network view components

4. Provide a GUI for controlling the layout algorithm

All of the above goals have been met by rcola. As rcola is implemented in the R language, it is implicitly integrated with the data analysis capabilities of the R platform. This satisfies the first goal. The remaining three goals are each addressed by a specific component of rcola. The design of rcola is shown in Figure 3. The core of the system is a link to the implementation of the IPSep-CoLa layout algorithm. The second component provides an abstraction for embedding network views from network visualization packages like GGobi [Swayne et al., 2003a] and Cytoscape [Shannon et al., 2003]. The final component is a graphical user interface to the layout engine.

#### 4.1 Integration with IPSep-CoLa

The IPSep-CoLa algorithm is implemented in the C++ library adaptagrams [Dwyer, 2007]. The rcola package provides a high-level interface from R to adaptagrams for sending the graph structure, constraints and other parameters to IPSep-CoLa, executing the algorithm, and returning the result. For each iteration of the algorithm, rcola calls back into R in order to update any network views. The graph structure is an instance of the graph class from the Bioconductor graph package. This ensures interoperability with other graph-related packages. Constraints are represented as S4 objects. A new type of high-level constraint may be defined through inheritance, as long as it provides a means for reducing itself to a set of separation constraints, which are passed on to IPSep-CoLa.

The direct specification of separation constraints gives the R user low-level control over the constraint specification. However, it is often more intuitive to express constraints at a higher level. The adaptagrams library provides several simple high-level constraints, and rcola makes some of these available to R. A description of each available constraint follows:

**Boundary** Forces a set of nodes to the left of (or on top of) another set

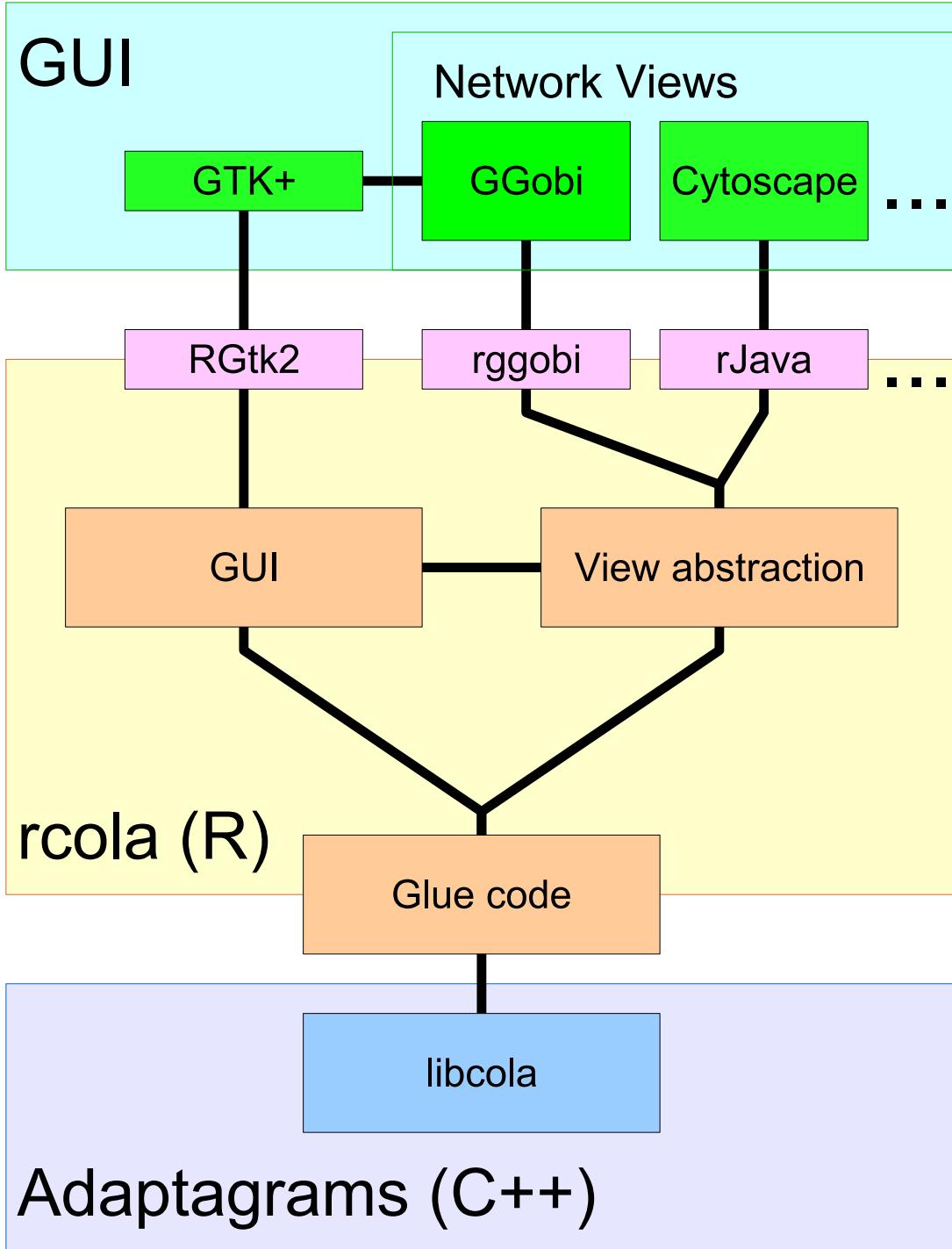


Figure 3 Design of rcola, which consists of a GUI, a network view abstraction layer, and a binding to the adaptagrams library [Dwyer, 2007]. The user interacts with the GUI to generate layouts using adaptagrams, and the layouts are passed to one or more external network view components.

**Alignment** A boundary constraint where the separation constraints use the = operator rather than  $\leq$ , resulting in the nodes having fixed separation (often zero) in the X or Y dimension

**Multiple Separation** An ordered set of alignment constraints with fixed or variable separation between them

**Page Boundary** Enforces empty margins on the edges of a drawing

On top of those listed above, the rcola package provides a *sequence* constraint, which is an ordered set of boundary constraints. Both the sequence and multiple separation constraints are useful for ordering nodes along one dimension in the network view. The sequence constraint is essentially a less rigid variant of the multiple separation constraint. By employing boundary constraints, the sequence constraint does not force the nodes of the same rank to be aligned, as in the multiple separation constraint.

The ability to order nodes along one of the layout dimensions is one means for representing supplemental data in the layout. The order may be derived from a variable describing the nodes. Categorical variables are especially applicable, as nodes generally have more values in common along a categorical variable compared to a real-valued quantitative variable. The more values the nodes have in common, the less constrained and likely more understandable the layout.

The use of ordering constraints is similar to plotting the nodes in a scatterplot with one dimension from the unconstrained layout and the other from the ordering variable. However, there are several differences. Unlike the scatterplot, the constraints do not fix the positions of the nodes. The layout algorithm is still free to optimize the layout by moving the nodes as long as their order in the specified dimension is not violated. Also, in the scatterplot, the layout algorithm is not aware of the positions of the nodes in the fixed dimension, so it is unable to use that information to optimize the coordinates in the free dimension. Finally, the order constraints can be applied to a subset of nodes, while the scatterplot fixes the position of every node.

## 4.2 Network views

There is a wide variety of tools available for displaying networks, many of which are specific to particular problem domains. Some packages visualize networks in conjunction with other data, while others focus on graph editing or simulation and analysis of graphical models. The analyst should be able to choose the right tool for the job at hand. The rcola package aims to integrate with existing tools by providing an abstraction layer that adapts external network views to the rcola framework.

A network view component is primarily responsible for drawing the network layout. At the very least, the user should be able to select nodes with a pointer device. Network views should support the registration of callback functions that are invoked in response to user events, such as node selection and position changes. When the layout changes or a network is opened or closed, rcola notifies the view to update its drawing. The precise API that must be implemented by a network view is provided in the rcola documentation.

It is common for multiple views to be used simultaneously during an analysis, so rcola provides a simple means for coordinating views: linked selection of nodes. This provides a visual link between views and keeps the selection state consistent between them. More linking mechanisms may be scripted in R on top of rcola.

As we are primarily interested in the integrated visualization of networks and supplemental data, the default viewer in rcola is based on GGobi [Swayne et al., 2003a], a tool for multivariate interactive graphics. A screenshot of the GGobi view in rcola is shown in Figure 4. GGobi is essentially a point and line drawing tool and draws networks using points for the nodes and straight lines for the edges. Supplemental data on the nodes and/or edges may be displayed using GGobi data plots, which include scatterplots, parallel coordinate plots, and barcharts. Data points, including the nodes in the network, may be colored using the brush tool. When a point for a node is brushed in one plot, the corresponding point in each of the other plots is given the same color. This links the data plots and the network view through visual cues. GGobi is a stand-alone tool written in C. The rggobi package [Temple Lang, 2001b] provides the interface between R and GGobi.

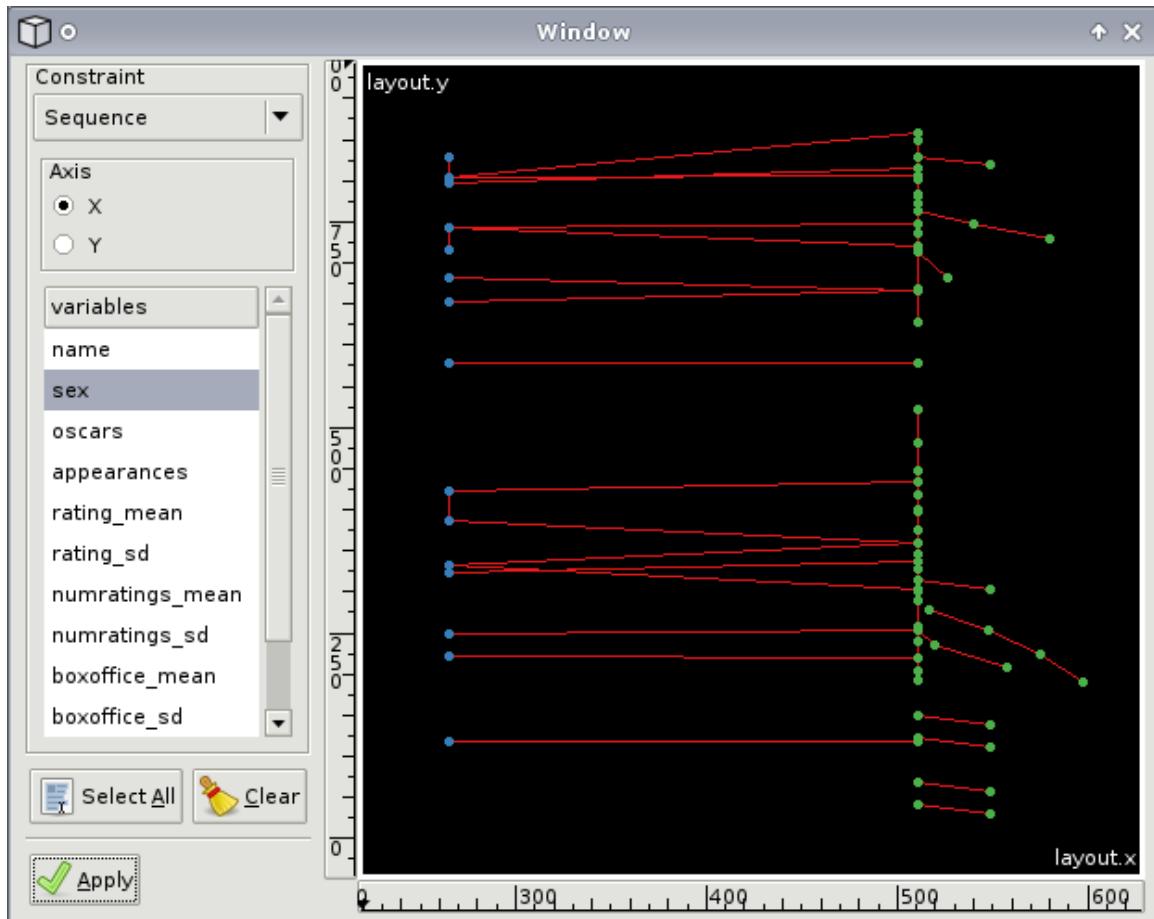


Figure 4 The GGobi-based rcola network view. The GUI for specifying layout constraints is to the left of the network drawing. The network is one of actors where edges connect the actors that have co-appeared in a movie. A sequence constraint has been applied to separate the actor nodes in the X dimension by the sex of the actor. The nodes are colored according to their sex, where blue represents female and green represents male.

### 4.3 The GUI

Each network view is associated with a graphical control panel for configuring and applying constraints on the selected nodes in the view. In Figure 4, the control panel is to the left of the network drawing. When the user selects constraint type in the above combobox, the panel below is updated to include controls for configuring the parameters specific to the constraint type. In the figure, the user has selected the sequence constraint, so the GUI displays a list of variables that may be used determining the order of the sequence. Below that is a set of radio buttons for choosing whether to apply the constraint along the X or Y dimension. Finally, there is a button for applying the configured constraint to the selected nodes.

The GUI has been designed to be extensible, so that it can support constraint types that are not included with rcola. To extend the GUI, the developer needs to implement a constraint factory that creates instances of the custom constraint type. The factory must also provide the GUI controls for configuring the parameters specific to the constraint type.

As a control panel is associated with a particular network view, it needs to be embedded into the same GUI as the view component. This is a challenge, because network views are implemented in a variety of GUI toolkits. It would be impractical to reimplement the control panel in every toolkit. Thus, we have written the rcola GUI using the gWidgets package [Verzani, 2007a], which provides a cross-toolkit API for R. Several implementations of the gWidgets API are available, including GTK+ [Krause, 2007], Java/Swing [Sun Microsystems, 2007], and tcl/tk [Ousterhout, 1994]. Depending on the gWidgets implementation in use, the control panel is embeddable into GUIs constructed with any of those toolkits. The GTK+ backend permits embedding with GGobi, as its GUI is based on GTK+. The control panel has also been embedded into the Cytoscape biological network viewer [Shannon et al., 2003] using the Java/Swing backend.

## 5 Conclusion

In summary, the rcola package is an extensible system for interactive constraint-based network visualization based on the IPSep-CoLa algorithm. The package provides a GUI for

specifying the constraints and supports the definition of custom constraint types for particular applications. The system is designed to embeddable into third party network analysis software and can serve as a bridge between views from separate software packages.

Future development of rcola will focus on adding more network view backends and constraint types. There is particular interest in constraints that are derivable from supplemental data, in a similar vein to the existing sequence constraint. This furthers our overall goal of integrating network visualization with multivariate data analysis.

## INTERACTIVE BIOCHEMICAL NETWORK VISUALIZATION FOR EXPERIMENTAL DATA ANALYSIS

A paper to be submitted to a Bioinformatics journal

Michael Lawrence

### Abstract

Biochemical systems are commonly represented as networks, and biochemical networks are helpful for experimental data analysis, as they relate the individual measurements to the biological system. This has motivated the development of specialized tools for viewing and analyzing biochemical networks. When drawing the networks, it is important to communicate the biological information encoded in the network structure and metadata. We have developed a software system for drawing biochemical networks according to biological aesthetics. The visualization is interactive and incrementally adapts the drawing to fit the evolving focus of an exploratory data analysis. The software is integrated with the exploRase application for the exploratory analysis of high-throughput experimental data.

### 1 Introduction

In this paper, we have developed methods for the interactive visualization of biochemical networks in the context of experimental data analysis. The proposed approach to network visualization attempts to satisfy the following requirements:

- An incremental layout algorithm designed to communicate biological information through its drawings.

- A visualization that adapts to the changing focus of a data analysis through interaction with the user.
- Integration with experimental data analysis.

Biological experiments often measure multiple aspects of the cellular system. Microarrays measure levels of RNA transcripts, while mass spectrometry and other methods detect protein and metabolite levels. Each dataset provides one perspective on the state of the cell. In order to understand their data, biologists need to integrate it at the cellular level.

Networks benefit experimental data analysis by providing a framework that relates low-level measurements to the biochemical system. They integrate experimental datasets from different sources by specifying the interactions between biological entities of different types, such as enzyme transcripts and metabolites. This helps biologists form system-level hypotheses from measurements on individual biochemicals.

Effective graphics enhance the comprehension of data. This applies to both experimental data and network data. The central problem in the drawing of biochemical networks is the assignment of node positions. The size and complexity of biological networks precludes manual layout. Thus, algorithms are necessary to automatically determine node positions based on a set of aesthetic criteria.

Numerous network layout algorithms have been proposed, but most do not specifically consider biological semantics. Thus, they often fail to communicate biologically interesting information. Biological network analysis tools often rely on node and edge visual attributes, such as color and shape, to communicate biological information. This is an effective technique; however, it ignores the utility of the layout in communicating information to the user. By encoding information in the layout, the appearance of nodes and edges can be used for other purposes, such as visual cues for linking the network drawing to plots of experimental data.

We have identified the following aesthetic criteria as important for the effective layout of biochemical networks:

**Flow** The reactant and product nodes of a reaction should be correctly segregated on opposite sides of the reaction node. This makes the flow through reactions more obvious.

**Catalysis** The catalyst node(s) of a reaction should be positioned orthogonally to the axis formed by the reactant and product nodes.

**Co-factors** Co-factors or “side compounds” should be distinguishable from the primary reactant and product nodes.

**Cycles** Cycles in the network should have a symmetric layout.

**Compartments** Nodes representing biochemicals are often associated with a particular cellular compartment. This can be communicated by segregating nodes into separate regions of the drawing according to their compartment.

A second concern is the incremental adaptability of the layout. While interactive network visualizations are fairly common, the network layout is rarely adjustable by the user, except by direct positioning of individual nodes or recomputing the entire layout with a different algorithm. The user should be able to adapt the layout so that it best answers the current question of the analysis, without excessive user intervention or disorienting changes to the layout [Eades et al., 1991].

The biochemistry of the cell is large and involves thousands of interactions. It is difficult to satisfy strict aesthetic criteria when drawing large, complex networks. The energy landscape tends to have many local minima, and conflicts often arise between layout constraints. A layout strategy based on user feedback partly solves this problem by only requiring constraints to be applied in regions of interest to the user. This likely helps the optimization process to find a lower local minimum by placing less constraints on the algorithm. It also provides an elegant means of resolving conflicts between constraints: the most recently applied constraints are given precedence over the others.

In summary, we have derived the following requirements for a biology-aware network layout algorithm:

1. Satisfaction of the biological aesthetic criteria stated above.
2. Incremental optimization scheme for adapting to user feedback and animating transitions.

3. Scalability to networks on the order of hundreds or thousands of nodes.

The software system we have developed incorporates biology-specific aesthetic criteria and adapts the layout in response to user feedback. In the interest of integrating networks with experimental data analysis, the software is implemented on top of the R platform for statistical computing [R Development Core Team, 2005] and the network visualization is linked to plots of experimental data. The software is based on the rcola package and is available as part of the exploRase application [Lawrence et al., 2007a, 2006]. The implementation is a work in progress. Everything is functional, except for the biology-specific constraints, which are currently under development.

This paper continues with a review of existing algorithms for biology-specific network layout. We then introduce the constraint-based layout algorithm that we have extended to consider biological semantics. This is followed by a technical description of the software. We conclude with a summary and a discussion of future work.

## 2 Existing biology-aware layout algorithms

An early example of an algorithm designed specifically for the layout of biochemical networks grew out of the Ecocyc project [Karp and Paley, 1994]. The apparent goal of the algorithm is to approximate the style of network drawings in biological textbooks. The authors acknowledge the need to draw cycles in a circular way, while orienting the flow of the acyclic pathways in a single direction. Their algorithm distinguishes between branched and linear pathways, drawing the former as a tree and the latter as a “snake” that winds back and forth to fill a page. The distinction between branched and linear pathways seems unnecessary, given the tendency of biologists to visualize large, highly connected networks that would include much more than a single completely linear pathway. Furthermore, the pan, zoom, and scroll capabilities of modern user interfaces mitigates the need to fit an entire linear sequence on the screen. The authors also suggest that the participation of catalysts and side compounds should be drawn perpendicular to the main flow of the reaction and on opposite sides of the reaction.

The Ecocyc algorithm begins by finding the largest cycle in the network. The cycle is laid out using a circular layout algorithm. These steps are repeated recursively on the remaining portion of the network, until a purely cyclic or acyclic subgraph is encountered. In the case of an acyclic subgraph, a tree or linear layout algorithm is applied, depending on whether the subgraph is branched or linear. The subgraph layouts are collapsed into super-nodes, which are laid out using a tree algorithm, since there is only one connection between each supernode. The super-nodes are then expanded in the final drawing. The main drawback to this compound layout approach, as illustrated in Figure 1, is that the subgraphs are laid out in isolation. Nodes are not placed proximal to their adjacent nodes in other subgraphs, so the final drawing is suboptimal. In particular, there is a large number of edge crossings.

Becker and Rojas [2001] propose a modification to the Ecocyc algorithm with the goal of reducing the number of generated edge crossings. The layout of the supernodes is calculated using a force-directed algorithm where the endpoints of the edges are each attached to a “center of gravity” in a supernode. A center of gravity for an endpoint is the average position of the nodes in the supernode that are adjacent to a node in the supernode at the other endpoint of the edge. The hope is that considering the positions of the nodes within the supernode will help avoid edge crossings in cases like Figure 1.

Wegner and Kummer [2005] extend the Becker and Rojas approach so that it deemphasizes side compounds by drawing them to the side of the sequence of main compounds. Also, they search for the smallest, instead of the largest, cycles when breaking the network into subgraphs. If a node belongs to two cycles, it is split into two nodes, so that each subgraph is node disjoint. When the subgraphs are reintegrated, the split nodes are joined. This results in drawings with many circular layouts chained together. Node splitting is also used to reduce the degrees of nodes and to minimize edge crossings. This is claimed to reduce the complexity of the drawing, but it also seems to consume more space due to the large number of dummy nodes.

The BioPath layout algorithm [Schreiber, 2002] adapts the Sugiyama layout to produce a compound layout of a biochemical network, using similar aesthetic criteria to the Ecocyc approach. One additional consideration is the layout of “open cycles” referring to recursive

biological processes, such as in the metabolism of fatty acids. The algorithm relies on a specific database to detect these in the network; such detection is not feasible in general. A super node is formed for each reaction containing its catalyst and side compounds. Cycles are also clustered into super-nodes. The Sugiyama layout is adapted to handle the varying sizes of the super-nodes.

The layout algorithm from the BioMAZE project [Gabouje and Zimanyi, 2005] follows a very similar approach. Their algorithm integrates metabolic and regulatory networks into a single layout by forcing the regulatory pathways to flow orthogonally to the metabolic pathways. This seems analogous to the orthogonal placement of enzymes by the Ecocyc algorithm, since a catalyst is performing a regulatory role in its reaction. The algorithm lays out cycles, branching pathways and linear pathways individually, using a compound variant of the Sugiyama layout very similar to that of the BioPath approach.

The compound layout algorithms generate layouts that are reasonably effective at communicating biological information. Thus, they satisfy, to a degree, our first requirement. However, they fail to meet the second requirement for an interactive network visualization: an incremental optimization scheme. Incorporating incremental optimization into a compound layout algorithm would likely be a difficult task due in large part to the need for nodes to transition between layout algorithms. The invisible seams in the layout would inhibit seamless transition and interaction. For example, if the user requested that a node be moved into the region of a supernode other than its parent, it is not clear how the algorithm should adapt to accommodate the change. Compound layout algorithms do not seem amenable to an interactive setting.

An alternative approach is taken by the PATIKA software [Dogrusoz et al., 2004]. [Dogrusoz et al., 2004], which implements a force-directed algorithm for drawing biochemical networks. Force-directed layout algorithms [Eades, 1984] treat a graph as a physical system where nodes are rings and edges are springs attached to the rings. As the system is simulated, it tends towards an energy minimum. Thus, the optimization process is naturally incremental and there are no seams in the layout. This suggests that the force-directed approach may be a

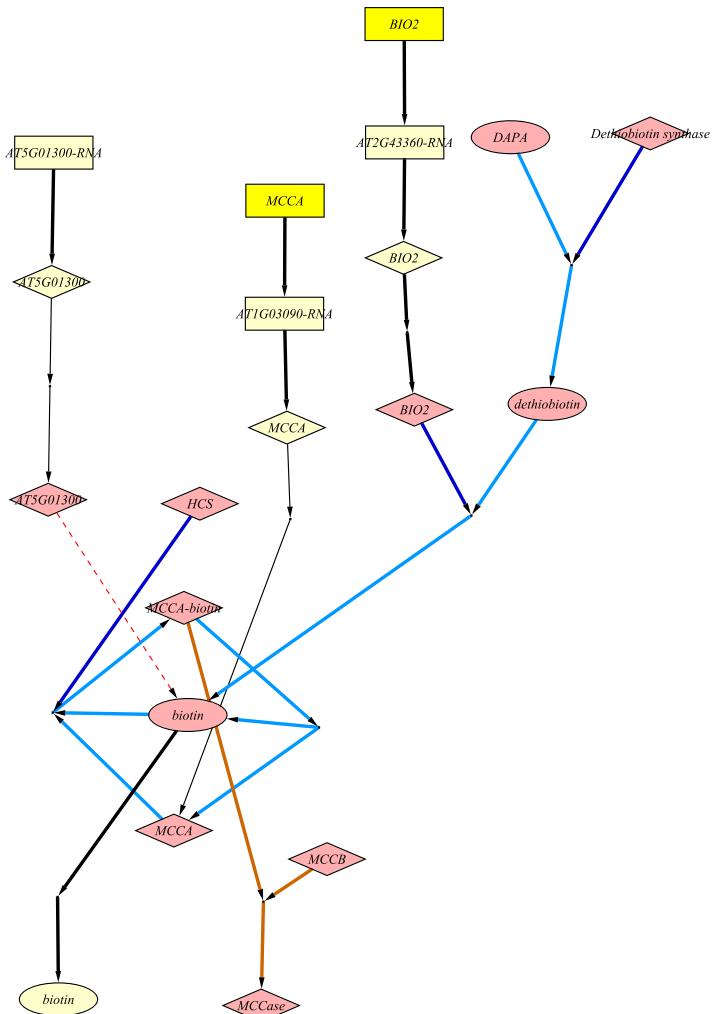


Figure 1 Generalized illustration of the compound layout strategy to biochemical network layout. The network represents a few steps of biotin synthesis and the production of MCCase, from the MetNetDB [Wurtele et al., 2003] Acetyl-CoA-biotin pathway. The nucleic acids are boxes (genes bright yellow). The enzymes are diamonds and the metabolites are circles. The colors indicate the compartment: yellow for nucleus, off-white for cytosol, and magenta for mitochondrion. The edge colors are also meaningful: cyan for enzymatic reaction, dark blue for catalysis, orange for forming a complex, dashed red for negative regulation (degradation), and black for transcription, translation, and translocation. The cyclical, branched, and linear structures are laid out separately and are easily recognized. However, the independent layout of the cycle poorly positions the pink biotin at the center of the cycle, which leads to two node-edge overlaps and many edge crossings.

better candidate than a compound algorithm for incorporation into an interactive visualization system. The challenge is to design a physical system that emphasizes biological aesthetics as its energy is minimized.

The PATIKA approach extends an existing force-directed placement algorithm [Fruchterman and Reingold, 1991] so that it considers several biological aesthetics. One of the goals is to organize nodes into rectangular representations of subcellular compartments. The rectangles are modeled as having elastic borders. Nodes belonging to a particular compartment are trapped within the rectangle for that compartment. The borders of the compartment are free to expand and contract as the drawing of its subgraph changes in size. PATIKA also introduces a “relativity force” that encourages the reactants and products to group together on opposite sides of a reaction. This helps indicate the flow through the network. The same force pushes the catalyst off the axis formed by the reactants and products.

The PATIKA approach seems compatible with our stated aesthetic goals, and it also meets our requirement for an incremental algorithm. However, it is not clear how well the quality of the layouts generated by PATIKA scale with the number of nodes in the network. The various forces may begin to contradict each other in large, complex networks. The conflicts might negate the benefits of the constraints. Also, the organic appearance of a purely force-directed layout does not often have the clarity of the more structured drawings of biochemical networks found in textbooks. Stricter constraints could be applied, but they would further hinder the optimization process. The algorithm presented in the next section addresses these concerns.

### 3 Biochemical network layout using separation constraints

We require an algorithm that supports the incremental application of biology-specific constraints to subsets of nodes as indicated by the user. IPSep-CoLa [Dwyer and Marriott, 2006] is an incremental force-directed layout algorithm that incorporates node separation constraints of the form given in Equation 1.

$$a_X + g \leq b_X \quad (1)$$

Separation constraints enforce an ordering and minimum separation between two specific nodes in either the X or the Y dimension. Equation 1 forces a node  $a$  to the left of a node  $b$  with a minimum separation of  $g$  in the X dimension. When a constraint is applied, IPSep-CoLa incrementally optimizes the layout to satisfy it.

Controlling a layout through direct use of separation constraints becomes tedious as the number of nodes increases. The implementation of IPSep-CoLa in the adaptagrams library [Dwyer, 2007] provides several compound constraints. One of them, the alignment constraint, forces a set of nodes to have the same coordinate in either the X or Y dimension. Appropriate application of vertical and horizontal alignment constraints to a layout imbues it with a structure that is reminiscent of network drawings in biological textbooks. The boundary constraint is another high-level constraint. It forces a set of nodes to be to the left or above another set.

We have found that alignment constraints, in certain combinations, are particularly effective at improving the clarity of reactions in a network drawing. These combinations have been encapsulated into a high-level *reaction constraint*. The reaction constraint can orient the transition from reactants to products in the North, South, East, or West direction. For reversible reactions, reactant and product are ambiguous terms, so the resulting layout is also ambiguous. Without loss of generality, we explain the reaction constraint in its South orientation, as diagrammed in Figure 2. The base reaction constraint applies separation constraints to force the reactant above the reaction and the reaction above the product. The alignment provides the option of horizontally aligning a single reactant and a single product with the reaction node. The choice of the reactant and product is generally random, but side compounds are excluded, if they can be identified. If a reactant or product node has a zero incoming or outgoing edges, respectively, it is heuristically identified as a side compound. Next, the reactants and products are segregated by boundary constraints so that the reactants are above reaction node and the products are below. The reaction constraint optionally applies a vertical alignment constraint separately to the primary reactants and products, so that they are distinguished from the side compounds. The catalyst may be aligned vertically with the reaction node.

The user could achieve the same effect as the reaction constraint through direct use of

alignment, boundary and separation constraints; the reaction constraint is merely a convenience for improving the clarity of biochemical network drawings.

A reaction constraint may be applied to any reaction of interest. This means that the algorithm only needs to constrain the layout of nodes that are of specific interest to the user. This likely increases the clarity of the most interesting parts of the drawing compared to an algorithm that applies constraints globally. If conflicts are introduced between constraints, the more recent constraints take precedence, in order to follow the focus of the user.

The one aesthetic criterion that the reaction constraint does not satisfy is the geometric segregation of nodes according to their compartment. The IPSep-CoLa algorithm is able to distribute groups of nodes into disjoint convex regions of the drawing. The size and shape of each region adjusts to accommodate the drawing of the nodes within it. This achieves the desired effect of visually representing the physical separation of chemicals between cellular compartments.

#### **4 Software implementation**

We are in the progress of implementing our methods as part of the exploRase application [Lawrence et al., 2007a, 2006]. The network view and layout GUI are operational, but the reaction constraint and compartment segregation are under development.

The exploRase package is based on the R platform for statistical computing [R Development Core Team, 2005] . The purpose of exploRase is to help biologists understand their data through the integration of numerical algorithms and interactive visualizations of experimental data and biochemical networks. The improved exploRase network visualization will assist biologists as they analyze their experimental data.

The implementation of the system is based on the rcola package [Lawrence, 2007a], which provides an interface to the IPSep-CoLa algorithm in the adaptagrams library [Dwyer, 2007], as well as a GUI for controlling the algorithm. It also defines an API for integration with external network view components. The package includes an implementation of the API based on the rggobi package [Temple Lang, 2001b], which interfaces R with the GGobi tool for multivariate

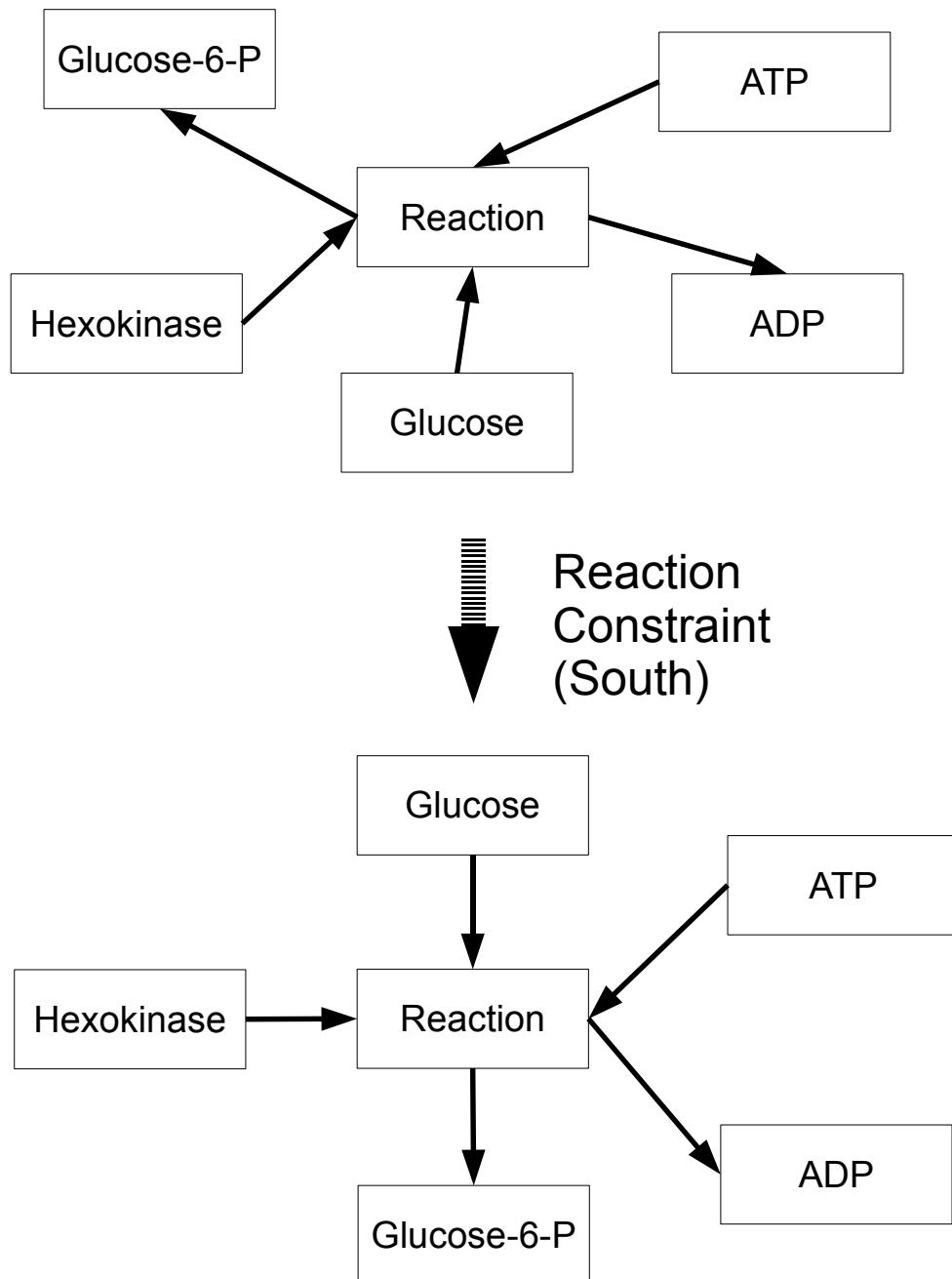


Figure 2 Illustration of the rearrangement of a reaction via the reaction constraint. The catalyst is hexokinase and ATP and ADP are considered side compounds.

interactive graphics [Swayne et al., 2003a]. A screenshot of the GGobi-based rcola view is shown in Figure 3.

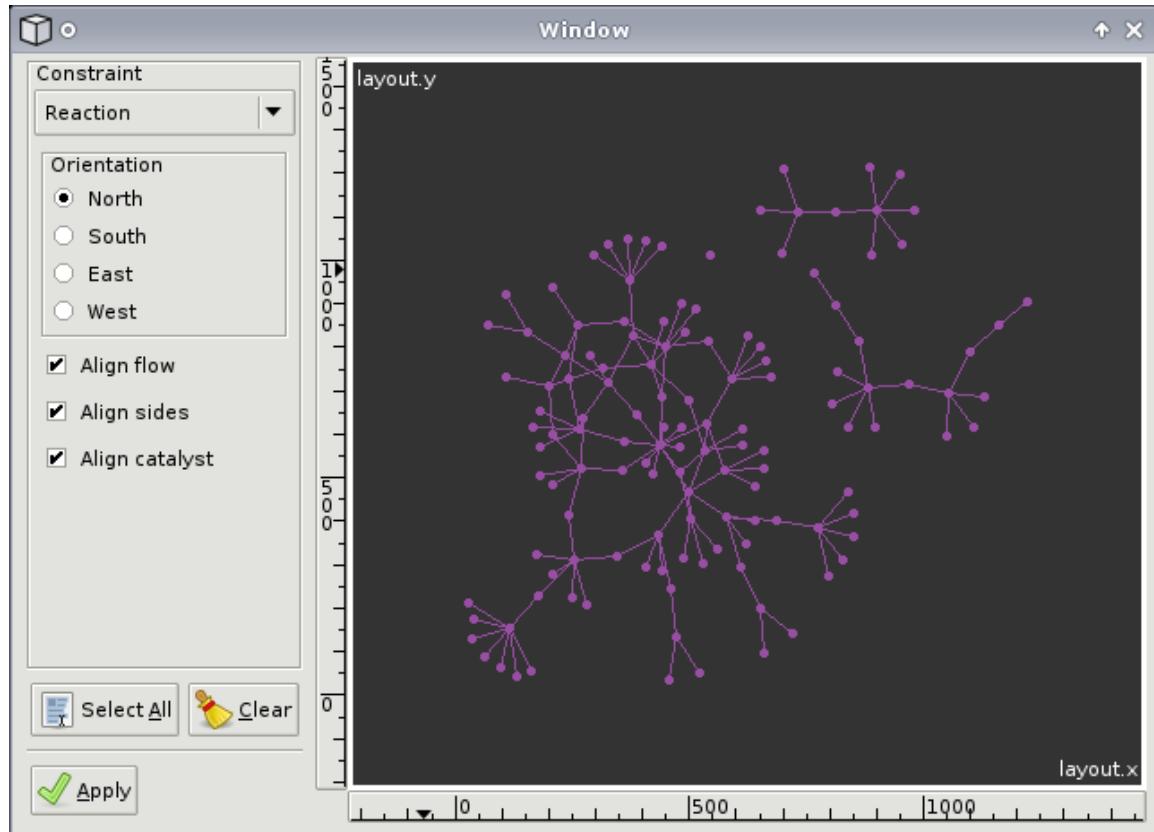


Figure 3 The GGobi-based rcola network view. The GUI for specifying the reaction constraint, and others, is to the left of the network drawing. The displayed network represents a portion of biotin and fatty acid metabolism in Arabidopsis.

This integration of rcola with GGobi is fortunate, as exploRase already relies on GGobi for interactive plot of experimental data. Interaction with the network view is linked to the experimental data plots. For example, if the user brushes a node a certain color, the point corresponding to the node in each of the other plots is assigned the same color. This provides a visual cue for linking the high-level system with the low-level experimental measurements.

The rcola package is designed to be extensible, which facilitated the incorporation of the reaction constraint into the rcola GUI. The reaction constraint configuration controls are shown on the left in Figure 3. The user can apply the reaction constraint to any set of reaction nodes

selected in the drawing on the right. The segregation of nodes according to their compartment is possible through a different part of the exploRase GUI.

We have also extended rcola to support an additional network view, based on the Cytoscape application [Shannon et al., 2003]. The layout GUI is directly embedded within the Cytoscape interface. The Cytoscape view is somewhat complementary to the GGobi view as Cytoscape draws networks in a visually rich style but does not provide any data plots. The biologist may find the Cytoscape style more accessible, for certain purposes, than the skeletal points and lines drawing of GGobi. The rcola package links the selection between the network views, so selecting a node in Cytoscape highlights its points in the GGobi plots, and vice versa.

## 5 Conclusion

This paper has presented methods for interactive biology-aware network visualization that relies on a flexible, constraint-based layout algorithm. By integrating the methods in the exploRase package, we have made them directly accessible to biologists analyzing experimental data.

We plan to progress by conceiving more high-level constraints that help the user retrieve biological information from network layouts. We are also considering the development of additional network views to complement those based on GGobi and Cytoscape.

## EXPLORASE: MULTIVARIATE EXPLORATORY ANALYSIS AND VISUALIZATION FOR SYSTEMS BIOLOGY

A paper published in Journal of Statistical Software

Michael Lawrence, Dianne Cook, Eun-Kyung Lee, Heather Babka, Eve Wurtele

### Abstract

The datasets being produced by high-throughput biological experiments, such as microarrays, have forced biologists to turn to sophisticated statistical analysis and visualization tools in order to understand their data. We address the particular need for an open-source exploratory data analysis tool that applies numerical methods in coordination with interactive graphics to the analysis of experimental data. The software package, known as exploRase, provides a graphical user interface (GUI) on top of the R platform for statistical computing and the GGobi software for multivariate interactive graphics. The GUI is designed for use by biologists, many of whom are unfamiliar with the R language. It displays metadata about experimental design and biological entities in tables that are sortable and filterable. There are menu shortcuts to the analysis methods implemented in R, including graphical interfaces to linear modeling tools. The GUI is linked to data plots in GGobi through a brush tool that simultaneously colors rows in the entity information table and points in the GGobi plots.

ExploRase is an R package publicly available from Bioconductor (<http://www.bioconductor.org/>) and is a tool in the MetNet platform (<http://www.metnetdb.org/>) for the analysis of systems biology data.

## 1 Introduction

In recognition of the need for biologists to analyze multidimensional, high-throughput data, we have developed a software tool with the following goals:

- Support exploratory analysis of experimental data and networks through the integration of numerical methods and interactive graphics.
- Leverage biological information in the analysis of experimental data.
- Provide a graphical user interface (GUI) on top of statistical methods so that they are accessible to biologists.

High-throughput experiments have become commonplace in biology. The popular microarray measures the levels of tens of thousands of gene transcripts at once. GC-MS and other analytical methods currently have the potential to detect hundreds of metabolite levels, providing a snapshot of the metabolism in a cell. The exploRase package has been developed for analyzing datasets with measurements on tens of thousands of biological entities, such as genes or metabolites. The exploRase interface and analysis algorithms are designed for experiments with up to approximately 50 samples (columns in the experimental data matrix).

The untargeted nature of high-throughput experiments pairs well with an approach to data analysis that remains open to the unexpected and allows the analyst to form hypotheses during analysis. The intent is to facilitate the search for interesting features in the data, such as differentially expressed genes or metabolites that follow a similar pattern. This approach is known generally as exploratory data analysis and is the main philosophy behind the design of our software tool.

Interactive graphics are generally useful in exploratory analysis, and they are particularly applicable to analyzing experimental data for several reasons:

- Experimental datasets are multivariate and so benefit from the ability to view different combinations of variables in different ways, simultaneously.

- Results from numerical methods can be interpreted and validated by relating them back to the original data through visual cues.
- Linked interaction can relate individual measurements to the biological system by integrating plots of experimental data with drawings of biochemical networks.
- The network structure itself could be used to specify the linking between plots, possibly from different data sources, according to interactions in the biological system.
- The size and complexity of a typical biochemical network requires readjusting the network drawing during data analysis to expose and focus on particular characteristics of the network.

There is a number of software projects for analyzing data from high-throughput biological experiments. One of these is Bioconductor [Gentleman et al., 2005], a free collection of R packages [R Development Core Team, 2005] that provide numerical and graphical methods for helping biologists comprehend their data. There are packages in Bioconductor for analyzing and visualizing networks and various types of experimental data, including microarray and mass spectrometry data.

While R and the Bioconductor project provide sufficient support for numerical and general graphical methods, the R graphics system is not designed for interactivity. A specialized application for multivariate interactive graphics, GGobi [Swayne et al., 2003a], has been interfaced with R through the package rggobi [Temple Lang, 2001b]. GGobi provides interactive scatterplots, parallel coordinate plots, barcharts and other types of displays. Edges may be displayed in scatterplots in order to draw networks. Interaction modes include linked brushing between plots by identifier and categorical variable, point and edge querying, and pan/zoom.

The complementary nature of R/Bioconductor and GGobi encourages their combined application to the analysis of biological data. However, this is hindered by their lack of accessibility to biologists. All R packages, including those in Bioconductor, are driven through the R language, which facilitates their application to a wide variety of data analysis tasks. The flexibility and expressiveness of a script-driven interface is a double-edged sword, however. Bi-

ologists unfamiliar with programming and command-line interfaces struggle to take advantage of R and Bioconductor packages that lack a GUI. GGobi is designed to be flexible and open-ended, with the goal of supporting a wide range of analyses. However, this generality means that the biologist receives no biology-specific guidance during data analysis and visualization tasks.

We have developed a GUI-driven R package named *exploRase* for the graphics-intensive exploratory analysis of biological experimental data and networks. It is a tool in the MetNet platform [Wurtele et al., 2003, 2007] for systems biology data analysis and is publicly available from Bioconductor as of Bioconductor 2.1.

The *exploRase* package has the following general features:

- A collection of numerical methods, such as distance measures and linear models, implemented in the R language [R Development Core Team, 2005].
- Linked data plots and network drawings based on the GGobi tool for multivariate interactive graphics [Swayne et al., 2003a].
- A GUI, designed in collaboration with biologists, that integrates numerical methods with graphics and provides general features such as the loading and subsetting of data.

Section 2 presents an overview of the software and the next section describes the numerical and graphical methods it provides. The layout of its GUI is detailed in Section 4. This is followed by a tutorial that guides the user through a hypothetical analysis. Finally, the paper will conclude by discussing the future of *exploRase*.

## 2 Overview

As its name suggests, *exploRase* is designed to facilitate the exploratory analysis of biological data by combining the numerical methods of R (and Bioconductor) with the graphical methods of GGobi. Most users will only interact with *exploRase* through its GUI, and the GUI and plots of GGobi. The *exploRase* GUI is shown in Figure 1.

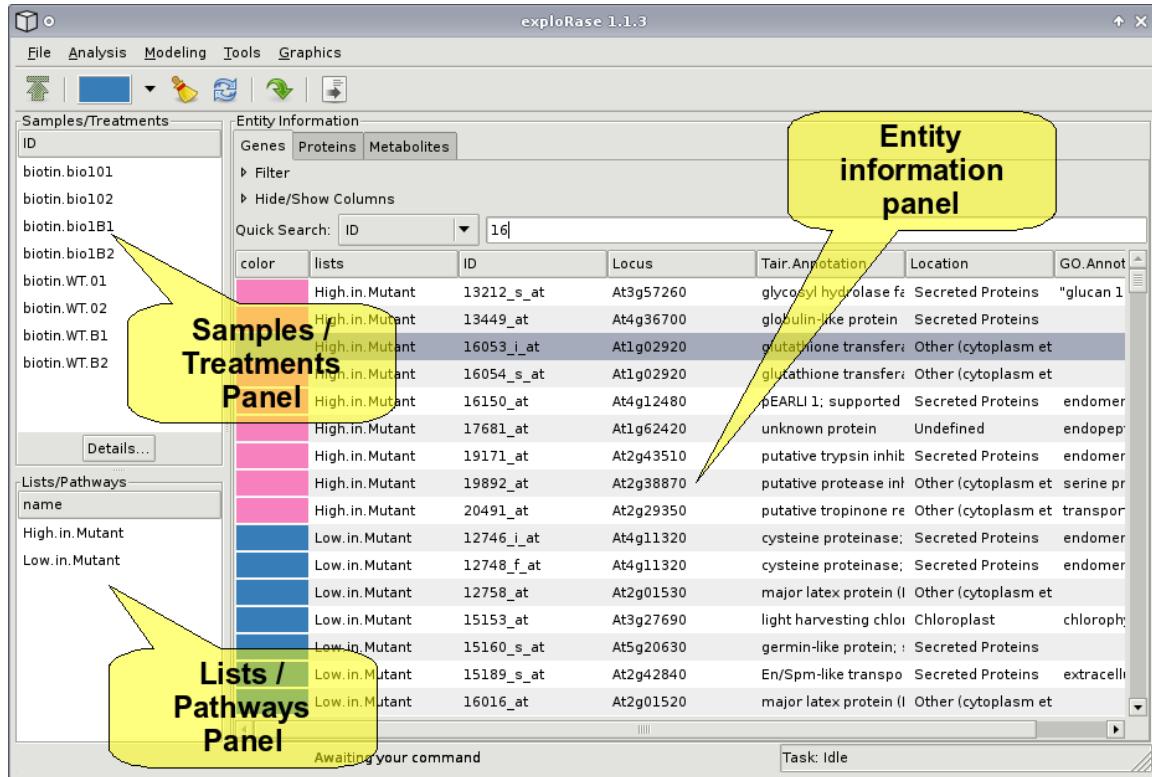


Figure 1 The main GUI of exploRase with the three main panels denoted by the yellow blurbs. The *Entity information* panel, located on the right in the GUI, is dominated by the entity information table, which contains metadata and analysis results for each entity in the experiment. Above the table is a quick search bar and tools for filtering the rows and columns of the table. To the left of the *Entity information* panel are two panels. The top one is the *Samples/Treatments* panel and lists the samples, such as microarray chips, in an experiment. Below is the *Lists/Pathways* panel, which contains the names of user-created entity lists. At the top of the GUI are the toolbar and menubar with options for loading and saving the data, analyzing the data and performing other functions. Please note that the appearance of the GUI (especially the button icons) depends on the GTK+ theme, so there may be some aesthetic differences between the screenshots in this paper and a user installation of exploRase.

The user begins an exploRase session by loading information describing one or more experiments. This includes the actual experimental measurements, as well as metadata and other supporting information. Each type of information is described below:

**Experimental data matrix** Measurements of the levels of transcripts, metabolites or some other biological entity for every sample in the experiment.

**Experimental design matrix** Description of experimental design, factors such as genotype or time. Each sample is labeled by the factor levels of the experimental design.

**Entity annotation matrix** Annotations, such as GO terms, of biological entities measured in the experiment.

**Entity list** User-defined lists of entities of interest, such as lists of metabolites in a common pathway or co-expressed genes.

**Network** Biological network model expressed as SBML [Hucka et al., 2003].

While no particular type of information is required to run exploRase, each feature has specific data requirements. For example, the linear modeling features require an experimental design matrix and the experimental data.

The data from a set of related experiments are organized into a *project*. The user creates a project by importing individual data files into an empty exploRase session. In future sessions, the user only needs to load the project to begin the analysis. Only one project may be loaded per session, and a project may contain at most one experiment for each type of biological entity. The entity types built into exploRase are genes, metabolites and proteins; expert users may define custom types through the R command line interface. The distinction between entity types is currently only for organizational purposes, though it may gain meaning in the future.

The files in a project are physically organized into a directory (folder) in the file system. The user need only specify the directory to load every file (i.e., experimental data files and metadata files), in the project at once. With the exception of the SBML network files, the format of every file is CSV, which is compatible with most spreadsheet applications. The type

of information contained in a file and the type of biological entity the information describes are indicated by the filename extension.

Once a dataset and related information is loaded into exploRase, the user may proceed with the analysis by performing operations such as:

- Subset the observations by criteria such as the minimum fold change and maximum variance across replicates.
- Check the quality of the data with GGobi graphics, such as scatterplots, scatterplot matrices and parallel coordinate plots.
- Browse entity metadata and analysis results in a filterable, searchable and sortable table.
- Color rows in the entity information table and the corresponding points in GGobi plots using the brush tool.
- Detect differentially expressed genes through the graphical interface to the limma package [Smyth, 2005] and view their profiles in a GGobi parallel coordinate plot.
- Find patterns in the data through hierarchical clustering and the explicit pattern query tool.
- Export analysis results and lists of interesting entities as CSV files.

### 3 Methods

#### 3.1 Numerical methods

The numerical methods in exploRase are designed to assist in finding biological entities with patterns that depend on experimental conditions or are similar to the pattern of a given entity or user-specified pattern. It is also possible to cluster entities according to a selected distance measure.

### 3.1.1 Finding entities with interesting patterns

**Two-sample comparison** One means of finding interesting patterns with exploRase is to compare two replicates or replicate means by a selected distance measure. The supported distance measures are difference, residuals from regressing one condition against the other, angle between the diagonal and the line from the origin to the point in the scatterplot of the two variables, and the Mahalanobis distance (e.g., Johnson and Wichern [2002]). Of these, the difference is the simplest and the most often applied to transcriptomics data. A distance measure may be used to check the agreement between two replicates or to evaluate differences in entity levels between treatments using replicate means.

**Linear modeling** Linear modeling is a more sophisticated technique for estimating the effects of experimental conditions on each entity. The exploRase GUI includes an interface to limma [Smyth, 2005] that fits a linear model to each entity and shrinks the variance using empirical Bayes analysis to yield p-value estimates that tend to correspond to what a biologist considers interesting: entities with relatively high basal levels as well as significant difference across conditions. The significance of each user-selected experimental design factor, as well as their interactions, is estimated without the user needing to manually specify any contrasts. The output of the limma tool may consist of the raw p-values, p-values corrected by FDR or another method, the corresponding F statistics, the contrast coefficients and the fitted values. The limma interface has an advanced feature for fitting contrasts that evaluate whether an entity pattern is linearly or quadratically dependent on time in time-course experiments.

There is also a separate polynomial model for estimating the effect of time. Its critical difference from limma is that it accounts for the order of the time points and thus is likely more realistic for time-course experiments. The user also can apply this polynomial model to evaluate interaction effects between time and other factors. The output contains the p-value and coefficient for each effect, as well as the F statistic and the sum of squares error for the overall model.

### 3.1.2 Searching for entities with specific patterns

**Matching the pattern of a specific entity** The exploRase package offers several distance measures for comparing entity patterns. These include: cosine angle (uncentered correlation), Euclidean, Pearson correlation and Canberra (e.g., Johnson and Wichern [2002]). The Euclidean distance is usually not of interest, as it is based on the magnitude of the levels, not their pattern; however, it may be useful for finding entities that are present at similar levels over the course of the experiment. The Pearson correlation disregards magnitude, so it may be useful for identifying entities with similar patterns regardless of their levels. This could, for example, help identify gene regulatory interactions, where the increase (or decrease) in one transcript results in the decrease (or increase) in another. In contrast, the cosine angle distance considers both pattern and magnitude and thus may be useful if the biologist wishes to focus on entities present at similar levels while still considering the pattern. If the level of the query entity is relatively high (i.e. above background noise), the cosine angle measure reduces the number of hits against entities that are present only at low (i.e. background) levels and may thus be considered uninteresting. The Canberra distance may be particularly useful for metabolomics data as it is numerically stable when faced with zero values, which result from the common practice of imputing non-detects as zeroes.

**Matching a user-specified pattern** The user may also search for an explicit pattern by specifying “Up”, “Down” or “Same” for each transition between adjacent samples in a user-specified list. The entity patterns are matched to the query by denoting a transition as “Up” if the transition value (difference between the pair of samples) is above the  $q$  quantile and “Down” if the value is below the  $1 - q$  quantile. Everything between the quantiles is marked as “Same”. The parameter  $q \in [0, 1]$  is specified by the user with a slider.

### 3.1.3 Clustering of entities

For clustering the observations, exploRase performs agglomerative hierarchical clustering using Ward’s linkage method (e.g., Johnson and Wichern [2002]). The user can choose from

the several distance measures: cosine angle, Euclidean, Pearson correlation and Canberra. These are the same distance measures described for comparing entity patterns against a query entity pattern. A very similar question is asked when clustering, so the same criteria apply for selecting a distance measure. One of the major differences between comparing entity patterns and clustering is that in clustering the distance is calculated between every pair of entities rather than between one entity and the others. Given the size of high-throughput datasets, it is computationally intensive to calculate the pairwise distance for every entity in the experiment, and the results can be difficult to interpret. Thus, it is recommended that the user select a small subset of entities for clustering. ExploRase supports efficient clustering of up to approximately 50 entities.

### 3.2 Graphical methods

The graphics in exploRase are interactive and designed for exploratory data analysis. The data plots are displayed by GGobi.

There are two general types of visualizations in exploRase: data plots and network drawings. Both are based on GGobi.

#### 3.2.1 Data plots

For visualizing the experimental data, exploRase provides scatterplots, scatterplot matrices, parallel coordinate plots and histograms.

**Scatterplots** The scatterplot, or the basic X vs. Y plot, is the most generally applicable exploRase plot type. For example, the user may compare two samples, such as a pair of replicates, or two conditions, averaged over the replicates. In this way, the scatterplot is the graphical equivalent of distance measures for comparing the levels of an entity between two samples or conditions. The scatterplot is also useful for interpreting analysis results. For example, comparing the p-value for a limma effect with its coefficient, as in Figure 2, is particularly effective for detecting patterns that are both significant and of high amplitude.

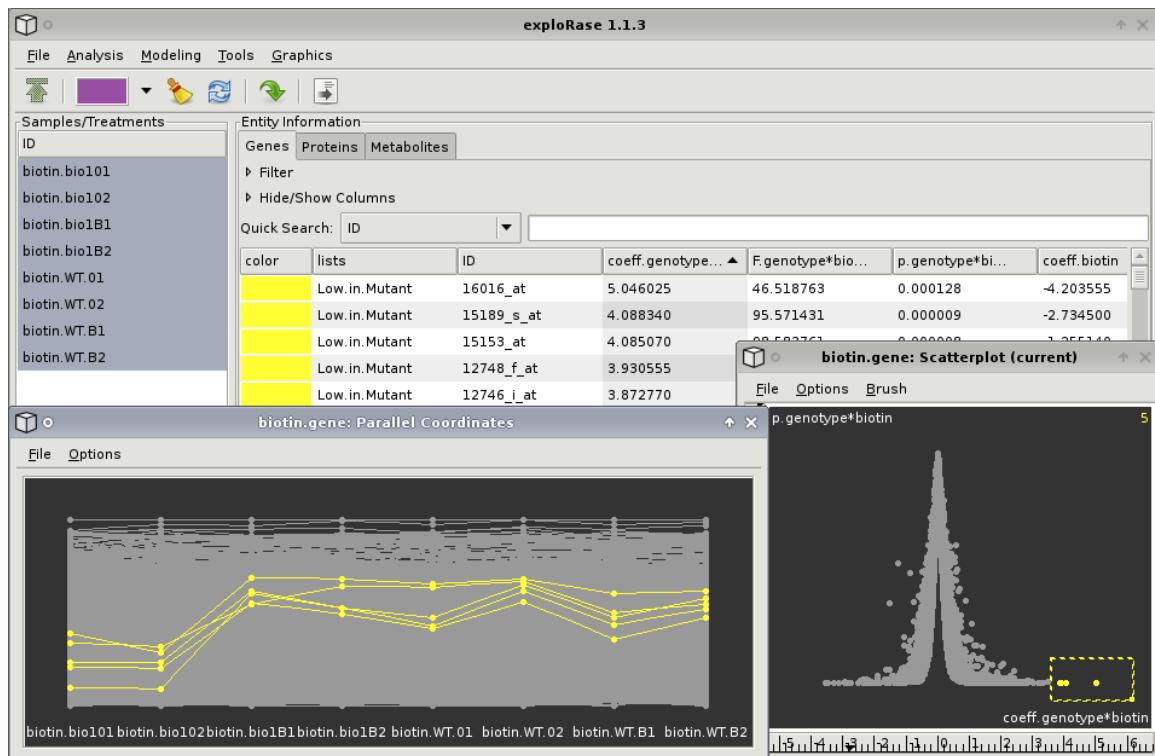


Figure 2 Visualizing and validating Limma results with exploRase. The dataset is from the microarray experiment described in Section 6. On the right is a scatterplot of p-value versus coefficient for the genotype contrast fit by Limma. By highlighting a point in the scatterplot, the corresponding profile in the parallel coordinate plot on the left is also highlighted.

**Scatterplot matrices** The scatterplot matrix is a symmetric grid of scatterplots. Every row has the same variable on the vertical axis and every column shares the variable on the horizontal axis. Along the diagonal, the X and Y variables are the same, so histogram is displayed for that variable. Scatterplot matrices are useful for obtaining an overview of the data, because multiple variables are compared at once. One common application is the comparison of replicate sets during data quality checking.

**Parallel coordinate plots** The parallel coordinate plot displays a profile of each entity across a sequence of variables, normally the samples or replicate averages. Due to the large number of entities in microarray, proteomics and metabolomics experiments, the profiles tend to be overplotted to the extent that individual profiles are not recognizable (e.g., the grey profiles in the parallel coordinate plot in Figure 2). Linked brushing helps to overcome to this problem. For example, as shown in Figure 2, the user could brush outlying points in the scatterplot of p-value versus coefficient to highlight the corresponding profiles in the parallel coordinate plot. This permits the visual validation of patterns deemed significant by numerical methods.

**Histograms** The interactive histogram is another tool for interpreting analysis results and checking data quality. The user can view profiles that are outliers with respect to a statistic by brushing outlying points in the histogram of the statistic. For example, assume the user has brushed a particular gene of interest and wishes to view profiles that are similar to that of the chosen gene. The user could calculate a distance measure between the gene and the others and compare profiles in a parallel coordinate plot by brushing the outlying points (with a new color) in a histogram of the distances.

### 3.2.2 Network drawings

The exploRase biochemical network visualization has three goals:

1. Permit the user to adapt the layout to answer particular questions.

2. Communicate biological information in the network by considering biological aesthetics in the layout.
3. Integrate the drawing with experimental data through linked interaction with data plots.

Full details regarding the biology-specific interactive network visualization in exploRase are given in Chapter [biocola](#). The system is summarized below.

To accomplish the above goals, exploRase leverages the rcola package, which supports interactive network visualizations based on an incremental, constraint-based layout algorithm. Constraints may be applied to sets of nodes in order to arrange them in a way that helps answer a particular question. The incremental nature of the algorithm limits disorienting changes to the layout as constraints are added and removed. Transitions between layouts may be animated to help preserve the “mental map” of the user [Eades et al., 1991].

With the aim of communicating biological information in a network, exploRase provides two biology-specific constraints for the rcola layout algorithm. A synopsis of each constraint is given below:

**Reaction Constraint** Clarifies a reaction by segregating the reactants and products on opposite sides of the reaction node and aligning the catalyst orthogonally to the reactants and products (see Figure 2).

**Compartment Constraint** Communicates compartment information by segregating nodes into disjoint convex regions of the drawing according to their compartment attribute.

The biological information necessary for the constraints is obtained from the SBML description [Hucka et al., 2003] of the network loaded into exploRase. Thus, this visualization is limited to networks that may be described in SBML.

The generated layout is displayed in a GGobi scatterplot with nodes drawn as points and edges as straight lines. A GUI is associated with the scatterplot for controlling the layout algorithm. Through the GUI, the user can specify a constraint and apply it to the selected nodes in the scatterplot. By applying the appropriate combination of constraints, the user may achieve the appropriate layout for answering the question at hand.

## 4 GUI features

The main GUI of exploRase, shown in Figure 1, consists of three panels (*Entity information*, *Samples/Treatments* and *Lists/Pathways*), a toolbar and a menubar. The primary design considerations for the GUI are simplicity and usability. There is no attempt to completely map the features of the underlying tools to the exploRase GUI. Rather, the GUI supports a subset of the features most useful in the analysis of high-throughput transcriptomics, proteomics and metabolomics data and augments this subset with shortcuts and conveniences for biological data analysis. The GUI has been designed in collaboration with biologists, to help ensure that exploRase is accessible to those who are performing the experiments and generating the data.

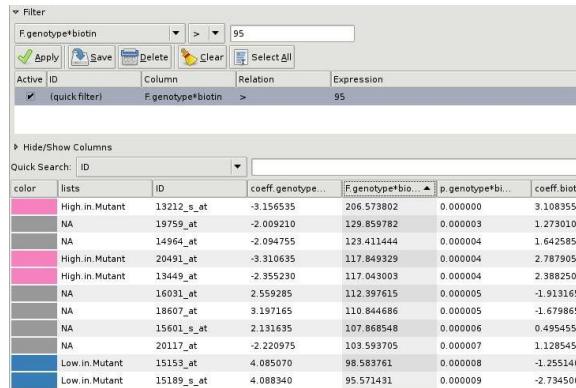
### 4.1 Main panels

#### 4.1.1 Entity information panel

The largest panel contains the entity information notebook, which has a tab for each entity type of interest. The default entity types are genes, proteins, and metabolites; new types are easily added by expert users through the exploRase R API. Each tab contains a table containing entity information (metadata and analysis results), an expandable panel for filtering the table rows, an expandable panel for hiding or showing table columns and an entry box for searching the table.

**Entity information table** The table has a row for each biological entity in the experimental data. The columns of the table contain metadata, such as biological function and biochemical pathway membership. There are two special built-in columns at the left. The first displays the color of the entity chosen by the user. This color matches the color of the glyphs for the entity in the GGobi plots. The other column indicates the user-defined entity lists to which the entity belongs. The table may be sorted according to a particular column by clicking on the header for the column.

**Filter** A filtering component, shown in Figure 3, is made visible by clicking on the *Filter* label above the table in the *Entity information* panel. This filters the entity information table, as well as the GGobi plots, by any column in the table. Columns containing text data may be filtered according to whether a cell value equals, starts with, ends with, contains, or lacks a user-input text phrase. Regular expression matching [Friedl, 2006] is also supported. Numeric values may be tested for being greater than, less than, equal to, or not equal to a user-input number. When filtering by color, the user may choose from the current palette of colors. It is also possible to filter by entity list membership, so that only the entities that belong to a specified list are included in the table. After applying a rule, it may be saved. The saved rules are displayed in a table below the rule editor. There is a checkbox in each row that toggles the activation state of the rule. Buttons allow the deletion of selected rules and the batch activation and deactivation of every rule. Only those cases that pass the intersection of all active filter rules are displayed in the entity information table and the GGobi plots. After filtering, the user might select all the visible entities and save them to an entity list for future recall.



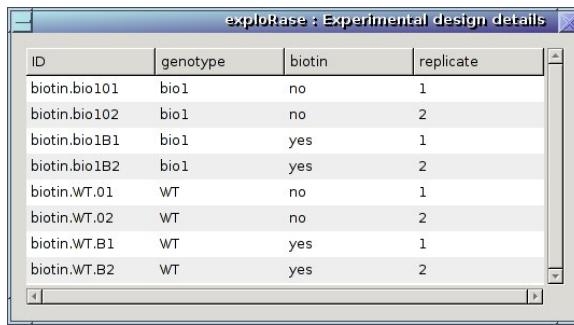
The screenshot shows the exploRase filter GUI. At the top is a filter dialog with fields for 'Column' (F.genotype\*biotin), 'Relation' (>), and 'Value' (95). Below the dialog is a table of entity information. The table has columns: color, lists, ID, coeff.genotype..., F.genotype\*biotin, p.genotype\*biotin, and coeff.biotin. The data rows are colored pink for High.in.Mutant and blue for Low.in.Mutant. The table shows various gene IDs and their corresponding values across the columns.

color	lists	ID	coeff.genotype...	F.genotype*biotin	p.genotype*biotin	coeff.biotin
High.in.Mutant		13212_s_at	-3.156535	206.57802	0.00000	3.108355
NA		19759_at	-2.009210	129.859782	0.00003	1.273010
NA		14964_at	-2.094755	123.411444	0.00004	1.642585
High.in.Mutant		20491_at	-3.310635	117.849329	0.00004	2.787905
High.in.Mutant		13449_at	-2.355230	117.043003	0.00004	2.388250
NA		16031_at	2.559285	112.397615	0.00005	-1.913165
NA		18607_at	3.197165	110.844686	0.00005	-1.679865
NA		15601_s_at	2.131635	107.868548	0.00006	0.495455
NA		20117_at	-2.220975	103.593705	0.00007	1.128545
Low.in.Mutant		15153_at	4.085070	98.583761	0.00008	-1.255140
Low.in.Mutant		15189_s_at	4.088340	95.571431	0.00009	-2.734500

Figure 3 The exploRase filter GUI that expands above the entity information table. The active filter rule accepts only the genes with a *F.genotype\*biotin* value greater than 120.

**Hide/show columns** Clicking on the *Hide/show columns* label in the *Entity information* panel expands a component that lists the column names of the entity table and allows the user to specify whether each column is hidden or shown. This helps keep the table clean when generating many columns holding analysis results.

**Quick search bar** Just above the entity information table is a *quick search* bar to search individual columns. The user selects the column to search in the pull-down menu on the left and enters a query in the text box. As the user types, the table scrolls to the first row that matches the query.



The screenshot shows a Windows-style application window titled "exploreBase : Experimental design details". Inside, there is a table with four columns: "ID", "genotype", "biotin", and "replicate". The data rows are as follows:

ID	genotype	biotin	replicate
biotin.bio101	bio1	no	1
biotin.bio102	bio1	no	2
biotin.bio1B1	bio1	yes	1
biotin.bio1B2	bio1	yes	2
biotin.WT.01	WT	no	1
biotin.WT.02	WT	no	2
biotin.WT.B1	WT	yes	1
biotin.WT.B2	WT	yes	2

Figure 4 The experimental design table, with a column for each factor and a row for each condition.

#### 4.1.2 Sample/Treatments panel

**Sample/Treatments list** To the left of the *Entity information* panel are two panels. The upper one lists the biological samples, such as chips for a microarray experiment, that are in the experimental data. The user may select samples from the list in order to limit the scope of the analysis. It is possible to select a range of samples by holding down the SHIFT key and clicking on the end-points of the range. This is particularly useful after the list has been sorted by an experimental factor using the experimental design table described in the next paragraph.

**Experimental design table** Clicking on the *Details* button below the *Sample/Treatments* list displays a table describing the experimental design, as shown in Figure 4. There is a column for each factor in the experiment, and the rows correspond to conditions. Similar to the *Entity information* panel, clicking on a column header sorts the table, as well as the *Sample/Treatments* list in the main GUI, by that column.

#### 4.1.3 Lists/Pathways panel

The bottom panel contains user-defined entity lists. These lists store a group of user-selected entities, usually based on the result of an analysis. Selecting an entity list automatically selects the corresponding entities from the list in the entity information tables. The selected entities may then, for example, be brushed or sent as a query to AtGeneSearch, as described in Section 4.2.

### 4.2 Toolbar

Above the main GUI panels is the toolbar (Figure 1), which contains many important buttons. We will describe the buttons from left to right. The button with the folder icon provides a shortcut for loading a project. Perhaps the most important button is the brush tool, which appears as a rectangle filled with the current brush color. Clicking on the brush button colors the selected entities in the currently visible entity information table and the same entities in the GGobi plots. The color is selected from a palette that drops down from the button. Besides the brush button are two more brush-related buttons: the first for resetting the colors to the default (gray) and the other for updating the colors in the entity information table to match those in GGobi. The next button to the right, shown in the screenshot as a single curved arrow, queries AtGeneSearch, a web interface to MetNetDB for accessing additional metadata about the selected entities [Wurtele et al., 2003, 2007]. AtGeneSearch provides links to other web data sources. The right-most button creates an entity list from the entities selected in the entity information tables and adds the nascent list to the *Lists/Pathways* panel.

## 4.3 Menubar

### 4.3.1 File menu

At the top of the main exploRase GUI is the menubar (Figure 1). The *File* menu provides options for loading and saving files and projects. The *Open* item is for loading already-created projects. To load individual files and merge them into an opened project the user may choose the *Import File(s)* item and select the files using a dialog. The *Save* item saves the entire project to a user-specified directory. The user may select an item from the *Export* submenu to save an individual project component, such as the entity information or a selected entity list, to a CSV file.

### 4.3.2 Analysis menu

The *Analysis* menu lists a collection of numerical analysis methods, as described in 3.1. The first set of methods consists of distance measures for comparing the levels of each entity between two samples or conditions. The next set of methods are distance measures for comparing a selected entity against the rest. The final two methods are hierarchical clustering and pattern finding. The cluster results are displayed in an interactive R plot, shown in Figure 5; clicking on a branch point of this R plot brushes the descendent entities. The results of each analysis are added as a column in the entity information table and as a variable in GGobi. This allows the user to sort and filter according to the results of the statistical analyses, as well as visualize these results in GGobi.

**The pattern finder** A unique feature of exploRase is the pattern finder, which calculates whether an entity is significantly rising or dropping relative to the others for each sample transition. The results are displayed as arrows embedded in the entity information table, as shown in Figure 6. The dialog named *Find Patterns* allows the user to query for specific patterns. To specify a query pattern, the user chooses whether a particular transition should be “Up”, “Down” or the “Same” (these match the terms used by the *Find Patterns* dialog). The vertical slider on the left of the dialog specifies the number of entity transitions, centered

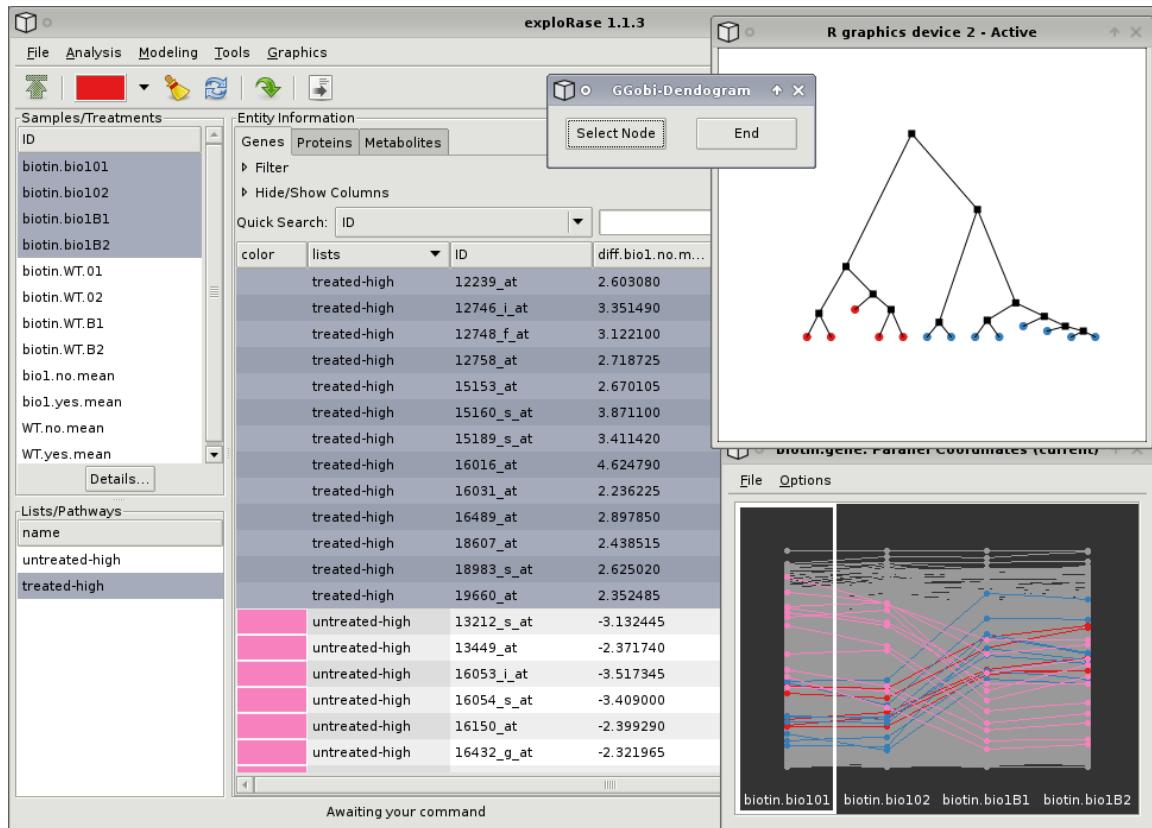


Figure 5 The hierarchical cluster browser. Clicking on a node selects its children in the exploRase information table. In this screenshot, the “treated-high” genes (originally all colored blue) have been clustered according to the mutant chips. The left child of the root has been clicked, resulting in the coloring of five genes in exploRase (and GGobi) with the current brush color (red). It appears from the parallel coordinate plot that the patterns of the red genes have more in common with each other than with the blue genes.

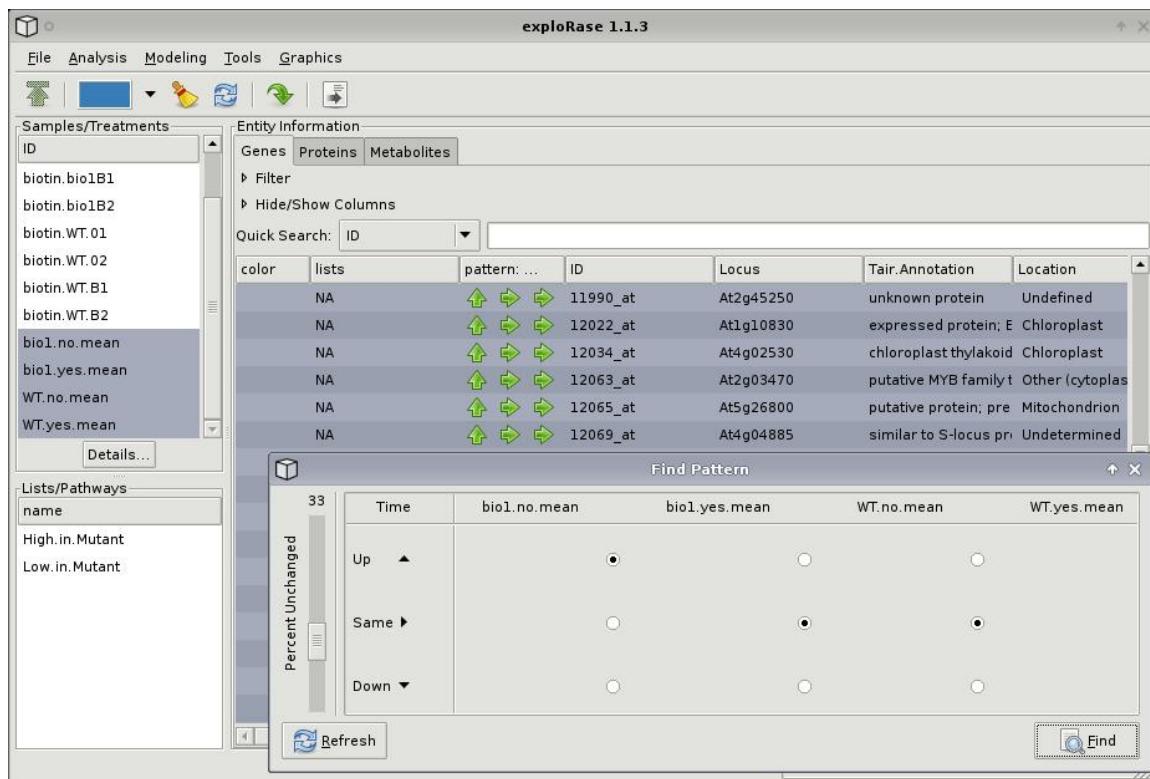


Figure 6 The pattern finder. The calculated patterns across a time course experiment are shown in the “pattern...” column as arrows representing the direction of each transition. The pattern finder dialog selects rows that match the specified pattern. Here, a constantly increasing pattern has been selected. The vertical slider on the left of the dialog specifies the number of entity transitions, centered on the median, that are considered to stay the “Same”. All transitions less than the assumed “Same” transitions are considered “Down” and the remaining transitions are considered “Up”.

on the median, that are considered to stay the “Same”. All transitions less than the assumed “Same” transitions are considered “Down” and the remaining transitions are considered “Up”. When the *Find* button is clicked, the entities with matching patterns are selected in the entity table.

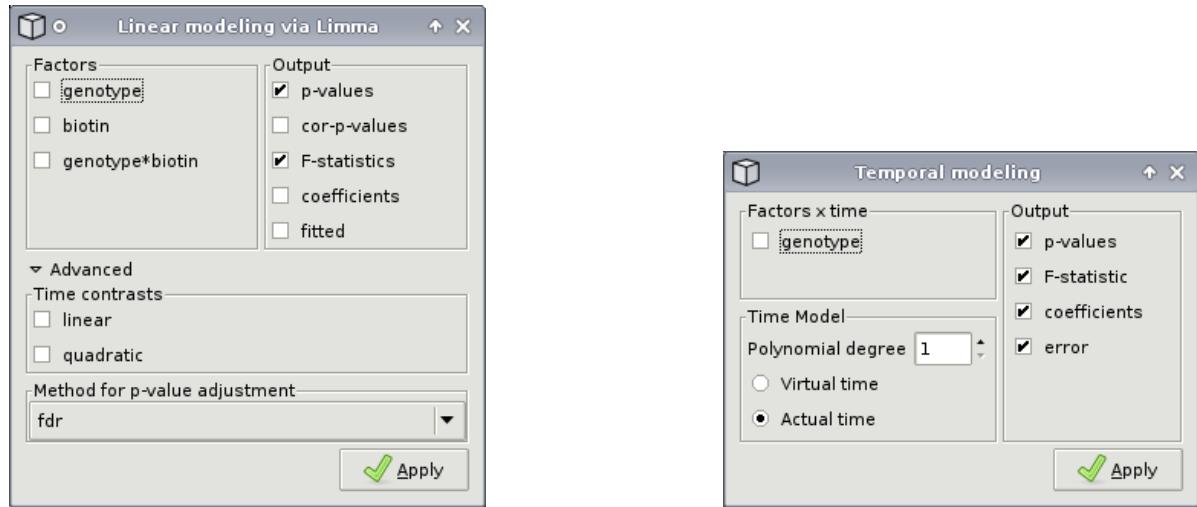


Figure 7 Linear modeling dialogs. On the left is the interface to limma, and on the right is the temporal modeling interface. The user may select the factors and outputs of interest, as well as specify various other parameters.

#### 4.3.3 Modeling menu

The *Modeling* menu launches graphical interfaces to linear modeling tools in R. Both interfaces are shown in Figure 7.

**Limma** The limma interface leverages the limma package [Smyth, 2005] from Bioconductor [Gentleman et al., 2005]. The interface prompts the user for the factors to include in the model, including interactions among factors. The user may also choose which results (p-values, corrected p-values, F statistics, coefficients, or fitted values) to include in the entity information table and GGobi. An *Advanced* drop-down offers additional options, such as the method for p-value adjustment and tests for time linearity.

**Temporal Modeling** An interface is also provided for time-course modeling. It fits a polynomial model in time. The user may define the degree of the time polynomial and choose whether the time variable should be treated as a quantitative variable (actual) or as an ordinal variable (virtual).

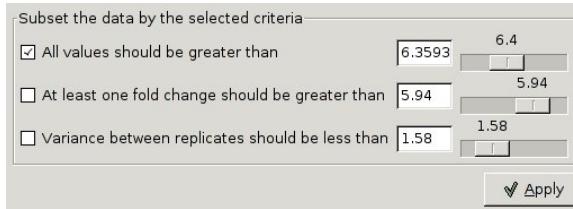


Figure 8 Simple subsetting GUI. The user may activate rules that filter entities based on their level, fold change, and replicate variance.

#### 4.3.4 Tools menu

The *Tools* menu contains methods for processing experimental data. There are items for calculating replicate means, medians or standard deviations and adding them to the data. The means and medians are automatically included in the list of experimental conditions, so that they may be used in numerical analysis. The second option launches the dialog shown in Figure 8 that provides several simple rules for filtering out entities based on the experimental data. The cutoffs are based on minimum value, minimum fold-change, and maximum variance between replicates. This helps the user focus on particular aspects of the data, such as entities that are changing more between treatments than within. The user may enter the test values directly or use the slider to get some idea of the range of values.

## 5 Getting started

The first step towards analyzing data with exploRase is to load the data. The exploRase package has not been designed for data preprocessing, so all preprocessing must be done before loading data into exploRase. Usually this involves steps like normalizing and log transforming the data. All files read by exploRase must adhere to the comma separated value (CSV) format,

as interpreted by the R CSV parser. This format is compatible with the output of Bioconductor tools and the CSV export utility of Microsoft Excel. Accordingly, the file containing the matrix of experimental measurements must be formatted as CSV, with the values from each sample (i.e. chips in a microarray experiment) stored as a column. The first row should hold the names of the corresponding samples. The first column, which does not require a name in the first row, should hold unique ids for each biological entity (transcript, protein, etc.) measured in the experiment.

In addition to the experimental measurements, exploRase supports (and, for some features, requires) several types of metadata, all formatted as CSV. The experimental design matrix is required for linear modeling and aggregating replicates. Like the experimental data, the first row should name the design factors, such as *genotype*, *time*, and *replicate*. Some factor names have special meaning. In particular, *time* is used as a factor in the temporal modeling tool and *replicate* is used in linear modeling and averaging over replicates. Each cell in the first column of the design matrix should match one of the sample names in the experimental data.

Another type of metadata is the entity annotations that are shown in the central table of the exploRase GUI. The only restriction is that the first column should hold entity identifiers that match those of the experimental data.

Finally, entity lists are stored as one or two column matrices. If two columns are present, the first column is interpreted as the type of the entity, such as *gene*, *prot*, or *met*. This allows storing entities of different types in the same list. The other column holds the identifiers of the entities that belong to the list. The name of that column is the name of the list in the exploRase GUI.

In order to automatically detect the type of data being loaded, exploRase expects the input files to be named according to a specific convention. The mapping from data type to filename extension is given in Table 1. The user must ensure that the input files are named according to that convention.

The data loading process is further simplified by support for projects: all of the data files may be placed into an empty folder and loaded in a single step by choosing the folder in the

Type	File Extension (+ .csv)	Example
Transcriptomic Data	gene.data	mittler.gene.data.csv
Gene Information	gene.info	affy25k.gene.info.csv
Gene Exp. Design	gene.design	mittler.gene.design.csv
Metabolomic Data	met.data	suh-yeon.met.data.csv
Metabolite Information	met.info	suh-yeon.met.info.csv
Metabolite Exp. Design	met.design	suh-yeon.met.design.csv
Proteomic Data	prot.data	some-proteins.prot.data.csv
Protein Information	prot.info	some-proteins.prot.info.csv
Protein Exp. Design	prot.design	some-proteins.prot.design.csv
Interesting Entities	list	favorite-metabolites.list.csv

Table 1 Mapping from data type to filename extension per the exploRase filenaming convention. ExploRase requires project files to be named using these file extensions. An example project with correctly formatted files is in the supplemental data.

open project dialog. The types of the files are determined by their file extension.

## 6 Demonstration

In order to briefly demonstrate the features of exploRase, we consider a microarray dataset from an experiment investigating the response of biotin-deficient *Arabidopsis* mutants to treatment with exogenous biotin [Cook et al., 2007]. The mutants were analyzed with and without biotin treatment. Wildtype plants were used as a control and there were two replicates for each set of conditions. Figure 4 summarizes the experimental design. The dataset was normalized using the RMA method.

The first step, after launching exploRase, is to load the data. One easy way to load data into exploRase is as a project. Projects are directories in the file system that contain the experimental data, design matrix, entity metadata, entity lists, etc, as files. A zip archive containing an example exploRase project for the biotin data, with correctly formatted files, is provided on the exploRase website [Lawrence, 2007c].

To load the project:

1. Click the *Open* button at the left-end of the toolbar (see Figure 1).

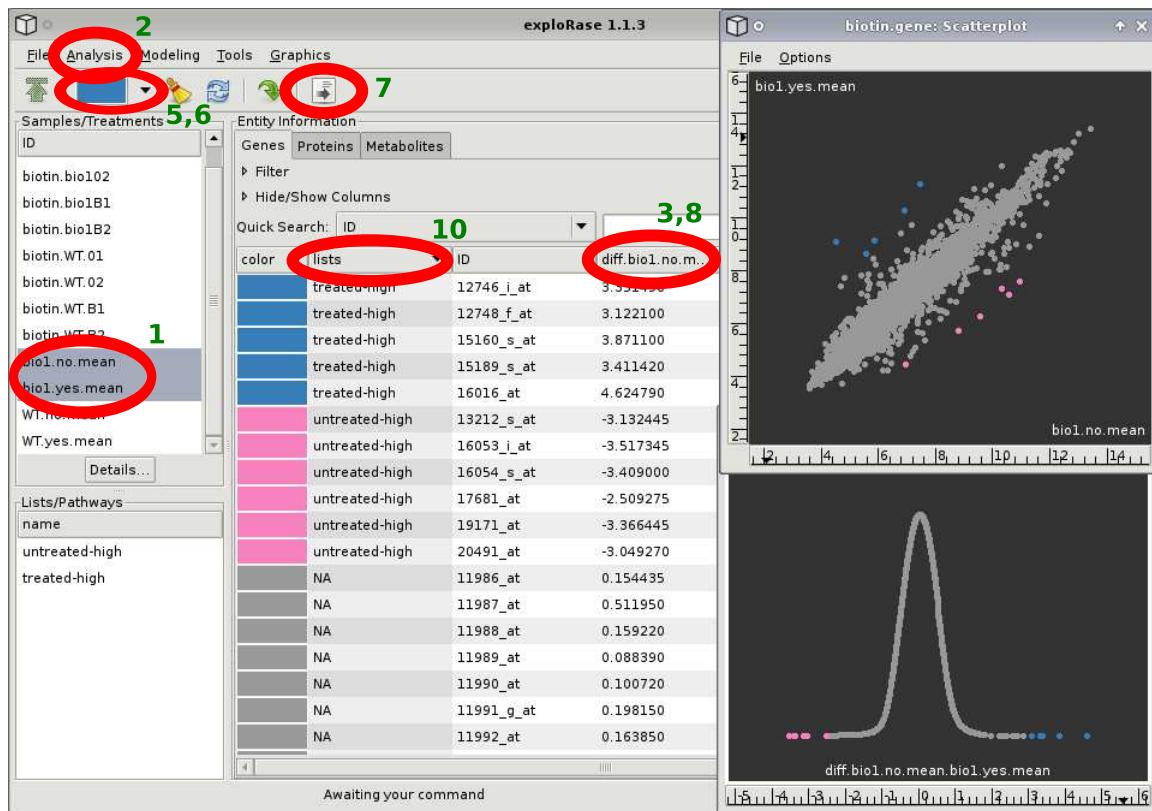


Figure 9 Difference calculation between the biotin mutant with and without external biotin. The GGobi scatterplot (above) compares the two conditions; below is a histogram of the difference calculation. The numbered red circles drawn on top of the screenshot illustrate the steps to calculate the differences, as explained in the example.

2. In the file open dialog, select the *biotin* directory from the (uncompressed) zip archive and click *Open*.

The primary goal of this example analysis is to determine which genes appear to respond to biotin treatment in the mutant. In order to compare across conditions without having to consider each replicate individually, the replicate mean values should be added to the experimental data, assuming that there are no major inconsistencies within the replicate pairs. To add the means to the data: choose the *Average over the replicates* option from the *Tools* menu.

Figure 9 displays the result of subtracting the untreated mutant mean from the treated mutant mean in the sample dataset. Sorting by the difference column in the information table allows the coloring of the selected extreme rows using the exploRase brush button. Alternatively, the user could brush the outlying points in the GGobi plots and then update the colors in the entity information table to match those in GGobi. The genes at each extreme are grouped into entity lists. The genes brushed in pink are those that have higher expression in the untreated plants compared to the treated, while the blue have lower expression in untreated plants.

To color and group the entities with the most extreme differences between treated and untreated mutant means, follow these steps (as illustrated in Figure 9):

1. Select *bio1.no.mean* and *bio1.yes.mean* in *Samples/Treatments* panel (use the **CTRL** key for multiple selections).
2. Open the *Analysis* menu in the menubar at the top of Figure 1. Choose the *Subtract* item from the *Find Difference (two conditions)* submenu.
3. Once the column containing the differences appears in the entity information table (the large table in the center of Figure 1), click on the column header (*diff.bio1...*) until the results are sorted in decreasing order.
4. Select a range of rows at the top of the entity information table (i.e. by holding down the **SHIFT** key).

5. Click on the downward-pointing arrow on the right side of the *Brush* button and select the blue color.
6. Click the *Brush* button to color the selected rows blue.
7. Click the *Create List* button in the toolbar and enter “treated-high” into the entry that appears in the *Lists/Pathways* panel at the bottom-left of Figure 1.
8. Click on the header of the difference column of the entity information table again to resort the rows of the entity table so that the rows are in increasing order.
9. Repeat steps 4-7 but use pink rather than blue as the brush color and enter “untreated-high” when creating the list.
10. To sort the rows in the entity table by their list membership, click on the header of the *List* column in the entity information table.

The GGobi scatterplot at the top-right of Figure 9 compares the two means, showing that the colored observations are indeed outliers. Below the scatterplot is a histogram showing the distribution of the difference.

To create GGobi plots follow these steps (as illustrated in Figure 10):

1. Use the GGobi control panel window (shown in Figure 10): select the *New Scatterplot Display* option from the *Displays* menu.
2. Select the two mean variables by clicking on the *X* button next to the *bio1.no.mean* label and the *Y* button next to the *bio1.yes.mean* label.
3. Select the *New Scatterplot Display* option from the *Displays* menu.
4. To change the scatterplot to an ASH plot (histogram), select the *1D Plot* from the *View* menu.
5. Click the *X* button next to the *diff.bio1...* variable, so that the histogram shows the distribution of the differences.

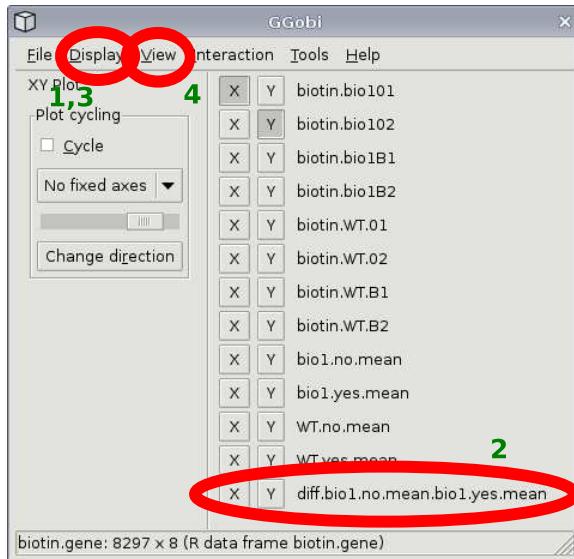


Figure 10 The GGobi control panel, used for creating and configuring GGobi plots. The *File* menu supports loading and saving of GGobi datasets, but the exploRase user is expected to load and save data through the exploRase GUI. The *Displays* menu contains items for opening each type of display, such as scatterplots and parallel coordinate plots. The *View* menu allows toggling between display view modes, such as histogram vs. XY plot. The *Interaction* menu has an item for activating each interaction mode, such as the brush. The *Tools* menu contains various utilities. Below the menubar are two panes. The left contains options for the current interaction mode. The other lists the variables in the current dataset and has toggle buttons for specifying which variables are plotted in the current display. The numbered red circles drawn on top of the screenshot illustrate the steps to create the plots mentioned in the example.

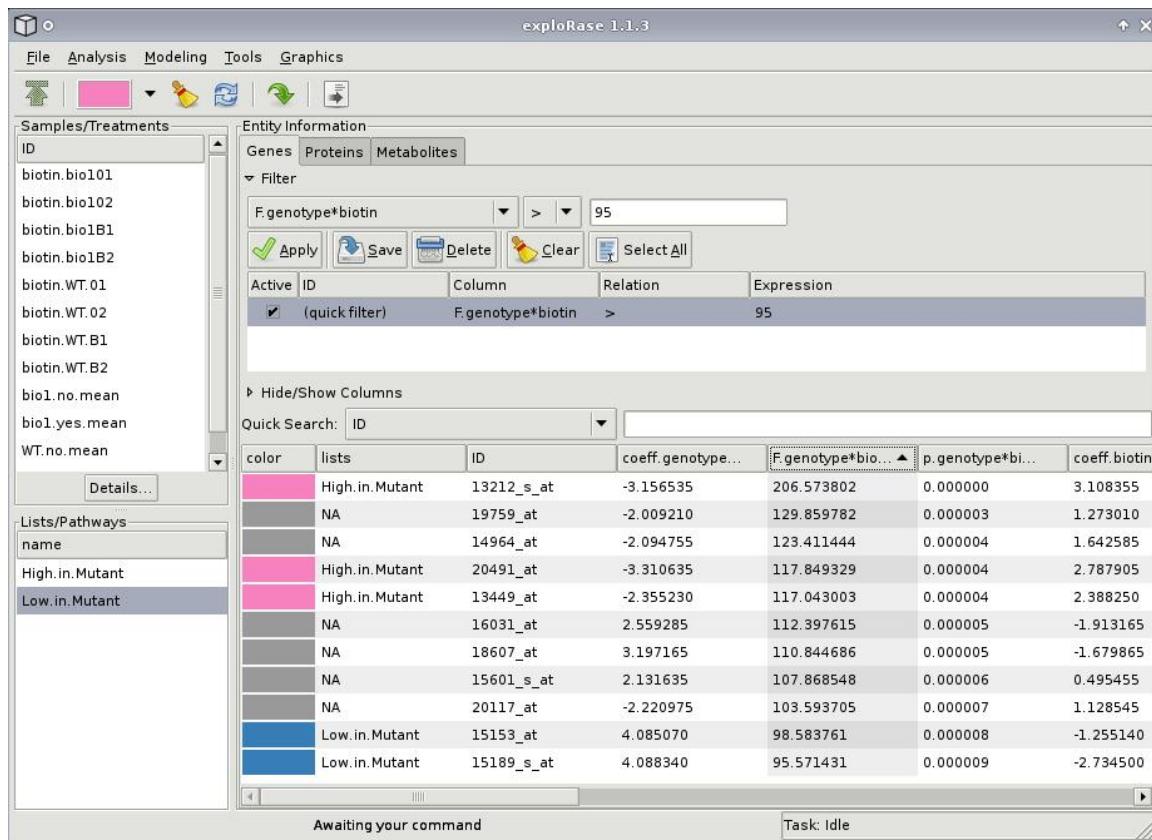


Figure 11 limma results for the biotin data. Only the entities with an F statistic  $\geq 95$  for the interaction of genotype and biotin are displayed. The numbered red circles drawn on top of the screenshot illustrate the steps to filter the table.

One way to verify that those genes are indeed dependent on biotin treatment would be to fit linear models using limma, including effects for the genotype, biotin treatment, and their interaction.

To do this, use the limma interface in exploRase:

1. Choose the *Linear modeling (limma)* option from the *Modeling* menu (in the menubar in Figure 1). This should open the *Linear modeling via Limma* dialog (shown on the left in Figure 7).
2. From the list of factors in the *Linear modeling via Limma* dialog, select the checkbox for the interaction of genotype and biotin. Note that this automatically selects the individual factors.
3. Click the *Apply* button to run limma.

Figure 11 shows the F values for the interaction of biotin and genotype. The table is sorted by the F value and filtered so that only the genes with the largest F values (F statistic  $\geq 95$ ) are included in the table. As one might expect, several of the pink- and blue-colored genes have extreme F values, indicating that biotin treatment has a genotype-dependent effect on those genes.

To identify those genes with the largest F values, follow these steps (as illustrated in Figure 11):

1. Click on the *Filter* label above the entity information table, so that the filter GUI is shown.
2. Select *F.genotype\*biotin* from the left-most drop-down menu in the filter panel.
3. Change the second combo box to  $\geq$ .
4. Enter “95” into the text field to the right.
5. Click *Apply* to apply the filter rule (*F.genotype\*biotin  $\geq 95$* ). Genes with an F value less than 95 are now excluded from the entity table.

6. Click the header of the *F.genotype\*biotin* column to sort by it.

One outlier is easy to recognize even from the table: 13212\_s\_at. The annotations in the table describe 13212\_s\_at as a glycosyl hydrolase. The functions of the other outlying genes, if known, may be found in the table, and if more information is needed, clicking the *AtGeneSearch* button in the toolbar spawns a web browser and queries the MetNetDB [Wurtele et al., 2003] for additional details.

This analysis could continue along many paths. For example, the user might search for genes that are similar to the 13212\_s\_at using the distance measures in exploRase (from the *Analysis* menu), or could continue to inspect the output of limma using GGobi graphics. Lists of genes could be exported to MetNet for pathway display (MetNet/Cytoscape) or evaluation in the context of public microarray data (MetaOmGraph) [Wurtele et al., 2007]. This example demonstrates only a fraction of the potential of exploRase.

## 7 Technical considerations

### 7.1 Suggested limit on number of samples

There are several reasons behind the suggestion that the number of samples (columns) in the experimental data be limited to a relatively small number (approximately 50 in our experience).

- The exploRase and GGobi GUIs are not designed for a large number of variables. In particular, it is cumbersome to navigate through and operate on the variable selection panels.
- The analytical methods in exploRase have not been designed/selected with a large number of samples in mind, though many of the same methods would still apply.
- The exploRase implementation (and, in general, R itself) has not been heavily optimized (in terms of space and time) for large numbers of samples nor extremely large data matrices.

The exploRase package could be modified to handle experiments with large numbers of samples, but it may be more feasible to develop a separate tool for that purpose.

## 7.2 Software infrastructure

ExploRase is written in the R language, facilitating integration with R analysis packages. This also enables other R packages to integrate with exploRase via its public API that is documented through the online R help in the exploRase package.

In order to provide its GUI, exploRase relies on the RGtk2 package [Lawrence, 2007b], a bridge from R to the GTK+ 2.0 cross-platform widget library [Krause, 2007]. RGtk2 allows exploRase to present, completely from within R, a visually pleasing, featureful GUI that is identical across all major computing platforms.

GGobi serves as the visualization component of exploRase. The rggobi package [Temple Lang, 2001b] links R with GGobi. With rggobi, R packages are able to load data from R into GGobi, retrieve GGobi datasets into R, get and set the color of observations, create and configure displays, and more. ExploRase uses rggobi to load high-throughput datasets and synchronize the color of observations in GGobi plots with the colors in the entity information table in the exploRase GUI. This provides the key visual link between the GUI of exploRase and the visualizations of GGobi. The network visualizations are loaded into GGobi using the biocola package, which also depends on rggobi.

The SBML network descriptions are loaded into R using the rsbml package.

## 8 Related work

ExploRase is unique among open-source tools in its integration of interactive graphics with R statistical analysis beneath a GUI designed especially for the biologist. The commercial microarray analysis program GeneSpring links to R and Bioconductor but offers limited interactive graphics. The free program Cytoscape [Shannon et al., 2003] is designed for viewing and analyzing experimental data in the context of biological networks and is integrated with R via plugins. However, it lacks interactive graphics outside of its network diagrams.

Many GUIs have been constructed in R, including several in Bioconductor. The limmaGUI package [Smyth, 2005] provides a GUI that leads the user from preprocessing microarray data to modeling it with limma and producing reports. Unfortunately, limmaGUI lacks the interactive graphics and breadth of analysis features of exploRase. The Bioconductor iSPlot [Whalen, 2005] package provides general interactive graphics using the R graphics engine but offers only a small subset of GGobi's functionality. Rattle [Williams, 2006] is an RGtk2-based GUI that leverages R as it guides the user through a wide range of data mining tasks.

## 9 Conclusion

The exploRase package is an effective tool for analyzing and visualizing high-throughput biological data. Its direct access to R analysis packages, such as limma and others from the Bioconductor project, allow it to take advantage of the latest advances in statistical methods for bioinformatics. The integration with GGobi, including synchronized brushing and the ability to add analysis results as GGobi variables, empowers exploRase to display a wide range of interactive multivariate graphics. All of these advanced statistical features are enveloped within a simplified GUI that is tuned for a biologist.

ExploRase has not yet realized its full potential. There are three predominant directions of planned improvement. The network visualizations in GGobi will be integrated with a convenient and reliable means for matching the identifiers of experimental measurements and nodes in biological networks. This will automatically match identifiers and provide diagnostics to help the biologist ensure the fidelity of the matchings. Analysis methods will be expanded, with a particular focus on clustering and metabolomic analysis. Finally, the visualization of categorical data, such as GO terms and cluster assignments, will be enhanced.

## Acknowledgements

We thank Suh-Yeon Choi, Ling Li and others in the MetNet group at Iowa State University for their helpful feedback in the development of exploRase. We also gratefully acknowledge our funding sources, NSF Arabidopsis 2010 DB10209809 and DB10520267.

## METHODS AND DIAGNOSTIC TOOLS FOR THE ANALYSIS OF GC-MS METABOLOMICS DATA

A paper to be submitted to a Bioinformatics journal

Michael Lawrence, Heike Hofmann, Suh-yeon Choi, Dianne Cook, Eve Wurtele

### Abstract

Gas Chromatography Mass Spectrometry (GC-MS) is a common method for the high-throughput measurement of metabolite levels in a biological sample. A GC-MS metabolomics experiment typically analyzes multiple biological samples, with 3-6 replicates per sample, where the measurements for each replicate are described by three variables: time, m/z, and intensity. We have developed numerical methods and graphical diagnostics for preprocessing the raw data to yield the amounts for each metabolite in each sample. The result is suitable for queries regarding the dependence of metabolite levels on experimental conditions. We present each of our pipeline steps and demonstrate our methods on a dataset from *Arabidopsis*. The analysis routines are implemented in a software tool named *chromatoplot*s, which is based on the R platform for statistical computing and the GGobi software for interactive statistical graphics. The chromatoplot package provides interactive visualizations for examining the result of each pipeline stage and elucidating the effects of parameter settings on the output. There is an Application Programming Interface (API) to chromatoplot for batch processing and a Graphical User Interface (GUI) designed for non-programmers.

## 1 Introduction

Metabolomics experiments often rely on Gas Chromatography - Mass Spectrometry (GC-MS) instruments for measuring the levels of metabolites. We have developed numerical methods and graphical diagnostics for converting raw GC-MS data into a dataset specifying the amount of each metabolite in each sample. In parallel, we have developed a set of graphical methods for diagnosing the algorithm at each pipeline stage. Both the preprocessing algorithms and the graphical diagnostics have been implemented in a package that is accessible to biologists through a wizard-based GUI. In this paper, we present our algorithm for each pipeline stage along with its corresponding diagnostic visualization.

The set of all metabolites in an organism is known as its metabolome. Metabolomics, the study of the metabolome, relies on technology for measuring the amount of a metabolite present in a sample. To assign biological meaning to the results, it is also important to identify the metabolite.

A popular tool for the identification and relative quantification of metabolites is mass spectrometry. A mass spectrometer scans its input across a spectrum of mass to charge ratio ( $m/z$ ) values, and the intensity at each  $m/z$  is measured. The resulting mass spectrum is useful for metabolite identification. Aggregating the mass spectrum for a metabolite is a measure of its quantity.

Biological samples, however, typically contain hundreds of detectable metabolites, so it would not be feasible to analyze the entire sample with a single mass spectral scan. The signals from each metabolite would overlap in the mass spectrum. Thus, a prior separation step is necessary. The sample is passed through a chromatography column and the mass spectrometer repeatedly scans the output of the column over time. Ideally, each metabolite moves at a different rate through the column and exits the column at a unique time. Assuming the separation is successful, the mass spectrometer analyzes a single metabolite in each scan. Gas chromatography (GC) is one of the primary methods for separating metabolites prior to mass spectral analysis. GC-MS is capable of generating consistent mass spectra through the use of electron ionization (EI). The consistency of the mass spectra is important for identifying

the metabolites.

A GC-MS metabolomics experiment typically compares multiple biological samples, with 3-6 replicates per sample, where the measurements from each replicate are described by three variables: time, m/z, and relative intensity. In this paper, we refer to the set of measurements for a single replicate as a *sample*. Raw GC-MS data needs to be processed before it is suitable for answering queries about the relationships between metabolite levels and biological conditions.

We have developed a pipeline for converting raw GC-MS data into a matrix that gives the level of each metabolite in each sample. The name and purpose of each stage, in order of execution, is as follows:

**Baseline subtraction** Remove the chromatographic baseline from each sample, as it may affect peak detection and quantification.

**Peak detection** For each sample, find and characterize each chromatographic peak, which represents the metabolite signal.

**Component detection** For each sample, form a component, a group of peaks that represents a metabolite, by clustering the peaks according to their position in time.

**Component grouping** Across samples, match components that represent the same metabolite.

**Retention time correction** Align samples in time so that components representing the same metabolite occur at the same time in every sample.

**Summarization** Aggregate the peak quantities in each component to derive a single quantity for the component.

**Normalization** Correct for per-sample biases to ensure that the component quantities are comparable across samples.

The numerical methods and graphical diagnostics for the preprocessing are implemented in a package called *chromatoplot*, a name which communicates the emphasis of the package

on statistical graphics. The chromatoplot package provides a visualization for each stage in the analysis pipeline. The main purpose of the visualizations is to help the user evaluate the results generated by each algorithm and diagnose any problems. A secondary goal is to assist the user in understanding the effect of parameter settings on algorithm output. While many GC-MS preprocessing applications provide visualizations of the raw data and the results, the chromatoplot visualizations are designed to help the user validate intermediate results and understand the reasons behind the errors.

The tool is written in the R statistical language [R Development Core Team, 2005] and is based on xcms, an R package that provides a common framework for analyzing LC-MS and GC-MS data [Smith et al., 2006]. The high-level R language facilitates the prototyping of the statistical and graphical methods. In order to provide its interactive visualizations, chromatoplot leverages GGobi, a standalone tool for multivariate interactive graphics. GGobi is embedded into R by the *rggobi* package [Temple Lang, 2001b, Lawrence et al., 2007c]. The R Application Programming Interface (API) to chromatoplot enables R programmers to script data preprocessing tasks. The package also provides a Graphical User Interface (GUI), based on the *RGtk2* package [Lawrence and Temple Lang, 2007], to make its functionality accessible to those who are not expert users of R.

The preprocessing methods and many of the graphical diagnostics methods presented in this paper are in their final form. However, the diagnostics for several of the later stages, specifically the component grouping, retention time correction, summarization and normalization stages, are still in the planning stages. The chromatoplot GUI is also in a nascent form. Nevertheless, this paper illustrates our current vision for these components, which will likely evolve over time.

The layout of this paper follows that of the pipeline. First, we describe the raw dataset that will serve as input, and the subsequent sections address each stage in order of execution. For each stage, we present our analytical methods in the context of existing work and then describe the visualization we have developed for examining the output of the stage. The paper concludes with a discussion of some of the open problems that complicate the preprocessing of GC-MS datasets derived from complex mixtures of unknown metabolites.

## 2 Raw Input Data

### 2.1 Description

The raw data from a GC-MS experiment consists of lists of m/z and intensity pairs for each scan (time point), which forms a matrix where the detections are rows and the columns are time, m/z and intensity. The chromatoplots software can import raw data from any instrument, as long as it is formatted as NetCDF [Rew and Davis, 1990].

The data used to demonstrate the steps of our preprocessing strategy was obtained from GC/MS analysis of Arabidopsis leaf extract. Two genotypes of plant with 4 replicates each were used; wild type and mutant engineered to overexpress 3 exogenous genes. Plants were randomly distributed in a same tray under long day light condition (16hr light, 8 hr dark). Rosette leaf tissues ( 100mg) were collected and fresh frozen in liquid nitrogen. Tissues were extracted in hot methanol, and derivatized for GC/MS analysis. Samples were analyzed using Gas Chromatograph (Agilent Technologies Model 6890) coupled to Mass Selective Detector (Model 5973) in electrical ionization (EI) mode. Detected Mass (m/z) range were 50-750, and total running time was 60 min. Mass spectral measurement began after 5 minutes. Data were exported to NetCDF format using Enhanced ChemStation G1701DA ver. 00.00.38 (Agilent Technologies).

The instrument used in this experiment outputs only the centroids, the local maxima, in intensity over m/z. Our methods assume that the centroids have already been detected. For instruments that are not capable of finding centroids, external software is available to perform that function, such as MZMine [Katajamaa et al., 2006] and xcms [Smith et al., 2006].

### 2.2 Preparation

Since the weight of the electrons is negligible, it seems reasonable to round each mass to the nearest whole unit. This allows the formation of what we call the *profile matrix* [Smith et al., 2006], which has a row for each mass and a column for each scan, in order of time. The ragged output of the mass spectrometer is thus aligned on a grid where each mass spectrum is represented by a column. A separate profile matrix is generated for each sample.

One complication encountered when forming the profile matrix is that the mass spectrometer does not report an intensity for every combination of time and m/z. Most of the time, this is due to the signal falling below the detection threshold of the instrument. The threshold for the mass spectrometer used in this experiment is 150. In this example, the corresponding cells in the profile matrix are imputed by assigning them the value of zero. Our decision to use zero is based on the reasoning that the detection threshold is small relative to the intensity of the detectable metabolite signals. We assume that this is true for any dataset of reasonable quality. Another common approach is to set the missing values to half of the detection threshold. The AMDIS software [Stein, 1999] locally imputes the intensities based on the fraction of time points cross the detection threshold within a time segment. This assumes that a large number of threshold transitions occur when the non-detected intensities are close to the threshold. The chromatoplots user has control over the value used for imputing the missing values.

Figure 1 shows an image plot of the log of the profile matrix for the first control replicate. Higher relative intensity is indicated by a more intense yellow color. The vertical yellow streaks occur at scan ranges where a metabolite with fragments (ions) at many different masses has been detected. The horizontal streaks indicate areas of relatively high background at particular m/z values. In the next section, we describe our strategy that begins with the separation of the signal, the vertical streaks, from the baseline noise, the horizontal streaks.

### 3 Data Analysis Pipeline

#### 3.1 Baseline Subtraction

To characterize each metabolite present in a sample, we need to detect the isolated areas of high intensity relative to the baseline at each m/z bin along the chromatographic time domain. We call these regions *peaks*. When observed across m/z, the peaks form the vertical streaks in Figure 1 that are indicative of metabolites.

Estimation of the baseline is the first step in the separation of the signal from the noise. Figure 2 shows a single row of the profile matrix at 51 m/z. This type of plot is known as an extracted ion chromatogram and shows how the intensity at 51 m/z changes over time. This

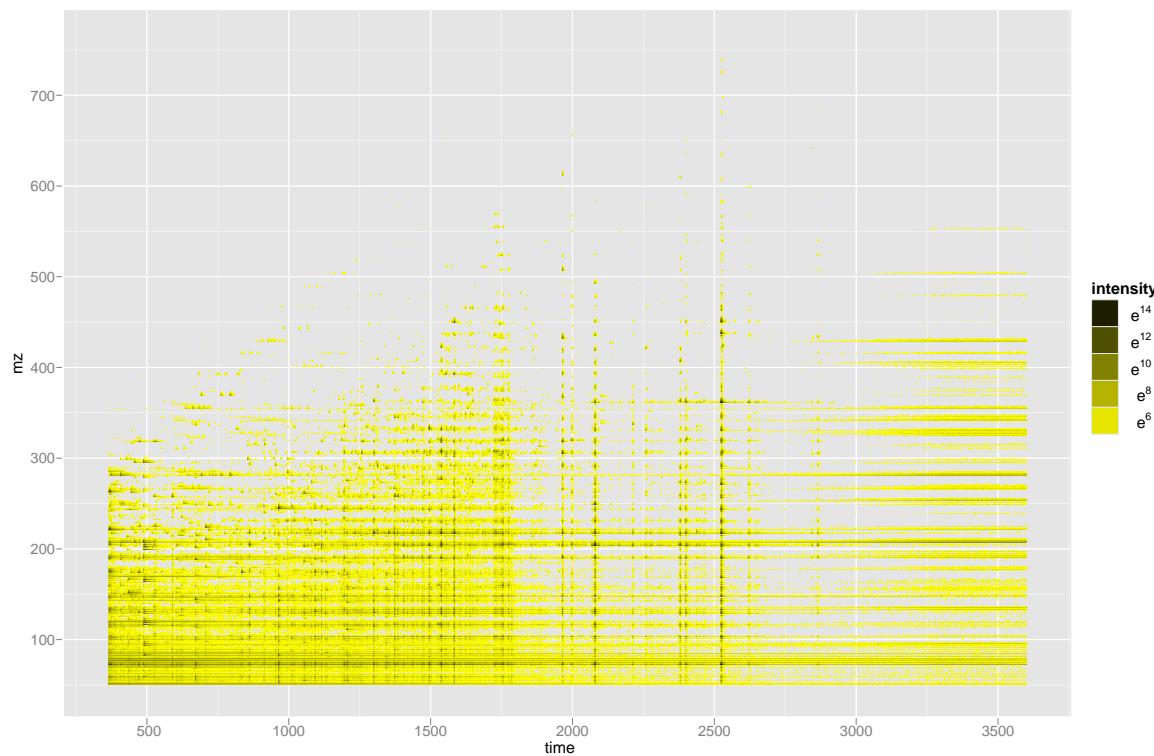


Figure 1 Image of log of the profile matrix. Horizontal axis is time, and vertical axis is m/z. The color scale ranges from yellow to black. The darker the color, the higher the intensity. Gray indicates a time and m/z combination where nothing is detected.

$m/z$  value is chosen, because its relatively low signal to noise ratio makes the baseline more visibly obvious. The “split” in the intensities in the latter part of the chromatogram is caused by setting the non-detected intensities to zero during the formation of the profile matrix. The artifact seems small relative to the peaks, so it is likely of little consequence. An important observation from the chromatogram is that the baseline is much less dynamic than the sharp peaks and the high frequency noise.

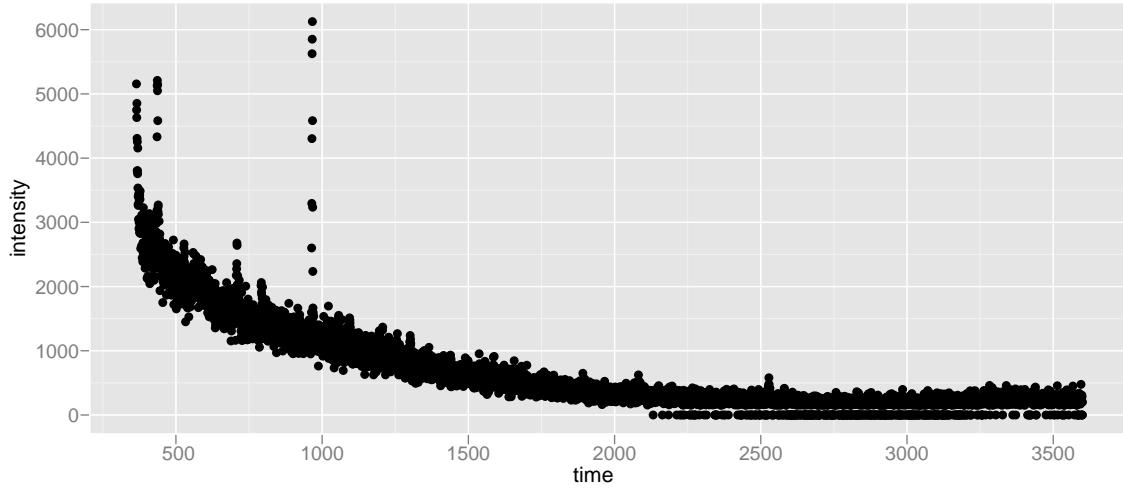


Figure 2 Chromatogram of raw relative intensities from the row of the profile matrix corresponding to 51  $m/z$ . A non-linear baseline is evident.

### 3.1.1 Existing Approaches

Virtually every chromatographic baseline correction algorithm in the literature leverages the slow moving property of the baseline. Some approaches transform the data so that the high frequency signal dominates the low frequency baseline. One strategy is to first smooth the data with a matched filter and then take the negative second derivative [Danielsson et al., 2002, Andreev et al., 2003, Smith et al., 2006]. The transition to curvature removes any linear baseline but yields quantities in terms of curvature rather than relative intensities. The MassSpecWavelet package [Du et al., 2006] transforms the data into the coefficient space for a

range of scales and translations of a wavelet function. This technique assumes that all peaks have the same shape as the wavelet.

Most other strategies rely on regression to fit the baseline. One simple solution is to perform a linear regression using only the data points at the very beginning and end of the time course [Johnson et al., 2003], but this can only correct for a linear baseline. More sophisticated approaches fit a non-linear baseline through local regression. AMDIS [Stein, 1999] fits a linear regression on the values below the median in each region surrounding a local maxima. This method is optimized for accurate peak detection, but not quantification, as the window it considers is too small. The SpecArray [Li et al., 2005b] software models the global baseline with a Savitz-Golay filter, a local smoother. Robust Baseline Estimation (RBE) [Ruckstuhl et al., 2001], implemented in MathDAMP [Baran et al., 2006], repetitively fits a weighted loess model, another form of local regression. The initial fit is uniformly weighted and subsequent fits assign the weights according to the residuals of the previous fit. The weights for negative residuals are set to one and positive residual weights are calculated by the Tukey biweight function. After several iterations, the peaks should no longer affect the fitting. The gray line in Figure 3 shows the RBE fit to the baseline of the chromatogram at 51 m/z. The fit seems correct, but RBE is too computationally expensive for a responsive end-user application.

The running window quantile filter is a simple and efficient alternative for baseline estimation. With a sufficiently small quantile and sufficiently large time window, quantile filters are usually unaffected by sharp peaks. The morphological top-hat filter calculates the minimum in each time window [Sauve and Speed, 2004, Kohlbacher et al., 2007]. The top-hat filter often results in a noisy baseline, because the minima are extreme values and thus sensitive to noise. The median filter is more robust to noise and tends to yield a smooth baseline [Moore Jr and Jorgenson, 1993]. The PROcess package [Li et al., 2005a] fits a loess model to the output of a running window quantile filter.

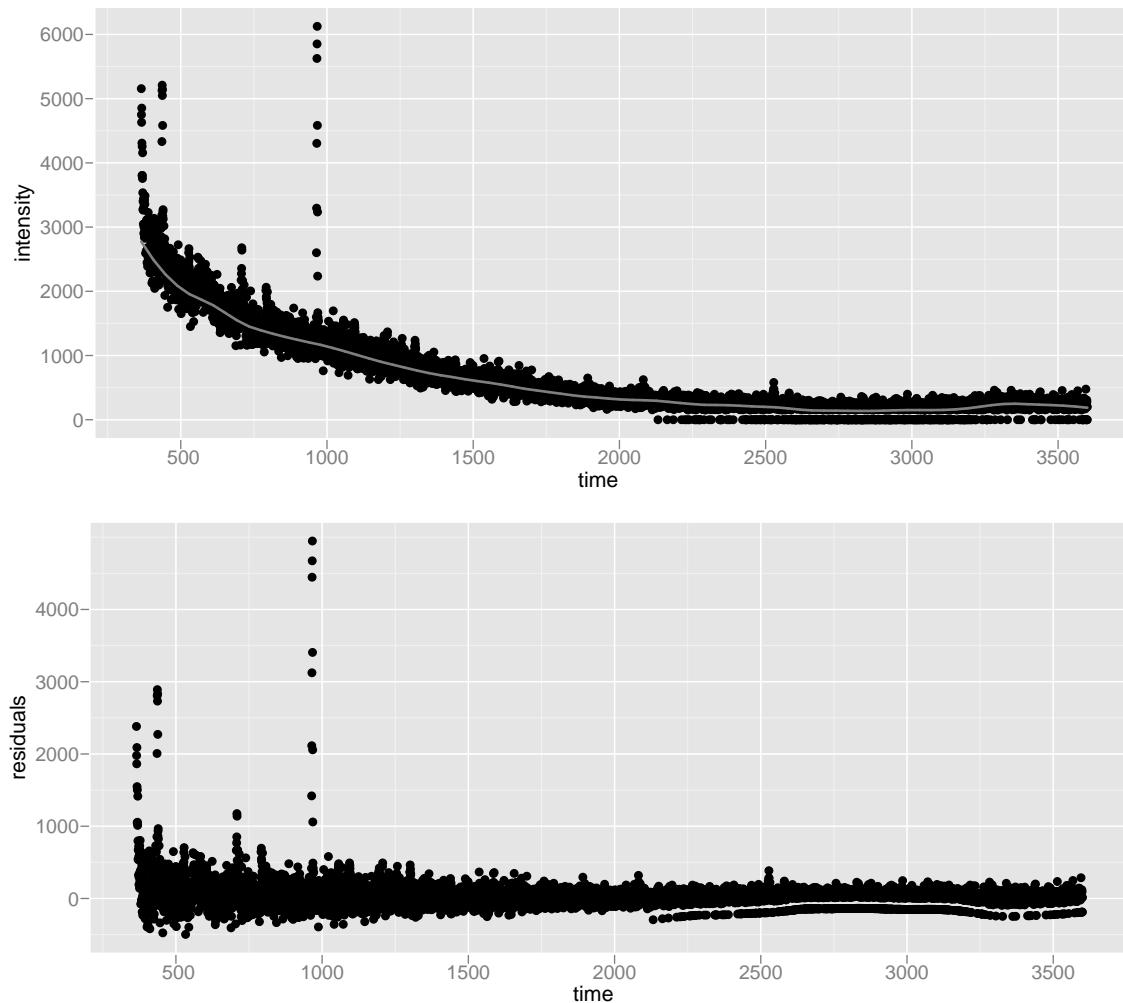


Figure 3 Fit and residuals for the RBE model applied to the chromatogram at 51 m/z. The gray line indicates the baseline fit.

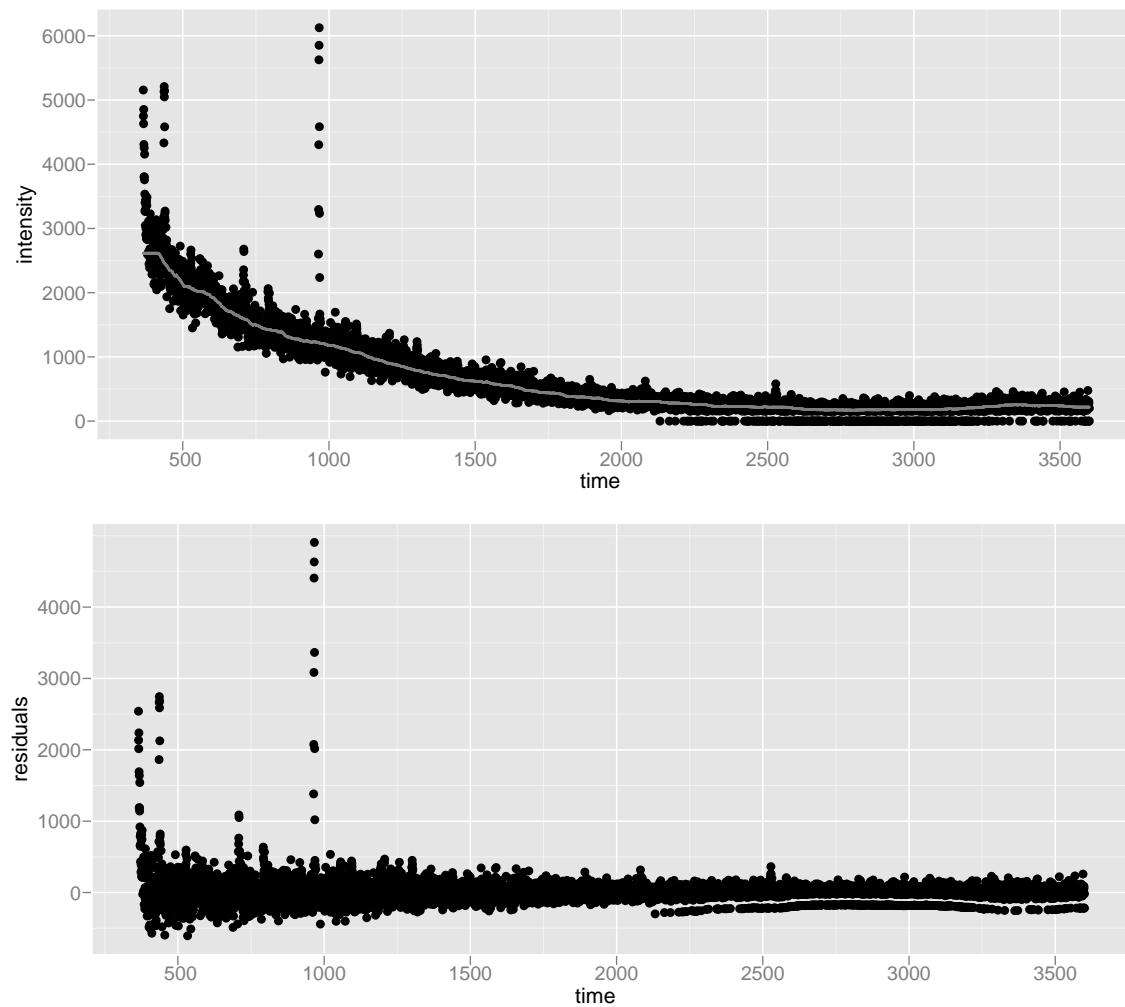


Figure 4 Fit and residuals for the median filter applied to the chromatogram at 51 m/z. The window size is 100 scans. The gray line indicates the baseline fit.

### 3.1.2 Our Approach

We have chosen the median filter as our baseline estimation algorithm. The method slides a fixed-size window along the time course, calculating the median at each step. As long as the window is about twice as broad as the broadest peak, it should not interfere with the peak signal. However, in practice the window may need to be significantly broader to account for the overlap between peaks that often occurs in complex metabolite mixtures. The window is configurable by the user. In this case, we chose the window to be 100 scans. At the extremes, the window would be truncated on one side. This decreases the robustness of the filter, so the medians from the first and last windows are used to interpolate their respective extremes.

### 3.1.3 Results

The median filter, shown in Figure 4 is simple, fast and performs similarly to the RBE approach. No further smoothing seems to be necessary. Figure 5 is the image plot of the profile matrix after subtraction of the coarse median filter. Comparing that image to the raw profile matrix image, Figure 1, it appears that the filter has removed a significant amount of the baseline noise, the horizontal streaks.

### 3.1.4 Graphical Diagnostics

The diagnostic visualization, shown in Figure 6, enables the user to browse the profile matrix and inspect the correction at each m/z. At the top of the visualization is an interactive image plot of the profile matrix that supports the selection of horizontal slices of the image. As the user moves the mouse over the image, a horizontal line follows the pointer and indicates the slice that will be selected in response to a mouse click. When a row of the matrix is selected, the corresponding raw ion chromatogram is drawn in the plot below. The result of the median filter, using the current parameter settings, is drawn over the plot in gray. Below that plot is the chromatogram after subtraction of the median filter.

The aim is to facilitate the inspection of the baseline subtraction at m/z values of interest. The user could, for example, select an m/z in the profile matrix image that appears to have a

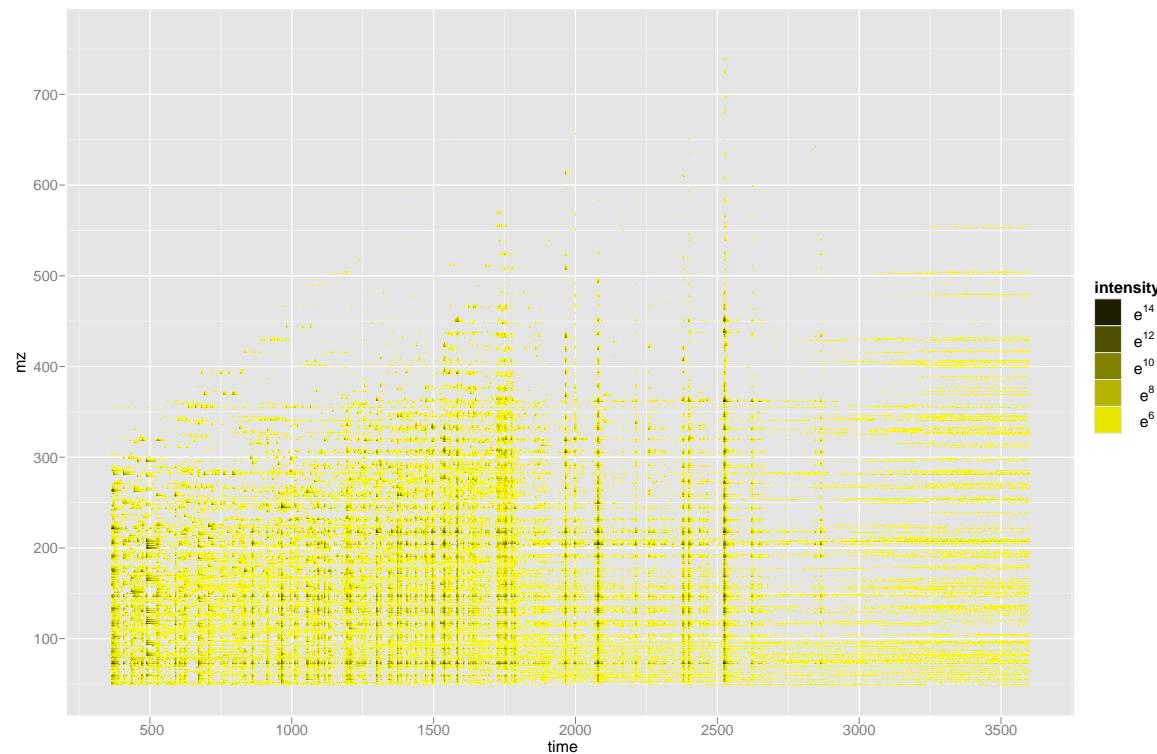


Figure 5 The image of the profile matrix after median filter baseline subtraction. The color scale ranges from yellow to black. The darker the color, the higher the intensity.

significant baseline. From the pair of chromatograms, the user could then observe the shape of the median filter and examine the effect of subtracting the filter from the raw intensities.

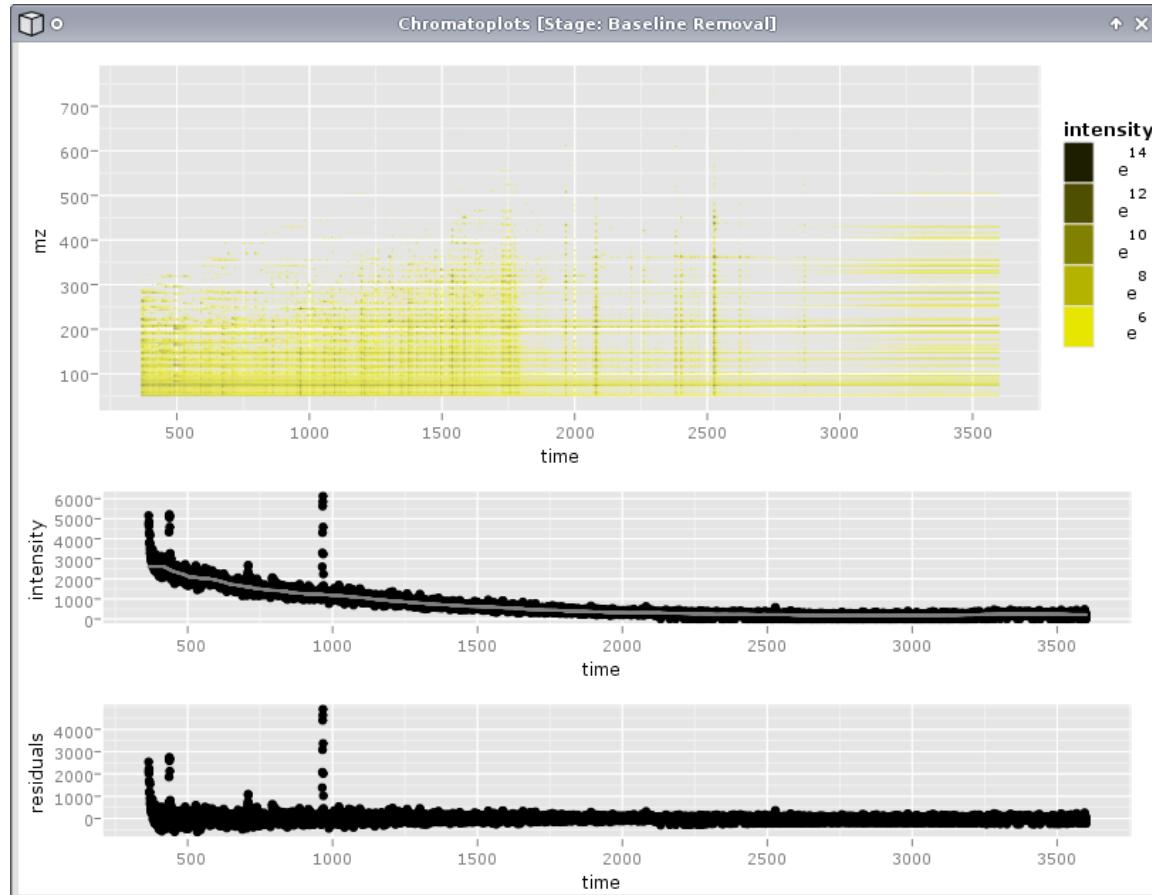


Figure 6 Chromatoplots visualization window for the baseline subtraction stage. The user selects rows in the interactive image plot to view the filter results at the corresponding  $m/z$ . The gray line in the middle plot represents the baseline fit. The bottom plot displays the result of subtracting the fitted baseline from the raw intensities.

### 3.2 Peak Detection

With the baseline removed, it is easier to detect the peaks, the localized regions of high intensity in the chromatogram that indicate the presence of a metabolite. The peak detection step reduces the data from a profile of intensities to a set of features, the peaks.

### 3.2.1 Existing Approaches

The goals of peak detection are to find the peak maxima and to determine the region and shape of each peak. The first challenge is to decide which local maxima in each extracted ion chromatogram, or row of the profile matrix, correspond to true peaks. The second is to determine the region of the peak along the time axis. The final step is to estimate the peak shape parameters, which are used in the peak filtering process and in subsequent pipeline stages.

The first task is to detect the local maxima and separate the peak maxima from those due to noise. Maxima are usually detected either directly or at the zero crossing of the first derivative. Most strategies use a signal to noise ratio as a threshold for filtering the maxima. The approaches mainly differ in their estimation of the noise. The default peak detection method in the xcms package [Smith et al., 2006], for example, calculates the noise from the average of the raw signal, baseline included. This likely overestimates the noise for chromatograms with abnormally high baselines. AMDIS [Stein, 1999] searches every ion chromatogram for flat regions and calculates their median absolute deviation (MAD) from the mean. The median of noise values from all such regions is used as the global noise factor. Other approaches calculate the noise locally, such as finding the moving-window MAD from a smoother fit [Morris et al., 2005] or calculating a quantile for each ion chromatogram [Hastings et al., 2002].

The second challenge is to estimate the peak region on the time axis. AMDIS uses a fixed size window that is truncated if the intensities fall below 5% of the local maximum or another local maximum is encountered. The additional local maxima are ignored if their height above their adjacent local minima is less than a signal to noise ratio cutoff. Other approaches regard the zero crossings in the first or second derivative as the peak limits [Christensen et al., 2005, Smith et al., 2006]. This relies on prior smoothing of the data, which could be problematic.

The final step in peak detection is to describe the peak shapes corresponding to the detected maxima. This is generally done by fitting a model to each peak. The MassSpecWavelet and xcms packages assume that every peak has an identical shape. While most peaks are similarly shaped, the shapes are not identical and some are asymmetric. There are arguments that the

assumption of a fixed shape is still a reasonable approximation [Andreev et al., 2003]. However, in complex mixtures of metabolites, many similar molecules nearly intersect in time and spectrum. When multiple metabolites overlap to form one peak, the shape is significantly different (i.e., flatter) than the shape of a singlet peak (i.e., one derived from a single metabolite). It would be beneficial to detect such convolution through abnormalities in the peak parameter estimates, rather than incorrectly fit the peak or ignore it entirely.

If it is invalid to assume that the peak shapes are identical and well-resolved, it is necessary to fit a model to each peak individually. AMDIS, for example, fits each peak with a parabola.

Two commonly used models for fitting chromatographic peaks are the Gaussian distribution (Equation 1) and some form of Exponentially-Modified Gaussian (EMG) distribution [Di Marco and Bombi, 2001]. One member of EMG class is the exponential-gaussian hybrid (EGH), given by Equation 2 [Lan and Jorgenson, 2001].

$$g(t) = H_r \exp\left(\frac{-(t-t_R)^2}{2\sigma_g^2 + \tau(t-t_R)}\right) \quad (1)$$

$$h(t) = \begin{cases} H_r \exp\left(\frac{-(t-t_R)^2}{2\sigma_g^2 + \tau(t-t_R)}\right), & 2\sigma_g^2 + \tau(t-t_R) > 0, \\ 0, & 2\sigma_g^2 + \tau(t-t_R) \leq 0, \end{cases} \quad (2)$$

The EGH is simple compared to other EMG variants and adds only a single parameter,  $\tau$ , to those inherited from the Gaussian.  $\tau$  weights the exponential component of the function, such that  $\tau = 0$  yields the original Gaussian distribution. It has been reported that the EGH has the flexibility to model a wide range of asymmetric peak shapes, while still remaining relatively stable [Lan and Jorgenson, 2001]. The EGH has been applied previously to the estimation of peak shapes in GC-MS metabolomics data [Christensen et al., 2005].

### 3.2.2 Our Approach

Our peak detection procedure begins by calculating the noise from a global quantile, calculated across all m/z, on the baseline-corrected data. Looking at Figure 5 one may notice that some m/z (rows) have more points of high intensity than others. We selected a global quantile, because it avoids penalizing maxima detected in ion chromatograms with a large number of

peaks relative to those with few peaks. It also has the advantages of simplicity and speed. Figure 7 shows the global cutoff, using the 99th quantile, applied to chromatograms of two different ions, at 51 and 73 m/z respectively. A limitation of the global approach is evident in the plot for 51 m/z, where the noise appears to change across the time course, while the cutoff value does not. However, the change in noise seems small relative to the cutoff value.

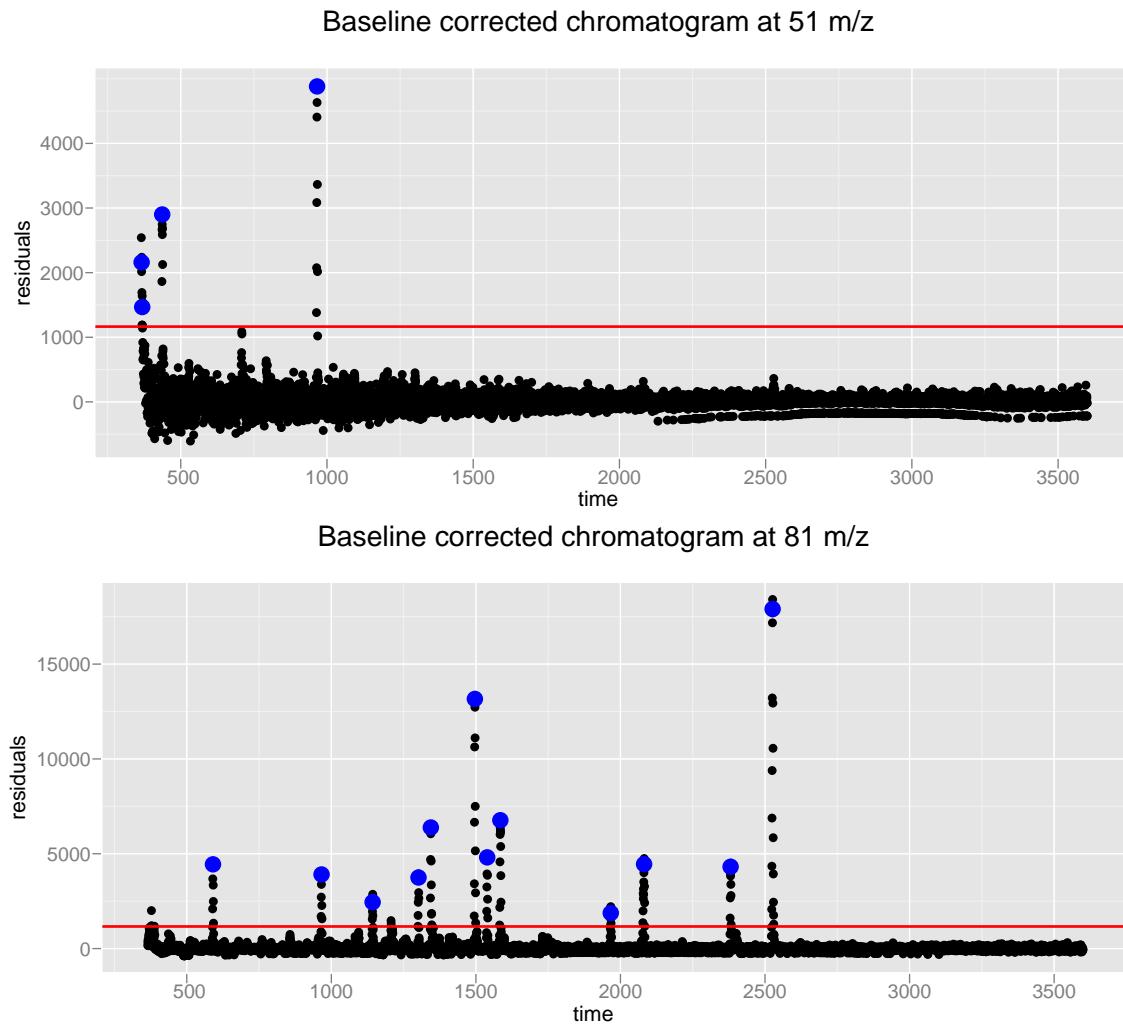


Figure 7 Local maxima detection at 51 m/z (top) and 72 m/z (bottom). The blue dots denote the detected maxima, and the red line represents the quantile cutoff.

Our method finds local maxima in the “islands” of data points above the global quantile.

By default, the island must consist of at least three data points. This is to ensure that there is enough signal to characterize the peak and represents a value that is less than the width of most peaks associated with metabolites. The peak region is extended on the left and right of the island an amount proportional to the size of the island with the restriction that the region cannot overlap with a second island. If multiple maxima occur within the same island, we filter each maxima based on whether its height above its flanking local minima is some multiple of a noise estimate based on the variance in the data below the quantile cutoff. If multiple maxima pass the filter, the island is cut at the local minimum between each remaining maxima. This strategy is similar to that of AMDIS, except for the noise estimation and in the way the peak region is expanded in absence of other maxima.

To obtain a functional description of the peak shapes, either the Gaussian or the EGH are fit to each peak region using non-linear least squares optimization. This is a computationally intensive operation, so a reasonable parameter initialization is important. Initial values for  $t_R$  and  $H_R$  are taken from the time and value of the local maximum. For the initial value of  $\sigma$ , the sum of the intensities in the peak region is divided by  $H_R\sqrt{2\pi}$ , according to the definition of the Gaussian distribution. For the EGH,  $\tau$  is initially set to 0, under the assumption that peak asymmetry is typically small. For each peak, the algorithm first attempts to fit the EGH. As it is less stable than the Gaussian, the EGH sometimes fails to converge, in which case the algorithm falls back to the Gaussian. If the Gaussian fails to converge, the peak is discarded, as its shape likely does not resemble that of a peak.

After the parameters are estimated for a peak, it is possible to detect and discard peaks with abnormal parameter estimates. Specifically, a peak is ignored if its  $t_R$  is outside the fit region or its  $\sigma$  is larger than the fit region. The discarded peak fits appear “flat” over the peak region, and the data in the region normally lack a discernible peak shape. Typically, these errors result from either noise that has risen above the quantile cutoff, incorrect estimation of the peak region, or severe convolution.

### 3.2.3 Results

The blue dots in Figure 7 denote the maxima of the detected peaks. There are a few cases where spikes in signal above the cutoff are not recorded as peaks. These are usually due to an insufficient number of sequential points above the cutoff. The lower limit in this analysis is three. In rare cases, intensities above the cutoff value are intermixed with missing values, which are imputed as zero. These cases do not adhere to a peak shape and are thus discarded during the peak fitting process. The cause of these aberrations may be errors in the centroid detection in the m/z dimension. The mass spectrometer is responsible for that step, so it is outside of our control.

Figure 8 displays two fits of the same peak, first by the Gaussian and then by the EGH. The residuals are shown below the corresponding fit. This particular peak appears to have an asymmetric shape and thus is better fit by the EGH function. However, a significant amount of asymmetry is left in the residuals. This may be due to convolution with other peaks.

The peak detection algorithm yields between 4500-5000 peaks per sample for this dataset. In effect, the set of profile matrices has been reduced to a set of peak matrices, with each row corresponding to a peak. Columns contain peak information, such as the value and time of the fitted maximum,  $H_R$  and  $t_R$ , and the peak shape parameters,  $\sigma$  and  $\tau$ . Figure 9 is an image plot of the peak locations in time and mass for the first control replicate. The intensity of the yellow color is an indicator of the peak height. The plot is essentially the filtered version of the profile matrix image in Figure 5. The data is now at the peak level, but it still does not directly describe metabolites. For that, it is necessary to perform the next step in the pipeline.

### 3.2.4 Graphical Diagnostics

Figure 10 shows the visualization for evaluating the peak detection results. Like baseline correction, peak detection considers each row of the profile matrix separately, so the visualization supports the browsing of the profile matrix by row. The baseline corrected chromatogram for the selected m/z is shown below the profile matrix image. The chromatogram includes a red horizontal line indicating the quantile cutoff, and a blue glyph is drawn over each detected

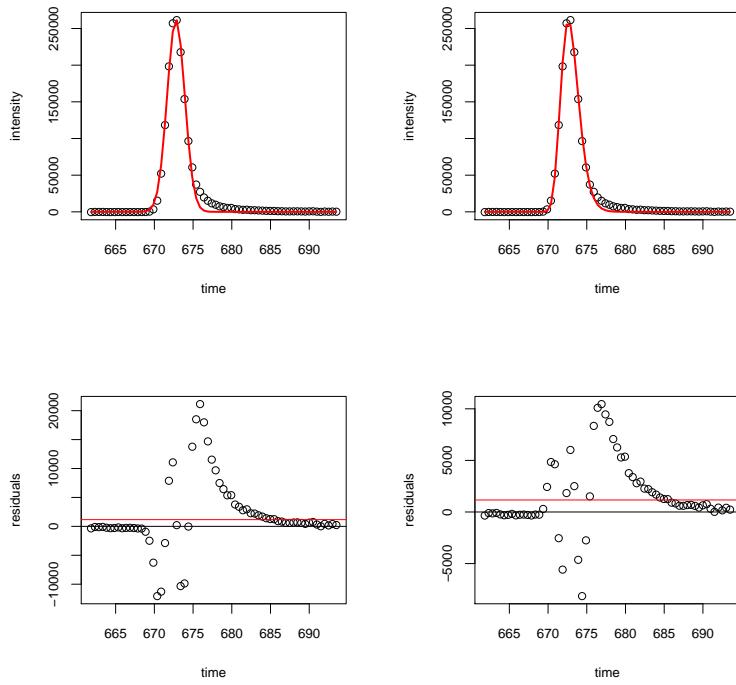


Figure 8 Comparison of Gaussian (left) and EGH (right) fits to an asymmetric peak. The top plots show the fitted curves, and the residuals are plotted at the bottom. The EGH appears to fit about half of the asymmetry.



Figure 9 ]

Scatterplot of the peak locations in time and m/z. The color scale is analogous to that of the image plots of the profile matrix intensities. The darker the color, the higher the peak.

maxima. From this, the user can check whether the cutoff is sensible and whether algorithm is detecting the appropriate maxima as peaks.

After detecting the maxima, the peak detection algorithm fits a function to each peak. To examine these peak fits, the visualization provides a GGobi plot of the detected peaks by time and m/z, which corresponds to the profile matrix image. The user can pan and zoom the plot and move the pointer over a point to display the corresponding peak fit on the right. The user may select other variables to show in the GGobi plot, such as  $t_R$ ,  $H_R$ ,  $\sigma$ , or  $\tau$ . For example, the user could plot the peaks by  $H_R$  and  $\sigma$  to find peaks with abnormally flat fits. By moving the mouse over the outliers, the user could check whether the algorithm is trying to fit a convolution of multiple peaks.

### 3.3 Component Detection

A *component*, in our terminology, is defined as the set of peaks in a particular sample that are generated by the same chemical. Theoretically, the peaks of a component all occur at the exactly the same time. However, due to noise and convolution between components, the peak detection step will rarely output identical times for two peaks in the same component. The times should, however, be nearly identical.

#### 3.3.1 Existing Approaches

Existing methods generally bin the peaks in time in order to form components. AMDIS considers the peak shape in its binning algorithm. Peak “sharpness” values are calculated from the peak fits and summed over every m/z value. AMDIS then searches for local maxima in the sum of the peak sharpness values. For each local maxima, a region in time centered on the maxima is determined with its length inversely proportional to the maximal value. As long as no other maxima occur within that region, the all peaks maximizing within that region are assigned to a single component. The time window calculation is based on the assumption that a sharp component has peaks that maximize within a smaller time window than broad components.

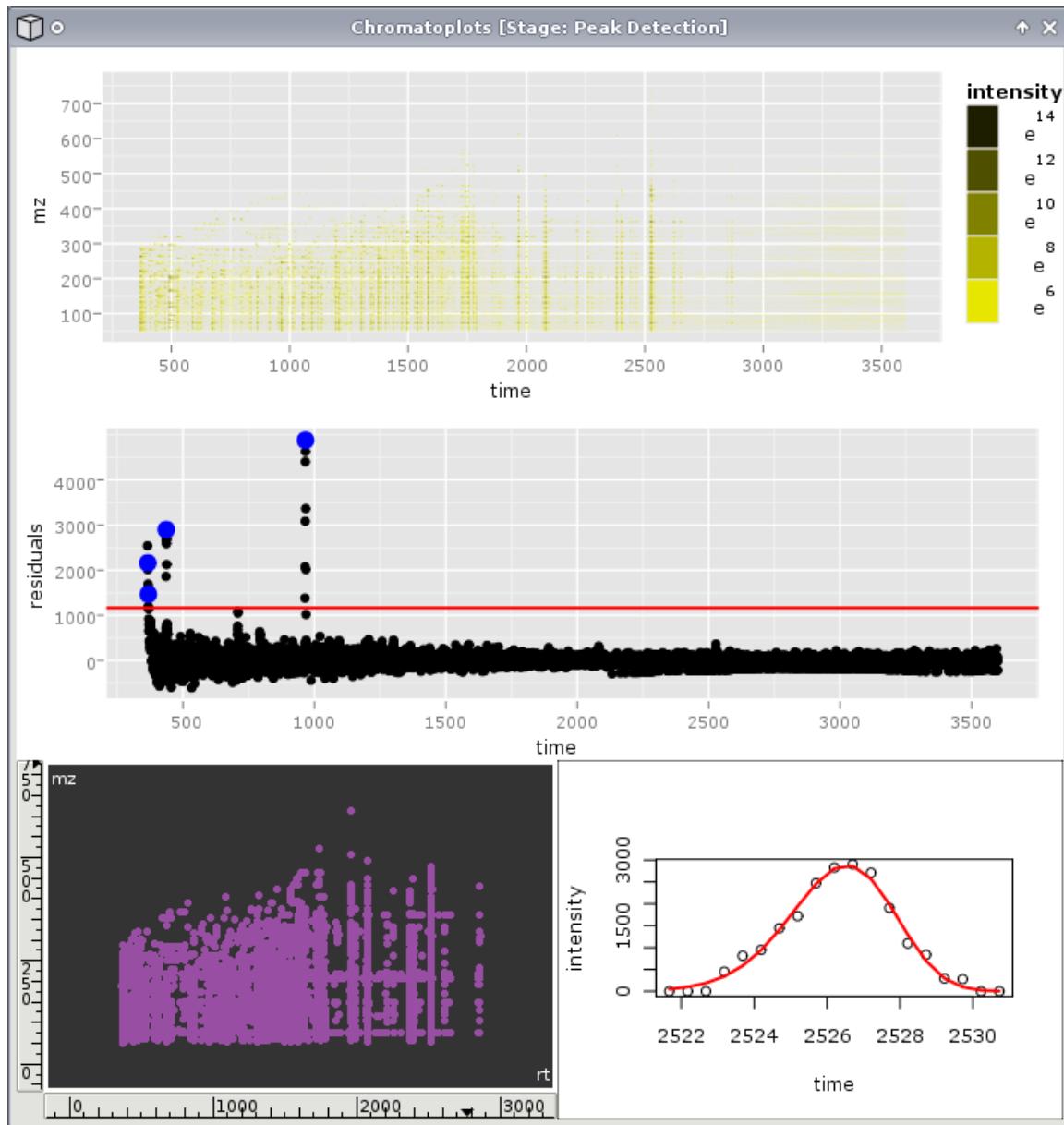


Figure 10 Chromatoplots visualization window for the peak detection stage. The user selects rows in the top image to view the baseline-corrected chromatogram at the corresponding  $m/z$ . The chromatogram is annotated with the quantile cutoff and the maxima of detected peaks. On the bottom is a GGobi visualization for exploring the peak fits.

### 3.3.2 Our Approach

We have also adopted a binning strategy for forming components. We take a different approach from AMDIS, though one similarity is that our algorithm also bases the component bin widths on the sharpness of the peak fits. In our case, the sharpness is represented by the *sigma* parameter. As an overview, the algorithm first creates the set of components, assigning each a time  $t_R$  and spread  $\sigma$ , and then assigns each peak to one of the components. This is done separately for each sample.

Each component is created from an initial seed peak. To initialize the search for seed peaks, the peaks are sorted in decreasing order by their height, under the assumption that the tallest peaks are the best resolved and have the most accurate retention time for their component. The algorithm proceeds through the list of potential seed peaks, from tallest to shortest, creating a component for each one. Each component is described by the  $t_R$  and  $\sigma$  from its seed peak. The time of the component is indicated by  $t_R$ , and  $\sigma$  is a measure of the component width in time. Any peaks that maximize within the time range of  $(t_R - \sigma)$  to  $t_R + \sigma$  of a newly created component are removed from the list of potential seeds and thus excluded from consideration in future iterations. An exception occurs when multiple peaks with the same m/z value fall within that range. Only a single peak from a particular m/z can be logically assigned to a component. Thus, the peak nearest to the group in time is excluded, while the others are left in the list of potential seeds.

Once the components are created, each peak is assigned to a component. The algorithm makes another pass over the peaks, assigning each to its closest component in time. As in the seeding step, no two peaks in the same component can have the same m/z value, and the time distance between a component and each of its peaks must be less than the component  $\sigma$ . Any peaks not assigned to a component are converted to single-peak components in the same way as the seed peaks.

The rule that multiple peaks at the same m/z cannot be placed in the same component allows the algorithm to detect overlapping components and split the peaks between them. A limitation is that many of the peaks are actually convolutions of several peaks, each from a

different component. In such cases, each component should be given a peak, but the algorithm assigns the convolution to only one of the components. This produces under-specified components that may be problematic in subsequent stages of the pipeline.

### 3.3.3 Results

A component is essentially a mass spectrum, or approximately a column in the profile matrix, that represents a metabolite for a particular sample. Graphically, components correspond to the vertical streaks in Figures 1, 5, and 9. Figure 11 illustrates this by drawing vertical lines that chain together the peaks that belong to the same component.

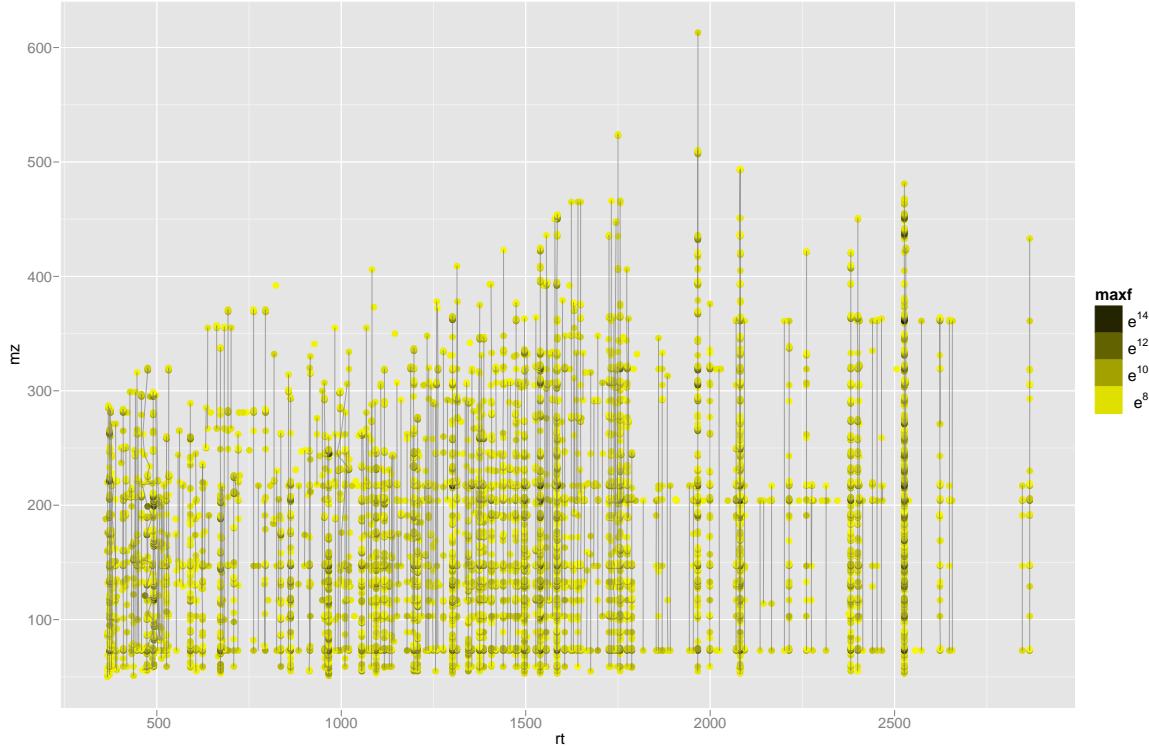


Figure 11 Same as Figure 9, except the components are indicated by line segments chaining together the peaks belonging to the same component.

A dataset of components is created with variables for  $\sigma$ ,  $t_R$ , the number of peaks, and the standard deviation of the peaks in time,  $\sigma_t$ . A plot of  $\sigma$  vs  $\sigma_t$  is shown in Figure 12. The two

variables appear to be positively correlated. Most of the components have  $\sigma < 4$  and  $\sigma_t < 2$ . The outliers in the plot are highly questionable and likely result from choosing a malformed peak as the seed for a component.

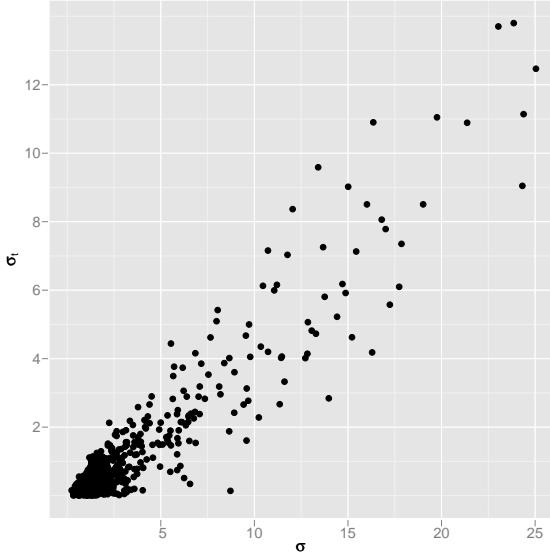


Figure 12 Scatterplot of the components by *sigma*, the width in time of the component, and  $\sigma_t$ , the standard deviation in peak maximum times. Most of the observations have a  $\sigma < 4$  and  $\sigma_t < 2$ , but there are a significant number of outliers. The two variables appear to be correlated.

### 3.3.4 Graphical Diagnostics

For diagnosing the components, the user needs to be able to visually identify erroneous components and query them to discover the cause of the error. A GGobi scatterplot for variables in the components dataset is at the top of the diagnostic visualization in Figure 13. The user may plot any of the variables, such as  $\sigma$  and  $\sigma_t$ . To the right is a plot that displays the mass spectrum for the currently selected component. Below that is the same plot at the bottom of the visualization for the the peak detection stage, except lines chain together the peaks in the same component.

The user can move the mouse pointer over the outliers in the top scatterplot to display

the component mass spectrum and highlight the peaks and lines corresponding to the group in the bottom plot. As in the peak visualization, mousing over the peaks in the second plot displays the peak fits and residuals. This allows the user to drill down to the peak fits to determine the source of any problems. Also, by zooming in on the peak plot, the user can investigate whether two components that are close in time have overlapping peaks. For severely overlapping components, one could inspect the peak fits to understand why the algorithm distributed the convoluted peaks between the two components in a certain way.

### 3.4 Grouping Components Between Samples

The overall goal of the pipeline is a matrix where each row contains the levels of a metabolite across every sample in the experiment. This requires that the components, which represent metabolites, be matched across samples. A *group* is our term for a matched set of components.

#### 3.4.1 Existing Approaches

The problem of matching features across chromatographic samples has been well-studied. The most common approach is to cluster the features, in our case the components, in time. Due to retention time drift between samples, it is not possible to assume that the times match exactly. Examples of time-based clustering methods include fixed-width binning [Johnson et al., 2003], the k-means variant of the MZMine software [Katajamaa et al., 2006], the hierarchical approach of msInspect [Bellew et al., 2006] and model-based strategies [Smith et al., 2006, Wang et al., 2007, Jaffe et al., 2006, Yu et al., 2006]. These methods are all specifically designed to match peaks between samples at each m/z region.

Our components are sets of peaks, so there is more information to leverage in distance calculations. Many distance measures have been proposed for comparing mass spectra [Wolski et al., 2005]. The measures have been evaluated for their effectiveness in the context of mass spectral library search [Wolski et al., 2005, Liu et al., 2007, Stein and Scott, 1994]. A consistently well-performing measure is the uncentered correlation, or the cosine-angle distance,

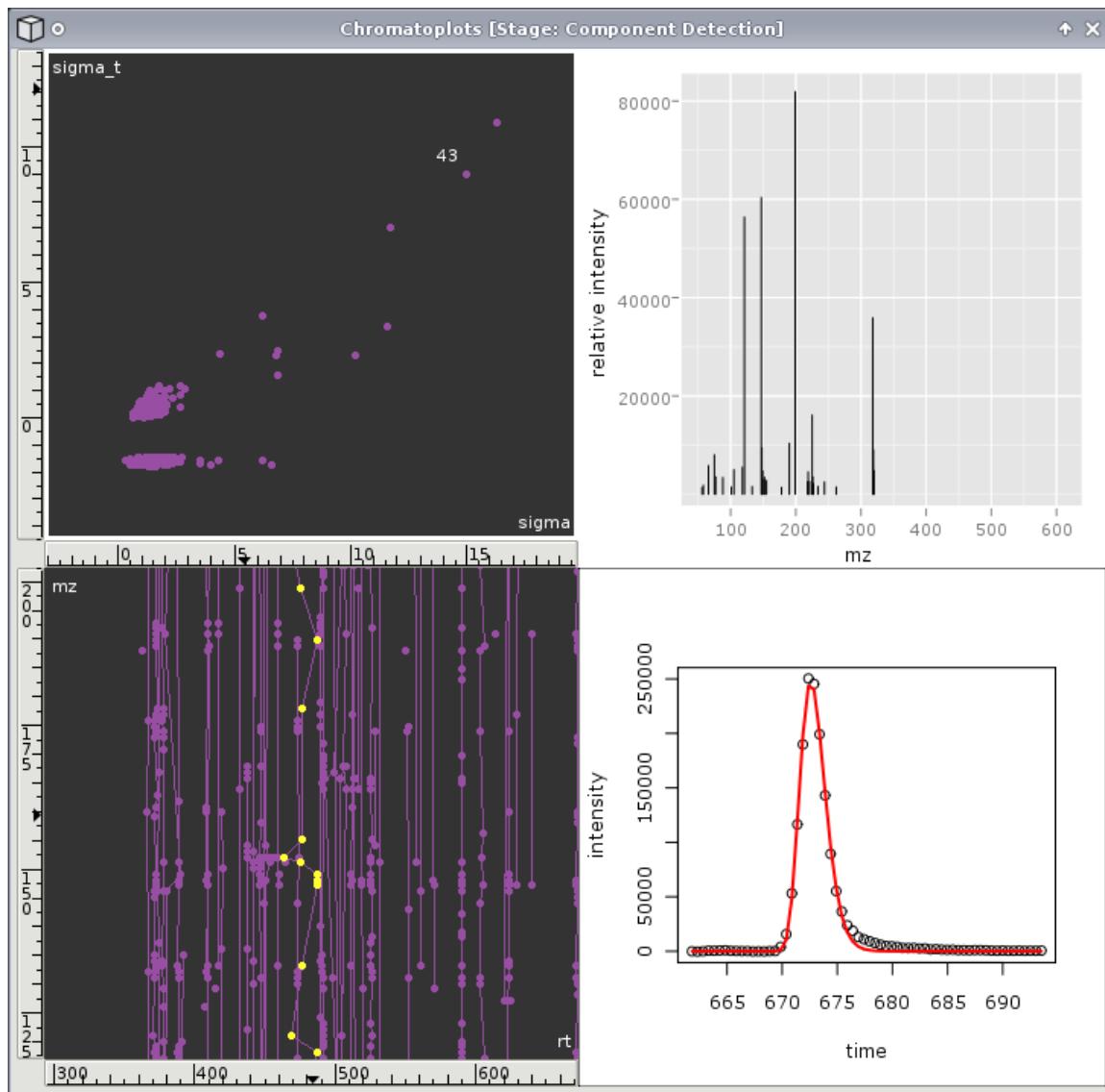


Figure 13 Chromatoplots visualization window for the component detection stage. When the user selects components in the top scatterplot, the mass spectrum for the component is displayed to the right and the corresponding chain of peaks is highlighted in the scatterplot on the bottom left. Selecting a peak in that plot displays the peak fit to the right.

defined in Equation 3.

$$\rho_{i,j} = \frac{S_i \times S_j}{\sqrt{(S_i \times S_i)(S_j \times S_j)}} \quad (3)$$

The uncentered correlation has been cited for its simplicity, well-understood statistical properties, and superior performance to other methods.

Proposed methods for matching components based on the spectral distance include the Needleman-Wunsch alignment algorithm [Prakash et al., 2006] and greedy matching with a distance and time cutoff [Krebs et al., 2006, Idborg et al., 2004]. In principle, any distance-based clustering method could be applied.

### 3.4.2 Our Approach

We have chosen the uncentered correlation for the calculation of distances between components. We convert the correlation to a distance measure by taking its absolute value and subtracting from 1, ie  $1 - |cor|$ .

For matching the components, our algorithm follows the greedy matching approach mostly due to its simplicity. The distances are considered in increasing order. For a given distance, the corresponding components are matched, as long as the distance and time cutoffs are satisfied and neither component has been previously matched to a component in the same sample as its putative partner. The last constraint ensures that each sample contributes at most one component to a group. The default spectral distance limit is the 5% quantile of all the pairwise distances. This is usually a conservative cutoff, though it does depend on the number of components. The time cutoff is set to zero for the initial matching, as the samples are not yet aligned in time.

The above algorithm is applied recursively in a post-order traversal of the tree formed by the experimental design factors. Figure 14 illustrates how the tree is constructed. The demonstration dataset has a single factor, genotype, with two levels, wildtype and mutant. The samples are split into groups according to their genotype. Those groups consist of replicate samples, the leaves of the tree. The matching algorithm recurses down the tree and begins matching within the genotype groups. Under the assumption that a replicate set has a larger

number of common features than sample sets at higher levels of the tree, we expect that matching components from replicates will result in larger groups with fewer mismatches. We expect that every level of the tree will have a more compatible sample set compared to its parent. A secondary motivation for the recursion is to avoid the computational complexity of calculating a pairwise distance for all samples. After group formation, the components with the median time in their group are collected into a set of groups meant to be representative of the current sample set. By selecting the group with the median time, we hope to avoid choosing mismatched components. That set of groups is then matched at the parent level of the recursion, and the matchings propagate to the components represented by the median set.

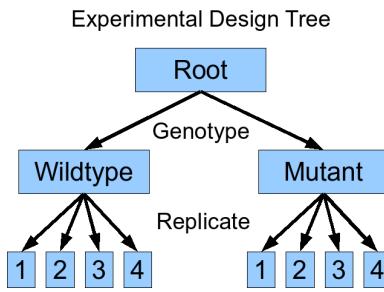


Figure 14 An illustration of how the experimental design is represented as a tree in the recursive algorithms for component matching and retention time correction. The *genotype* factor splits the samples into two groups, one for each of the levels, wildtype and mutant. Those groups consist of four replicates each.

### 3.4.3 Results

A group dataset is generated with variables for the number of components, the total number of peaks, the average number of peaks per component, the standard deviation of the peak times ( $\sigma_t$ ), and the average pairwise spectral distance between the components ( $\mu_d$ ). A scatterplot of the group dataset on the variables  $\sigma_t$  and  $\mu_d$  is shown in Figure 15. Most of the points are clustered in the lower left corner of the plot, indicating that most groups have similar spectra and times. This reflects positively on the performance of the matching algorithm.

The outlying points in the plot are most likely mismatches. One might expect a large number of mismatches given the conservativeness of the distance cutoff. Somewhat concerning is that there is a number of groups where the components are close in time but far in spectral distance and vice versa. Components that are highly similar in time but dissimilar in spectra are likely distorted by convolution. It is not clear whether these cases should be considered mismatches. Those with small spectral distances but similar times may only have detectable peaks at the most prevalent m/z values and are thus difficult to distinguish on the basis of spectra alone.

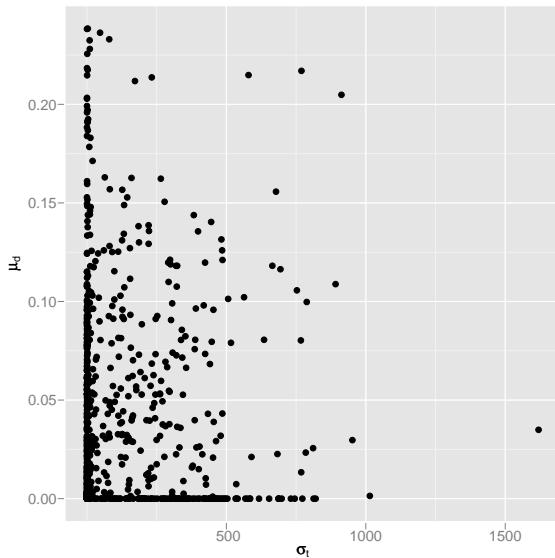


Figure 15 Scatterplot of groups with  $\sigma_t$  on horizontal axis and  $\mu_d$  on vertical axis. Groups with large values for either variable are likely errors.

To resolve some of the mismatches, we could enforce a fixed-size time window for valid matches [Krebs et al., 2006, Idborg et al., 2004]. As noted above, however, this is complicated by retention time drift between samples. Thus, the samples need to be aligned in time before time information can be incorporated into the matching algorithm.

### 3.4.4 Graphical Diagnostics

The overall goal of the visualization shown in Figure 16 is to help the user locate and diagnose groups with components that do not correspond in time or spectral distance. The top scatterplot is of the groups by their standard deviation in time,  $\sigma_t$ , and their average pairwise spectral distance,  $\mu_d$ . To the right is a plot of the components with sample on the vertical axis and time on the horizontal. Lines chain together components in the same group. This is analogous to the plot for the component stage that chains together peaks. The points are now components and the vertical axis is sample instead of m/z. This follows from the distinction that peaks are matched across m/z while components are matched across sample. At the bottom is a parallel coordinate plot of the components of a single group. It is similar in form to a mass spectrum, with intensity on the vertical axis and m/z on the horizontal. The points correspond to peaks. Only the m/z values that have at least one peak are placed on the horizontal axis.

By moving the mouse over an outlier in the plot of groups at the top, the corresponding components are highlighted in the plot to the right. In addition, the plot at the bottom is updated to show the components from the selected group. Mousing over a component in the middle plot highlights the corresponding peaks and the lines connecting them in the parallel coordinate plot at the bottom. The user may then examine the patterns from which the spectral distance is calculated. Through these interactions, the user can identify groups with components that are outliers in time, spectrum or both and determine the reasons behind the errors. For example, in Figure 16, an outlying group has been selected in the top-left scatterplot and, from the plot in the top-right, it appears that one of its components has an unusual time. This component has been selected to highlight its spectrum in the bottom plot. The spectrum is not an obvious outlier from the other component spectra. Thus, the result, though likely an error, is unsurprising. The user might conclude that this error would be resolved through the use of a time cutoff.

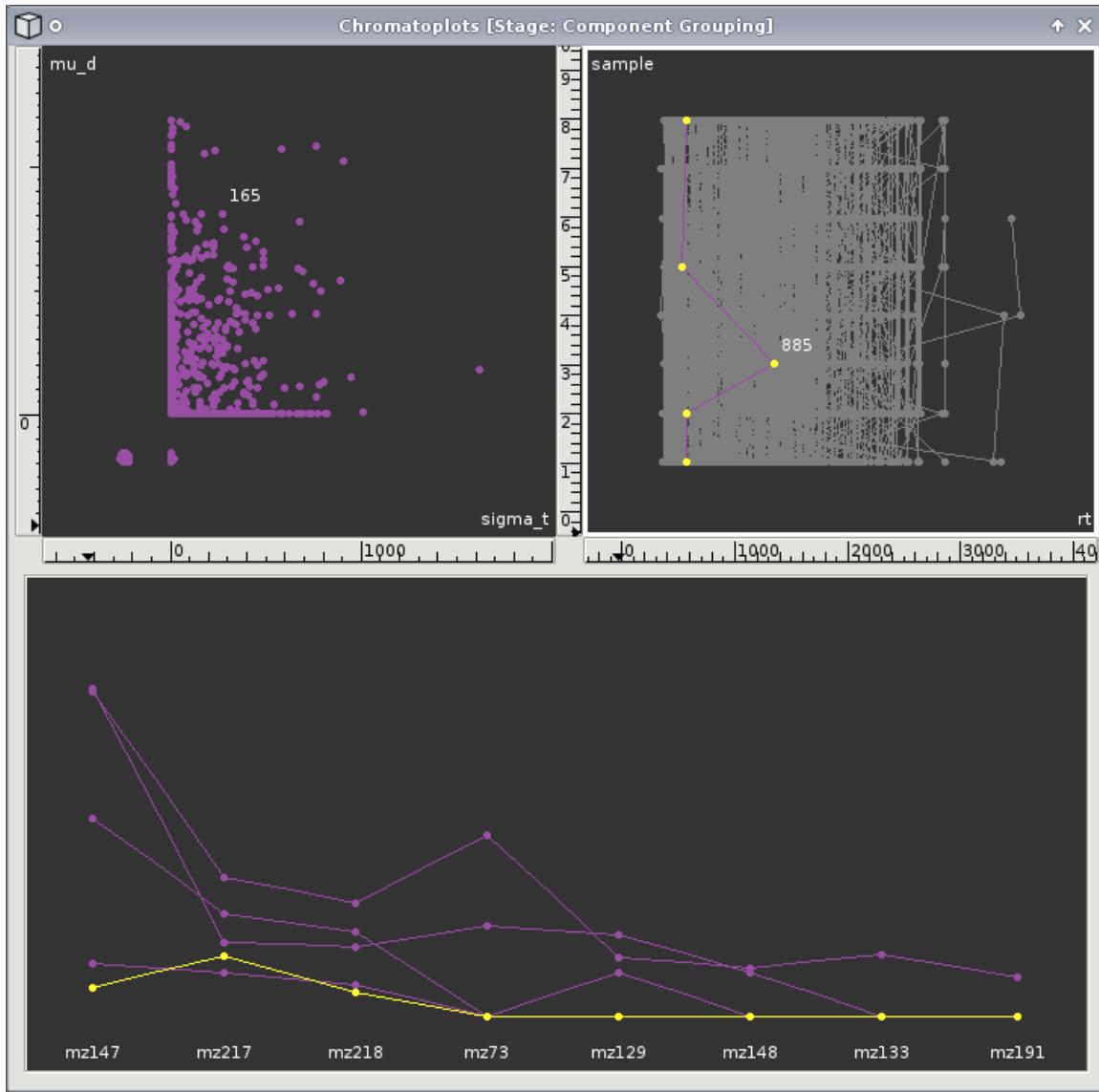


Figure 16 Chromatoplot visualization window for the component grouping stage. The user selects a group in the top scatterplot to highlight the corresponding chain of components in the plot to its right and display the spectra of the components in the parallel coordinates plot at the bottom. A group with relatively high values for  $\mu_d$  and  $\sigma_t$  has been selected in the top-left scatterplot. Its components are highlighted in the plot on the right. The outlier in time has been selected and its compressed spectrum is highlighted below. Despite the difference in time, the spectrum does not appear to be very different from the other component spectra. This explains why the component is assigned to the group. A time cutoff would avoid this error.

### 3.5 Retention Time Correction

The retention time drift between samples is problematic, because it precludes the use of time information when matching components across samples. If the time axis of each sample were warped to match a reference, one could assume that components in different samples corresponding to the same metabolite would occur at the same time, within some error tolerance.

#### 3.5.1 Existing Approaches

Methods for correcting retention time may be classified into two sets: those that directly manipulate the profile data and those that optimize a function that maps values between time axes. A method of the former class, Correlation Optimized Warping (COW), uses dynamic programming to stretch and compress segments of a time course to optimize its correlation with a reference [Nielsen et al., 1998, Baran et al., 2006]. COW has also been applied to the regions around detected peaks, which avoids correcting the entire profile [Christensen et al., 2005]. Hidden Markov models have been used to simultaneously align multiple profiles through the estimation of a “latent trace” [Listgarten et al., 2005]. This method relies on the assumption that every sample contains approximately the same set of compounds. In general, the methods that are limited to the warping of profile data have several drawbacks. They tend to be computationally intensive, and, as they do not consider high-level features, they lack the flexibility to handle changes in component elution order. Moreover, these methods do not directly address our need to adjust the retention times of the components in order to achieve a better grouping across samples.

Other retention time correction methods optimize a function that generally maps a time value, such as a time parameter of a feature, from one time axis to a reference axis. Several types of time warping functions have been proposed. Parametric Time Warping (PTW) fits a quadratic function that warps a given time series to a reference [Eilers, 2004, Kohlbacher et al., 2007, Jaffe et al., 2006]. PTW is extremely efficient, but due to its inflexibility, it often does not perform as well as COW [van Nederkassel et al., 2006]. Semi-parametric Time

Warping (STW) optimizes spline-based warping functions and is claimed to be more efficient than COW while producing similar results [van Nederkassel et al., 2006, Gong et al., 2004]. The Fuzzy Warping and Robust Point Matching approaches assign fuzzy correspondences between intensities or features in different samples and then optimize a function to minimize their time differences using the correspondences as weights [Walczak and Wu, 2005, Kirchner et al., 2007]. These methods rely on the questionable assumption that similar features, components in our case, occur at similar times. The xcms package adopts a local regression approach, based on the loess model, to fit the deviation of each peak from its group median [Smith et al., 2006]. This likely performs similarly to the spline-based smoothers. SpecArray [Li et al., 2005b] and msInspect [Bellew et al., 2006] employ robust regression algorithms to map retention times in the presence of feature mismatches.

An important consideration when fitting a time warping function is the choice of reference sample. Many simply choose a sample at random to be the reference. This may be dangerous, as choosing a distorted sample as the reference will likely distort every aligned sample. Others derive a synthetic reference sample. Averaging over the samples is a simple technique [Johnson et al., 2003], but it likely results in a flat reference. The xcms package [Smith et al., 2006] synthesizes a reference feature set from the median time in each feature group, which can be problematic in the likely case that all of the groups do not contain features from the same set of samples. One could also imagine performing a cyclical pairwise alignment, thus avoiding the choice of a single reference, but this would have a high computational cost.

### 3.5.2 Our Approach

Our algorithm directly addresses the reference choice issue. As in the matching step, we perform a post-order traversal of the tree formed by the experimental design factors. This means we start with the replicates and work up from there. This strategy is meant to improve our choice of the alignment reference and relies on the assumption that the matchings are more reliable between replicates, since they are likely to have more features in common with each other than with samples from different conditions. For the current set of samples, a

reference sample is chosen that has the maximal sum of cluster sizes for the clusters to which its components belong.

When not enforcing a narrow time window, our matching algorithm produces a significant number of mismatches, so it is important that our alignment method is robust. We have chosen the model employed by the Robust Baseline Estimation algorithm, described in section 3.1. Instead of estimating the baseline of the intensities, we fit the model to the absolute differences in time between each of the matched components for the current sample and the reference sample. This assumes that correctly-matched components are close in time relative to the mismatches and the absolute values of the differences collectively form a “baseline.” A second loess model is fit to the actual time differences using the weights from the robust fit. Figure 17 shows the pair of loess fits for an alignment between two control replicates. The robust loess model ignores the outliers and, as shown in the zoomed plot at the bottom of the figure, still manages to fit a slight shift in time. These steps are performed for each sample in the set, and the fitted values and weights are passed up as input to the parent call in the recursion. Preserving the weights is a key feature of the algorithm, as it prevents mismatches from affecting subsequent alignments. This is meant to counteract the detrimental effect of sample heterogeneity that likely increases as the algorithm works its way back up to the root and the sample set grows larger.

### 3.5.3 Results

Figure 18 graphically depicts the retention time correction on the chromatographic profiles. The corrected profiles are predicted by a loess smoother that models the corrected component times by the uncorrected times. Regions of the profile that are beyond the time range of the components are set to the predicted value for the first or last component, depending on the extreme. For the sake of clarity, the time scale in the figure is limited to the region that appears misaligned in Figure 17. The algorithm apparently improves the alignment of the samples in that region.

Having corrected the retention time in a manner that is robust to mismatches, we repeat

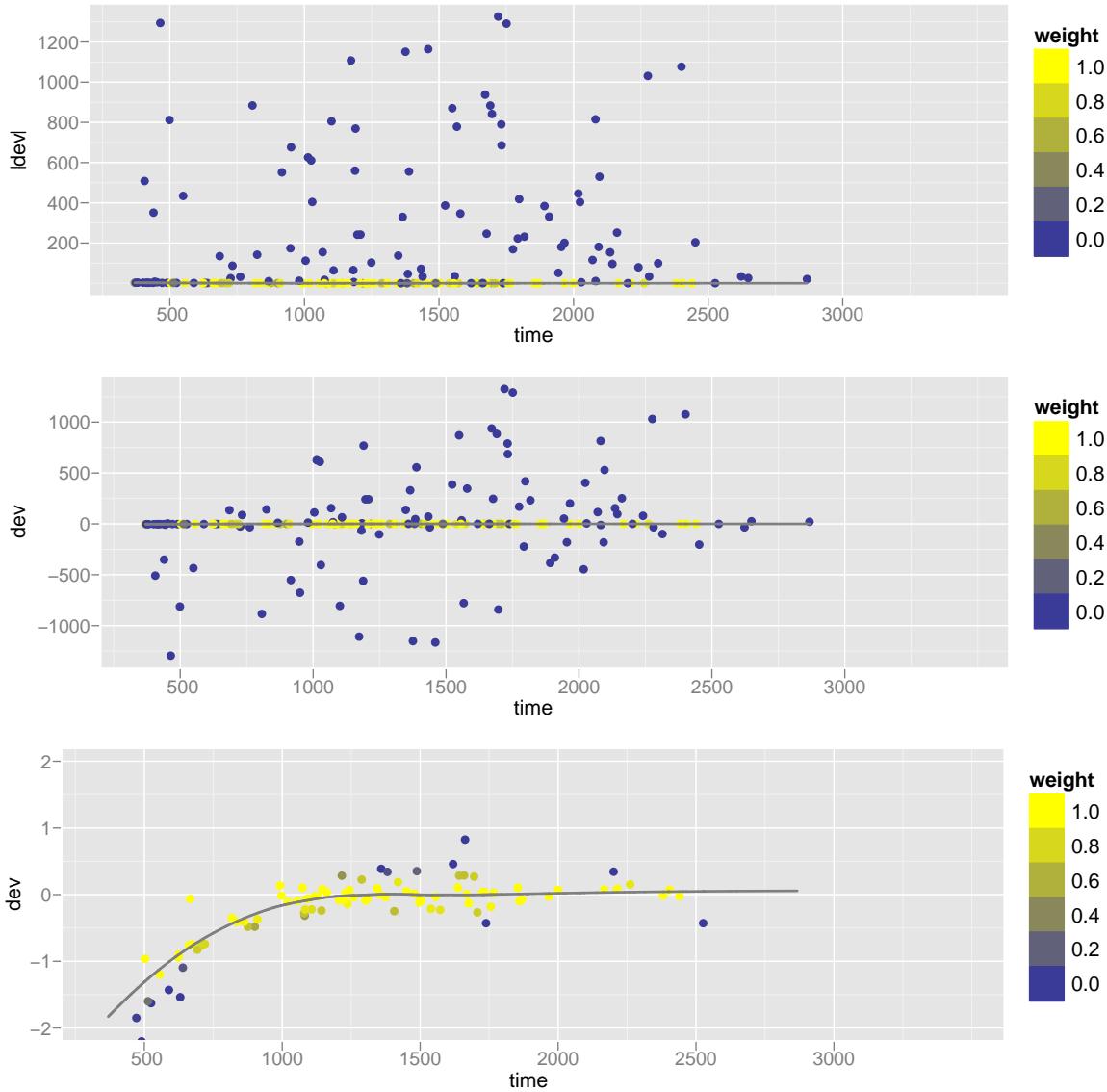


Figure 17 Loess fits for aligning two control replicates in time. The top plot shows the robust loess model fit to the absolute value of the differences in time between grouped components. The second plot is of the loess fit to the actual differences, using the case weights output by the robust fit. The model fits appear flat in the first two plots, but rescaling the vertical axis from  $[-2, 2]$ , as in the bottom plot, reveals that a slight retention time shift is fit by the model at the beginning of the time course. The color scale indicates the weight of each case used in fitting the model.

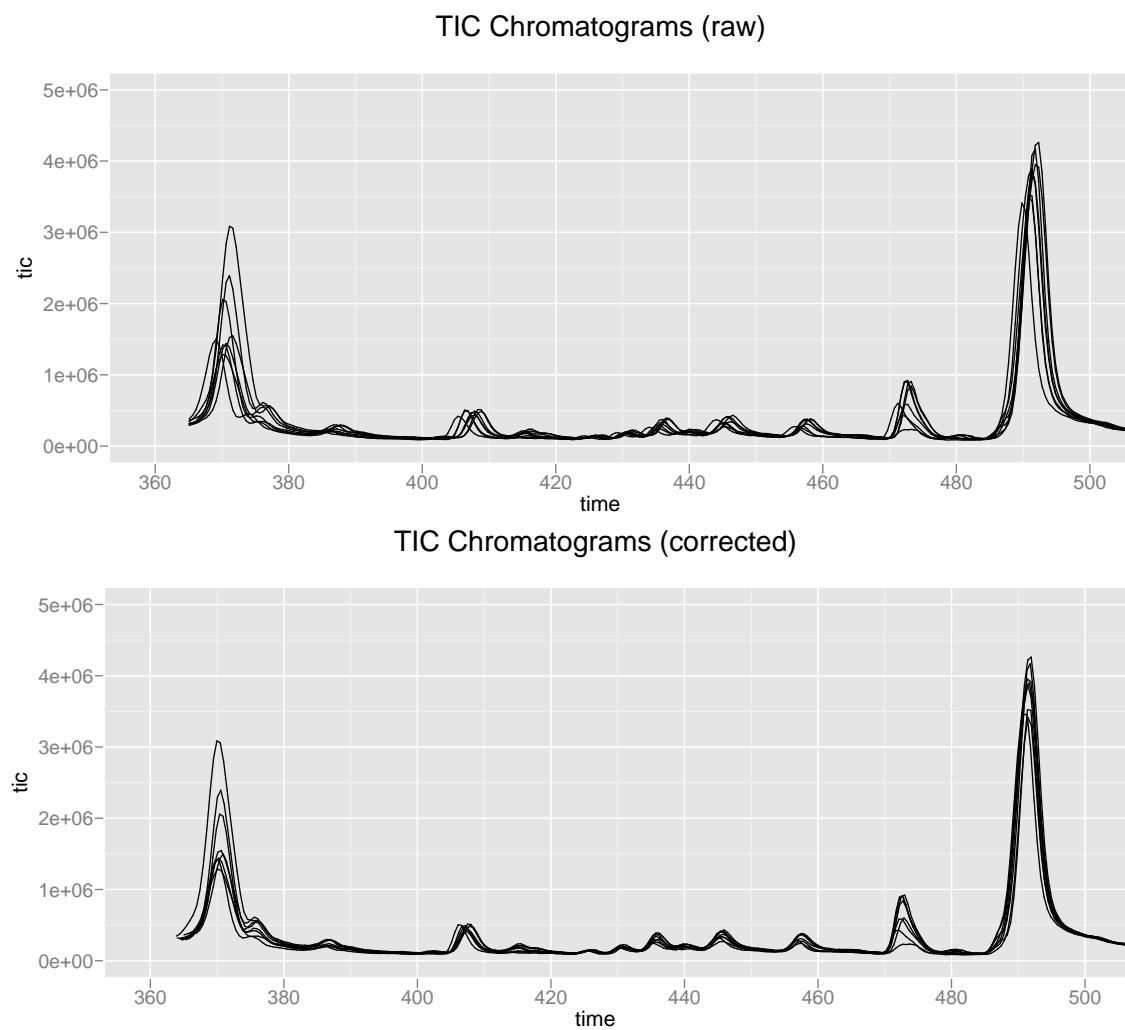


Figure 18 Overlaid Total Ion Current (TIC) chromatograms (intensities summed over  $m/z$ ) for every sample using the raw times (top) and corrected times (bottom). The time range is restricted to 360 seconds to 500 seconds in order to focus on the region that appeared misaligned in Figure 17. The retention time correction procedure has apparently improved the alignment of the samples in that time range.

the component grouping stage with the additional constraint of a time window. This strategy is very similar to that of msInspect. Based on observations of the alignment of the samples in time, we have chosen a retention time window of one second. The barchart in Figure 19 illustrates how the distribution of the component counts of the groups changes after adding the time window constraint. Many multi-component groups appear to have been broken down into singletons. It is likely that many of those groups consisted of mismatches. Interestingly, the number of groups with components from all eight of the samples increased by approximately two-fold. We assign a high confidence to these groups, as they are represented in every sample.

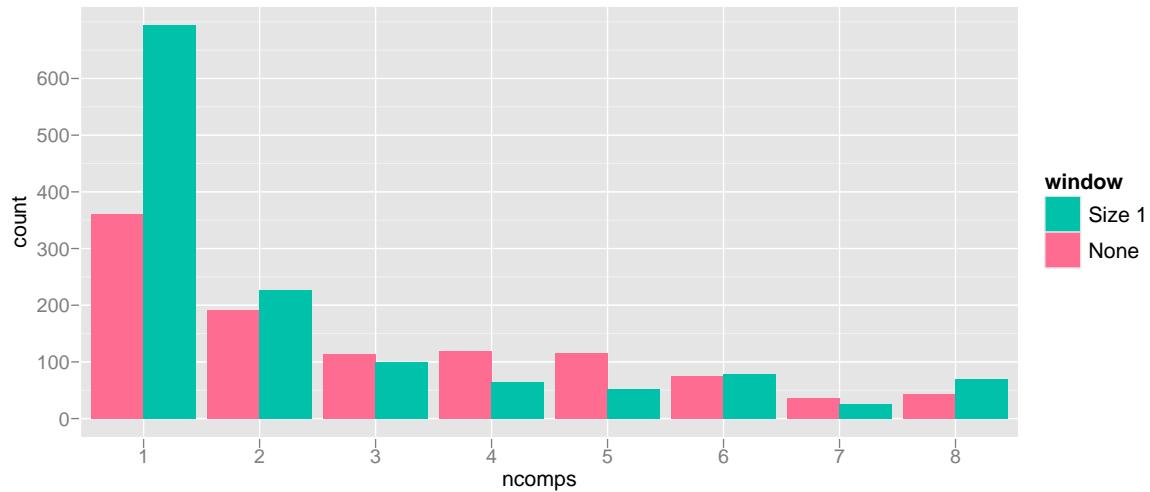


Figure 19 Barchart comparing the distribution of the group sizes before (None) and after (Size 1) applying the retention time window constraint of size 1. The number of singletons increases dramatically, as does the number of groups with components from all eight of the samples.

By assigning each component to a group, we have determined the number of rows, each corresponding to a group, in the result matrix. The rest of the pipeline aims to fill the matrix with values corresponding to the quantities of the components, each of which corresponds to a combination of group and sample.

### 3.5.4 Graphical Diagnostics

For diagnosing the retention time correction, we have developed a visualization for identifying and diagnosing profile regions that the algorithm failed to align. To show the overall result of the retention time correction, the top plot in Figure 20 overlays the chromatographic profiles from every sample. Below that is a scatterplot of the components from a single sample. The plot shows how the time correction, corrected minus raw, varies over the raw times. A line represents the loess model that predicts the corrected times. Below this is the same plot as the one in the middle of the visualization for the grouping stage. It plots components by sample and time with lines communicating the groups.

By moving the mouse pointer over a particular sample profile in the top plot, the plot below is updated to show the model fit on the components for that sample. From this, the user can check the fit and identify any points that may have adversely influenced the model. If the user moves the pointer over a component in the second plot, the group to which the component belongs is highlighted in the bottom plot. If the component is an outlier from its group, the user has a clue as to why the correction failed. Through the interactive graphics, the user has navigated from an error in alignment at the profile level to the root cause at the level of component matching. With this information, the matching stage could be repeated with the parameter tweaks necessary to correct the error.

## 3.6 Summarization

Assuming that the groups of components are correct, it remains to quantify each group in order to produce the final result matrix. Each component consists of peaks with known areas. Assuming that peak area is linearly proportional to the quantity of the fragment ion responsible for the peak, the problem of component quantification reduces to the summarization of peak areas.

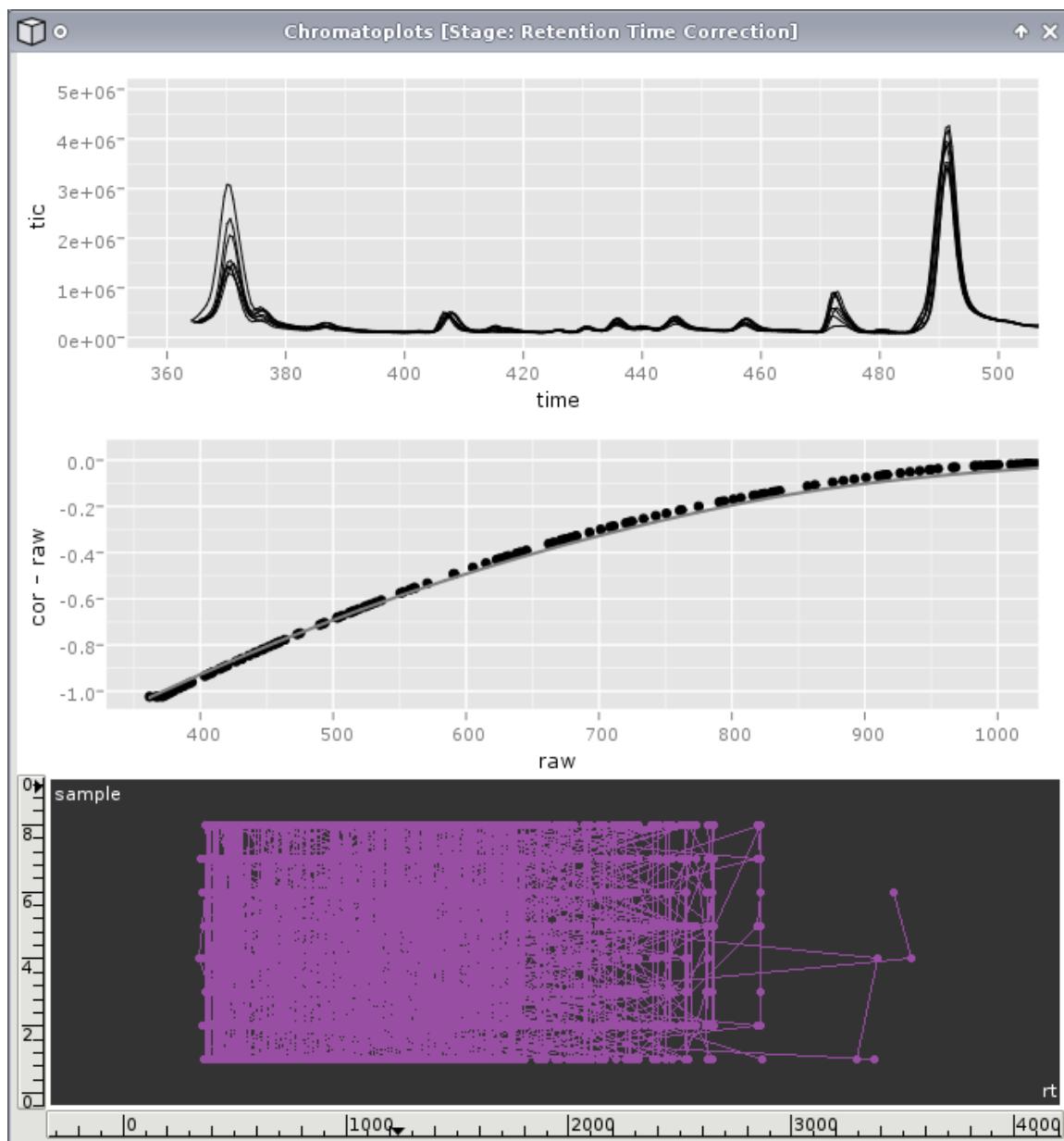


Figure 20 Chromatoplot visualization window for the retention time correction stage. By selecting a profile in the top plot, the user can view the time correction fit for the sample in the plot below. The bottom plot shows how the components are grouped. Selecting a component in the middle plot will highlight the component in the bottom plot and vice versa.

### 3.6.1 Existing Approaches

The MET-IDEA tool [Broeckling et al., 2006] summarizes each component from the area of its tallest peak. This is somewhat justified in that the tallest peak tends to be the best resolved and the most reliably integrated. However, relying on a single peak is risky. Any distortions in the peak are directly carried over to the metabolite quantity.

### 3.6.2 Our Approach

Our method is more robust in that it considers multiple peaks. For each group, the algorithm finds the  $m/z$  values at which every component has a peak. The quantity of each component in the group is obtained by summing the areas of its peaks at those  $m/z$  values. Thus, only the common peaks, as characterized by their  $m/z$  value, are used in the sum.

This strategy aims to avoid the inclusion of peaks that are not generated by the same metabolite. It also helps to minimize the effect of the peak detection cutoff on the quantification. Samples with relatively high signal will have more peaks above the cutoff. Including the extra peaks in the sum would introduce bias and perhaps, due to their small size, inaccuracy.

### 3.6.3 Results

The density estimation of the log transformed metabolite quantities for each of the samples are shown in the left plot of Figure 21. The distributions appear similar, but they are slightly misaligned. Under the assumption that the intensity distributions should be approximately the same, the next and final step of our pipeline normalizes the data in order to correct for the differences.

### 3.6.4 Graphical Diagnostics

The visualization in Figure 22 supports the identification of components with unusual quantities and the examination of peak areas for diagnosing any aberrations. The top plot attempts to expose groups with unusual quantities by plotting them by the average of their component quantities against the maximum difference between any pair of component quantities. To the

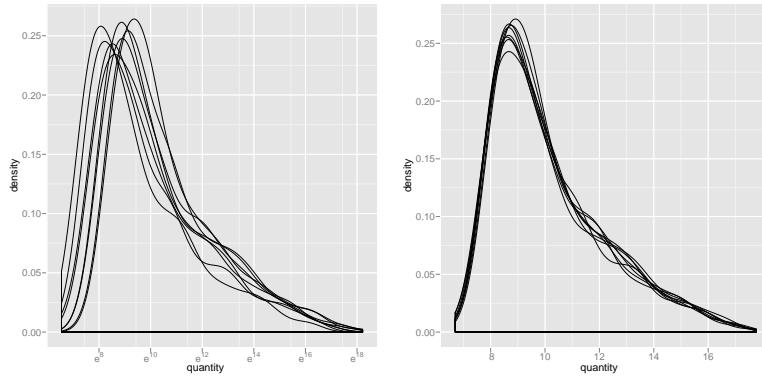


Figure 21 Plots of the density estimation for the log transformed intensities of each sample. The left plot is of the intensities prior to normalization. The distributions are similar but slightly misaligned. The plot on the right is of the normalized intensities. The agreement between the distributions is noticeably improved.

right is a barchart giving the counts of each group that have a particular number of components. Below is a parallel coordinates plot showing the trend for the group across samples. At the bottom is the same compressed mass spectrum display as in the group diagnostic. It shows the intensities of the peaks at  $m/z$  values where there is at least one peak in the group.

The user may interact with the visualization to find and diagnose groups with unusual quantities. Mousing over a point in the scatterplot at the top left displays the trend of the corresponding group across the samples in the plot below. From there, the user may identify the components that are not like the others. Mousing over a point in that plot highlights the peaks of the component in the bottom plot. The user may then observe that the unusual component does not share the same mass spectral signature of the others and may be a mismatch. The chain of interactive graphics has led the user from the group quantities back to the possible errors in the grouping stage.

### 3.7 Normalization

Factors such as errors in the preparation of the sample, injection errors and variation in the sensitivity of the instrument shift the distribution of metabolite levels differently for each

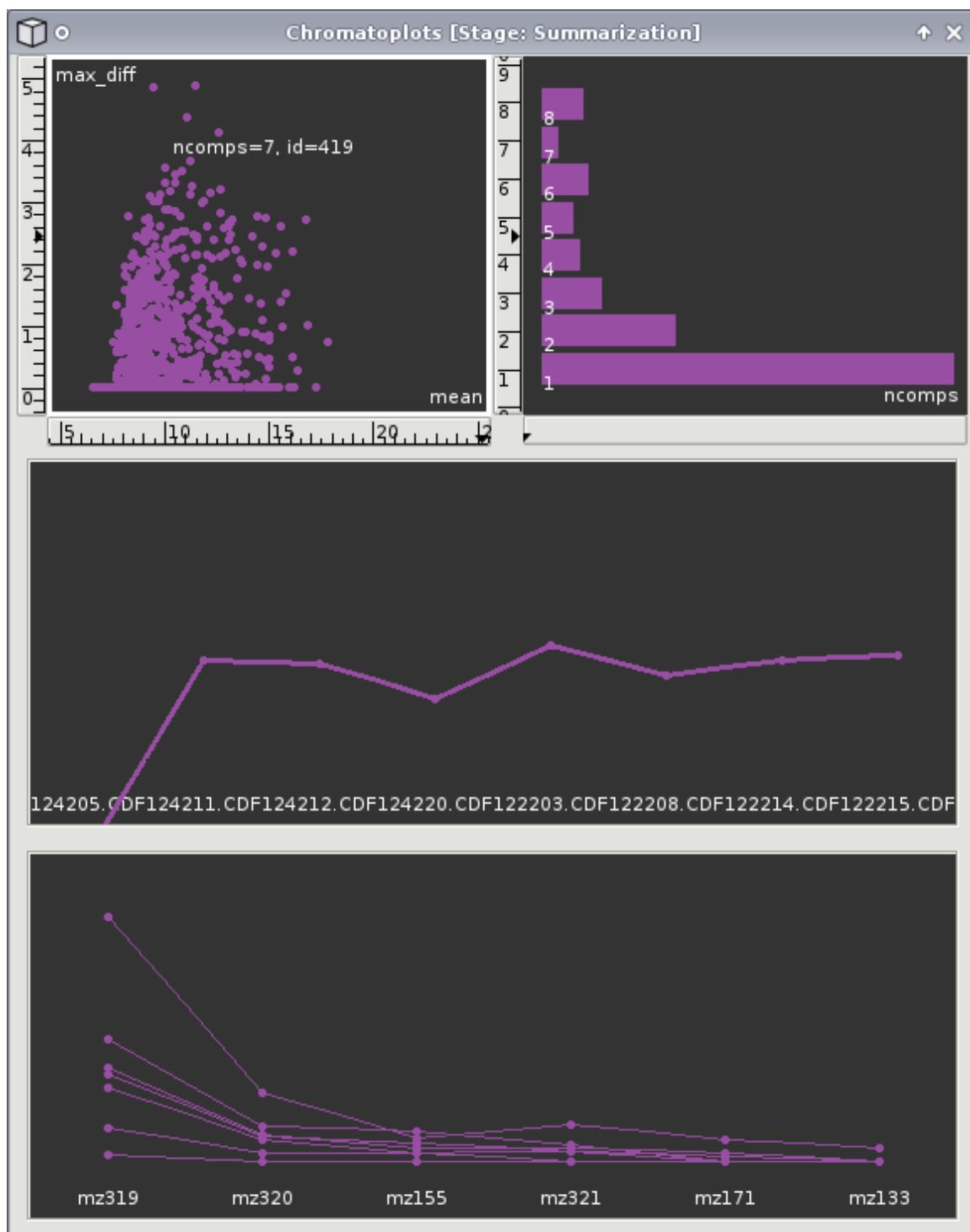


Figure 22 Chromatoplots visualization window for the summarization stage. By selecting a group in the top-left scatterplot, the user can examine the trend of the group over the samples and compare the mass spectra of its components.

sample. Thus, normalization is necessary to compare metabolite levels across samples.

### 3.7.1 Existing Approaches

A traditional method for the normalization of chromatographic profiles is to scale by the ratios of internal standards. This method is often applied to metabolomics data analysis [Syste et al., 2007]. Normalization by internal standard obviously requires the experimenter to add a set of standards to the sample prior to measurement. It is difficult to choose a representative set of standards, as every metabolite responds in a unique way to the chromatography. It is also necessary to identify the components corresponding to the standards in the data. Another drawback is that the approach depends on a small set of components and ignores other information.

Another approach is to scale each sample by the median of the metabolite ratios between the sample and a reference [Wang et al., 2003]. This method assumes that the metabolite with the median ratio is present at the same level in each biological sample.

### 3.7.2 Our Approach

If it were valid to assume that the distribution of the metabolite levels was similar across all samples, one could simply scale by the means. After a log transformation of the data, the distributions appear somewhat similar, as shown in the left plot of Figure 21. The distributions would likely be more similar if the histograms were not truncated on the left by the detection threshold of the mass spectrometer. One proposed solution is to scale by only the top  $k$  intensities in each sample [Wang et al., 2006a]. Assuming the distributions are the same except for the truncation at the threshold, one could assign  $k$  to the minimum number of intensities across all of the samples. However, we have found that the effect of the truncation, at least in this data set, is relatively insignificant. Thus, we assume that the sample distributions are consistent across samples and scale the intensities to the common mean.

### 3.7.3 Results

The right plot in Figure 21 shows the density estimation over the normalized log transformed intensities for each sample. By comparing the plots on the left and right of the figure, one may examine the effects of normalization. It appears that the procedure has improved the alignment of the distributions. The scatterplot matrix in Figure 23 compares the metabolite quantities between each pair of control replicates. The agreement between the replicates is reassuringly high and to some extent is a validation of our methods.

### 3.7.4 Graphical Diagnostics

The normalization algorithm depends on the distributions of the metabolite levels being similar, so chromatoplots displays the histograms before and after normalization, as shown in 21 and provides the same diagnostic visualization as for the summarization stage. This gives the user the opportunity for one last check of the result before exporting it to software designed for exploratory multivariate data analysis.

## 4 Discussion

### 4.1 Peak Convolution

Peak convolution occurs when two or more peaks overlap in time. Most of the time, convolution results in slight errors in the estimation of the peak parameters. In severe cases, however, the peak detection algorithm is not able to detect separate peaks in the convolution. The component to which the convolution is assigned depends on the relative heights of the overlapping peaks. If there are two overlapping components, for example, one component is assigned the convolution while the other is assigned nothing. The former component has an incorrectly estimated peak and the latter is missing a peak. These errors propagate with the components through the rest of the pipeline. An effective peak deconvolution strategy would help avoid these errors.

Since the peaks cannot be resolved in time, deconvolution strategies generally leverage other

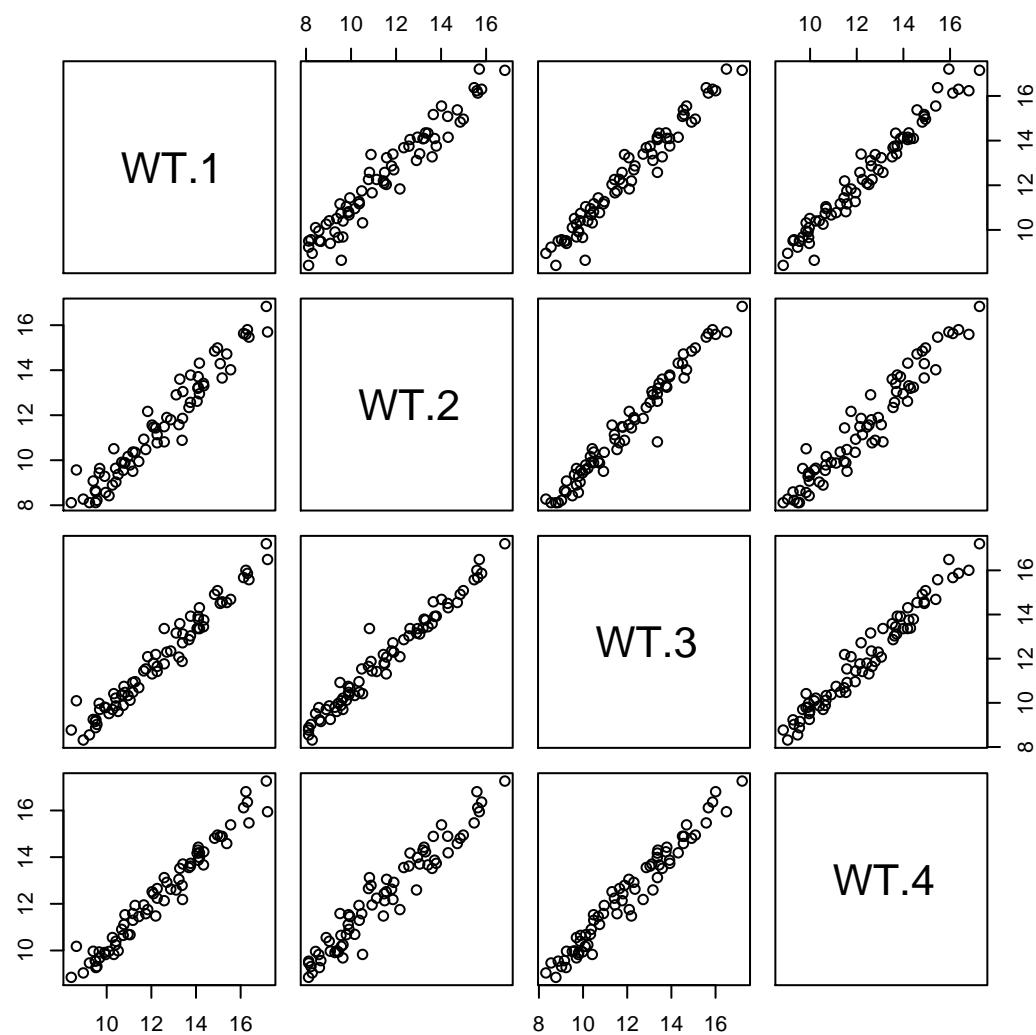


Figure 23 Scatterplot matrix comparing the normalized metabolite levels between the control replicates.

dimensions, such as m/z and sample. AMDIS, which only analyzes a single sample at a time, uses the m/z dimension to resolve components [Dromey et al., 1976]. For each component, the algorithm identifies a sharp peak that is assumed to be a singlet and thus a representative of the true shape of the peaks in that component. For each set of overlapping components, a linear regression is fit to the profile of the relevant region at each m/z with an effect for the model peak of each component, in addition to baseline effects.

The AMDIS approach seems reasonable, but it ignores the sample dimension and fails to separate peaks that are completely overlapped in time. Most metabolomics experiments involve tens to hundreds of samples, so it may be beneficial to leverage that information when deconvoluting the peaks. Many methods have been proposed for the resolution of the components in the three dimensional cube formed by stacking the profile matrix from each sample in the third dimension. This process is generally referred to as Multivariate Curve Resolution (MCR) [TAULER, 1995]. MCR generates two matrices, one describing the levels of each metabolite across the samples and the other containing the mass spectral signature of each metabolite. The first matrix is the goal of our pipeline. The second is useful as the input to a metabolite identification algorithm, a topic which is discussed in the next section. In principle, MCR resolves the raw data to the result in a single step. However, that assumes that the effect of the baseline is minimal and that the samples are aligned in time. It is difficult to apply MCR to data typically encountered in metabolomics, due to the size and complexity of the datasets. One workaround, known as Hierarchical-MCR (H-MCR), segments the data cube along the time axis [Jonsson et al., 2005]. The main drawback to MCR is that the algorithm is somewhat like a black box and is difficult to diagnose.

Another way to leverage sample information for separating components overlapping in time is to consider the pairwise correlations of peak heights or areas at each m/z across samples [Tautenhahn et al., 2007]. Those peaks belonging to the same component should have a fixed ratio in every sample. Clustering by the correlation distances may be a way to distribute coeluting peaks between components. We aim to address the problem of deconvolution in the future.

## 4.2 Metabolite Identification

Although our pipeline has produced a matrix suitable for comparing metabolite levels across conditions, it is difficult to derive biological meaning without knowing the chemical identities of the metabolites. There are two general strategies for identifying metabolites from mass spectra. The first is searching databases of identified mass spectra for a spectrum that closely matches a query spectrum. Spectral distance measures, such as the uncentered correlation, are used to evaluate the quality of a match [Stein and Scott, 1994, Liu et al., 2007]. This is the strategy followed by AMDIS [Stein, 1999]. The main limitation of identification through library search is that there is no library available that contains an acceptable number of metabolite spectra. Given the cost in producing such a library, it would be more feasible to create a public database for contributed spectra.

An alternative to library search is to infer the set of molecular formulas compatible with a mass spectrum [Tobias and Oliver, 2007, Bocker et al., 2006] and possibly even evaluate the correspondence of the spectrum to known chemical structures [Sweeney, 2003]. We plan on investigating these approaches further, though the complexity of metabolomics data will likely overwhelm them.

## 5 Conclusion

This paper has presented numerical algorithms and graphical diagnostics for the pre-processing of GC-MS metabolomics data. The output of the pipeline is an experimental data matrix with rows describing metabolites and columns corresponding to experimental conditions. At each stage, the user may examine the output of the algorithm through interactive visualizations. The system is implemented as an experimental R package named *chromatoplot*.

The development of chromatoplot is in progress. With regard to the methods, the next step is to address the issue of metabolite identification, which is important for assigning biological meaning to the results. This may involve the construction of a database for public submission of spectra or the development of an algorithm for matching spectra to molecular structures. The peak convolution problem will also be investigated further. We aim to validate the methods

on a dataset with a known preprocessed result. In terms of the software implementation, it remains to stabilize the interactive visualizations and to construct the GUI.

## CONCLUSION

### 1 Impact

This work has already achieved significant impact on the research of others. RGtk2, rggobi, GGobi, rsbml and exploRase are publicly available, and their influence is evidenced by publications, derivative work by other researchers, citations, acceptance into peer-reviewed software archives and support requests.

The exploRase package has been applied by several biologists to the analysis of their data. Two examples are detailed in the Appendix. The first analyzes plastic production in *Arabidopsis* mutants, and the other is an investigation of *Arabidopsis* acetyl-CoA metabolism.

GGobi and rggobi have widespread use. In the month of September, 2007, there were 742 downloads of GGobi, including 650 downloads of the Windows installer, which I developed. The number of total downloads is likely an underestimate, because several Linux distributions independently maintain GGobi and rggobi packages. Before the improvements to GGobi and rggobi, which also involved the work of Hadley Wickham, these numbers were similar to those for an entire year. A Google search yields many thousands of real hits, including many course web pages that describe using GGobi for multivariate analysis and exploring microarray data. GGobi is on the first page of results for searches “data visualisation”, “parallel coordinates plot”, and “open source visualisation”. Are the mailing list numbers most recent? There are 136 members of the GGobi help mailing list and 638 messages have been sent since the last reset in February, 2007. I am ranked first in number of posts, with 107 replies to support requests.

There are now several R GUIs available that based on RGtk2: Rattle, pmg and playwith. The *Rattle* data mining tool [Williams, 2006] has been awarded the prestigious Australia Day

Medallion and is used in courses at several institutions including the Australian National University and Yale University. The *Poor Man's GUI* [Verzani, 2007b] is an R GUI designed for teaching R and statistics. The *playwith* package [Andrews, 2007] is an experiment to provide simple interactive features on top of the base R graphics system using RGtk2 and cairoDevice.

## 2 Future work

Although the general theme of this thesis is the application of graphical methods to biological data analysis, each component has a separate and unique focus. As such, there are numerous directions in which this work might progress. This section focuses on the angles with the most potential for improving the project as a whole.

An overall goal of the project is the integration of interactive graphics with numerical methods. The exploRase and chromatoplot applications both emphasize this goal and pursue it by interfacing R and GGobi through rggobi. A major limitation of this strategy is that GGobi is not meant to be a general interactive graphics engine. Rather, GGobi has been designed specifically for multivariate exploratory data analysis. It supports only a small subset of the graphics that may be produced through the R graphics system, which, unfortunately, is not designed for interactivity. While some flexibility is afforded by the extensibility of the GGobi pipeline, the expressiveness of R graphics remains out of reach. The effectiveness of the visualizations in chromatoplot and exploRase would be improved if a wider array of interactive graphics were available. There has been progress on a library that handles low-level tasks, such as incremental plot updates, commonly encountered by interactive graphics applications. This library could serve as the basis for both GGobi and an R package supporting a more diverse array of graphics. The R package will likely evolve from the ggplot2 package [Wickham, 2007b].

Another goal of this thesis and, in particular, the exploRase application, is the integrated analysis of networks and experimental datasets from different sources. Biochemical systems are heterogeneous, consisting of enzymes, metabolites and other factors, and each aspect of the system is measured by a particular experimental technique. This results in a set of datasets, which need to be integrated into a description of the entire system. By specifying the relation-

ships between biochemicals of different types, networks help integrate experimental datasets. This depends, however, on the ability to match experimental measurements to nodes in the network. Unfortunately, biological identifiers often do not follow precise standards. Even mappings between standard identifiers may be ambiguous. The size of experimental datasets prohibits manual mapping, and automation is complicated by imprecise and ambiguous identifiers. Work has begun in collaboration with Xiaoyong Sun towards an automated identifier mapping system that produces diagnostic reports to help the biologist detect errors in the mapping process. The tool has been designed to be extensible to support custom identifier systems and mapping schemes. A GUI for the package will be integrated with exploRase in the interest of convenience and accessibility to biologists. Overcoming the identifier matching problem is key to raising exploRase to the next level of effectiveness.

### 3 Summary

In summary, these are the contributions that my thesis research has produced for the bioinformatics community. I have presented methods for the analysis of biological data through the synergy of interactive graphics and numerical methods. The methods are implemented as a collection of independent software modules, centered around two applications, exploRase and chromatoplot. The exploRase package integrates the interactive graphics of GGobi with numerical methods for the analysis of preprocessed experimental data. This analysis is supported by interactive, biology-aware network visualizations, which have grown out of research in the field of interactive graph layout. The rcola package is the direct result of that research. Chromatoplot applies interactive graphics to the diagnosis of its preprocessing algorithms for GC-MS metabolomics data. Both chromatoplot and exploRase make analytical methods accessible, especially to biologists who are often not proficient in the R language. The goal of accessibility drove the development of RGtk2, which supports the construction of GUIs in R. RGtk2 is also a manifestation of research on the interfacing of software systems. The other software interface projects are rggobi, rsbml and rcola.

## APPENDIX

### Data Analysis Examples

#### **An analysis of microarray time course data investigating response to light stress in *Arabidopsis***

Michael Lawrence<sup>1</sup>, Heike Hofmann<sup>2</sup>, Dianne Cook<sup>2</sup>, Eve Wurtele<sup>3</sup>, Ron Mittler<sup>4</sup>

<sup>1</sup>Bioinformatics, <sup>2</sup>Statistics, <sup>3</sup>Genetics, Development and Cellular Biology, Iowa State University

<sup>4</sup>Biochemistry and Molecular Biology, University of Nevada - Reno

[{lawremi, hofmann, dicook, mash}@iastate.edu](mailto:{lawremi,hofmann,dicook,mash}@iastate.edu), [ronm@unr.edu](mailto:ronm@unr.edu)

#### **Abstract**

The survival of a plant depends on its ability to rapidly respond to its environment. The pathways involved in responding to biotic and abiotic stress appear to be dependent on reactive oxygen species (ROS) signaling networks. We obtained a microarray time course dataset investigating the dependence of the light stress response pathway in *Arabidopsis* on the H<sub>2</sub>O<sub>2</sub> removal enzyme cytosolic ascorbate peroxidase (APX1). The experiment compares wildtype plants to mutants lacking a functional APX1 gene. The original analysis of the data focused on the general genotypic differences in gene expression across the duration of the stress exposure. We believe an analysis concentrating on the genes that show significant response to light stress will help to further clarify the role of ROS signaling in plant stress response. We employ a conventional polynomial model to estimate the time dependence of each expression profile,

separately for the two genotypes. Throughout the study, the data is visualized in conjunction with analysis results using exploRase, a program for the exploratory analysis of systems biology data. A large portion of the genes found to be time dependent are thought to participate in ROS signaling or other stress-related pathways, including anthocyanin biosynthesis. Many of these genes no longer respond in the KO-APX mutant, but general stress response pathways involving abscisic and jasmonic acid appear to be up-regulated and might act to compensate for the loss of APX.

## Introduction

Plants respond to many different types of stress, including drought, heat, cold, and light. It is gradually becoming clear that many plant response pathways are bound by a single thread: reactive oxygen species (ROS) [Mittler, 2002, Mullineaux and Karpinski, 2002, Apel and Hirt, 2004]. ROS signaling is often activated in response to a particular stress [Pnueli et al., 2003, Foreman et al., 2003]. In plants, the chloroplast usually initiates the ROS signal, due to the high sensitivity of the photosynthetic apparatus to environmental conditions like light, temperature, and moisture [Foyer and Noctor, 2003]. The stress-induced ROS signal from the chloroplast may be amplified by membrane-bound  $H_2O_2$ -producing NADPH oxidases, perhaps regulated in part by the key stress response hormone abscisic acid (ABA) [Kwak et al., 2003]. It is thought that the ROS signal influences transcription via the oxidation of participants in other signaling pathways, such as mitogen activated protein (MAP) kinase cascades [Kovtun et al., 2000]. It is, at least in part, through this regulation of transcription that a plant adapts to a stressful environment.

In addition to stress, many different processes in the plant, including stomatal closure, development, regulation of photosynthesis, apoptosis, and pathogen defense are thought to be controlled to some extent by ROS signaling [Apel and Hirt, 2004, Mullineaux and Karpinski, 2002, Pnueli et al., 2003, Foreman et al., 2003]. Thus, ROS signaling seems to be a means of cross-talk between stress responses and other pathways and may serve as a “biochemical alarm”. It announces that the environment of the plant has become hostile and that the plant

should take extraordinary means to defend itself.

Excess light leads directly to enhanced photosynthetic activity and thus production of ROS. As ROS are extremely deleterious to the cell, evolution has afforded the plant means to control the ROS accumulation. Enzymes, such as superoxide dismutase (SOD) and ascorbate peroxidase (APX), scavenge ROS, mitigating cell damage even when the photosynthetic rate is extremely high. The different isoforms of APX are localized to specific regions of the cell. APXs in the chloroplast stroma (also found in the mitochondria), lumen, and thylakoid membrane [Mittler, 2002]. There are two APX isoforms in the cytosol, APX1 and APX2, but APX2 is only induced under extreme oxidative stress conditions [Karpinski et al., 1999]. Although the peroxidases in the chloroplast should be more than capable of protecting photosynthesis during periods of moderate light stress, photosynthetic activity is cut in half when APX1 function is lost [Mittler et al., 2004]. It seems that the function of APX1 may extend beyond protecting the cell from the damaging effects of H<sub>2</sub>O<sub>2</sub>. By modulating H<sub>2</sub>O<sub>2</sub> levels, APX1 may regulate a complex signaling network [Davletova et al., 2005, Neill et al., 2002].

A recent study by Davletova et al. [2005] investigated the function of APX1 in *Arabidopsis thaliana* during moderate light stress. Davletova et al. asked whether removing APX1 would affect the stress response. To answer the question, they performed a microarray time-course experiment across seven time points in which both wildtype and APX1 knock-out mutants (KO-APX) were subjected to light stress. The authors performed ANOVA on the results to determine the effect of genotype, time and their interaction on the expression of each gene. They specifically searched for genes with expression levels that responded differently to light stress between wildtype and KO-APX. They found that the hydrogen peroxide signaling network was completely disrupted in the KO-APX plants. They also noticed that many other pathways were affected by the mutation, including several involved in response to stresses other than light.

The observations of Davletova et al. [2005] are interesting in the light of the role of ROS signaling in cross-talk between stress response pathways. Our aim is to validate the previous results through an analysis that integrates numerical methods with interactive graphics. Like

the original authors, our analysis will search for expression patterns that appear to be changing in direct response to light stress. We believe such an analysis will yield additional insights into the role of ROS signaling as a mediator between stress response pathways in *Arabidopsis*.

A common approach to the analysis of Affymetrix microarray data consists of several stages. First, the raw data is preprocessed. This includes calculation of expression values and their normalization. The processed data should then be validated to ensure that it has no obvious systematic problems. This often involves exploratory visualization. After validation, one looks for genes with expression patterns that are highly dependent on the experimental conditions, such as genotype. Knowing that the expression of a gene is dependent on certain conditions allows one to make guesses about its role in the cell. For example, if a plant is exposed to a period of stress and a gene is strongly dependent on time, one might expect the gene to be involved in stress response. Linear models are a useful tool for estimating these treatment effects. At the same time, one looks for general expression patterns that are shared among subsets of genes. Clustering methods are often employed toward this end. Genes with high similarity in their expression patterns are more likely to participate in the same cellular process. For example, genes with similar expression patterns under stress might be grouped into the same stress response pathway. Such hypotheses may then be tested back in the lab. Our data has already been preprocessed, so we begin with the validation step and then search the data for differential expression and other patterns.

## Results

### Data Validation

Davletova et al. extracted the mRNA from wildtype and APX1 knockout plants at seven different time points during the exposure to light stress. There were two replicates for each of the 14 different combinations of genotype and time, for a total of 28 Affymetrix 25k chips. The data we were given had already been normalized by RMA, so we will begin with the second step in the microarray analysis process: visual exploration and validation. Specifically, it is necessary to evaluate the agreement between replicates. The variance between replicates may

be judged relative to the variance between genotypes and times. Figure 1 contains a scatterplot matrix comparing the replicates at two time points (0m and 15m) from the wildtype. It first appears that the replicates for both time points agree fairly well. However, it appears that the scatterplot comparing the first replicates of two time points are more correlated than the replicates at 0 minutes. This is somewhat suspicious but not very surprising given that we would expect the 0 and 15 minute time points to be very similar. The comparison of the 0 and 90 minute times in Figure 2 reveals another inconsistency. The scatterplot comparing the replicates at 90 minutes does not match the structure of the others. Nor does the histogram for the first 90 minute replicate resemble those of the other chips. This is likely a relatively harmless artifact from the normalization step, but this aberration should be recalled in subsequent analyses.

### **Mining time-dependent profiles in the wildtype**

The cursory visual inspection does not identify any significant problems with the data, so the focus of the analysis shifts to finding genes with interesting expression patterns. The first criterion for an interesting pattern is time dependence, which indicates that the expression of a gene may be tied to light stress. It is possible to compare expression values between pairs of time points using scatterplots, as with the replicates, but deriving patterns based on numerous pairwise comparisons would be tedious and error prone. Parallel coordinate plots are not an option either, because with over 20000 genes there is far too much overplotting. A possible solution would be a method that detects patterns automatically.

ExploRase, a software package for the exploratory analysis of systems biology data [Wurtele et al., 2003], provides rudimentary pattern finding functionality. It decides based on quantiles (assuming a normal distribution) whether a profile is increasing, decreasing, or staying level between each pair of time points. It is then possible to query exploRase for a certain pattern, such as one increasing for all six transitions in the dataset. Figure 3 shows the results of this analysis on the data where the replicate pairs have been averaged. ExploRase identifies several profiles that increase throughout the stress exposure. The next step is to find which patterns

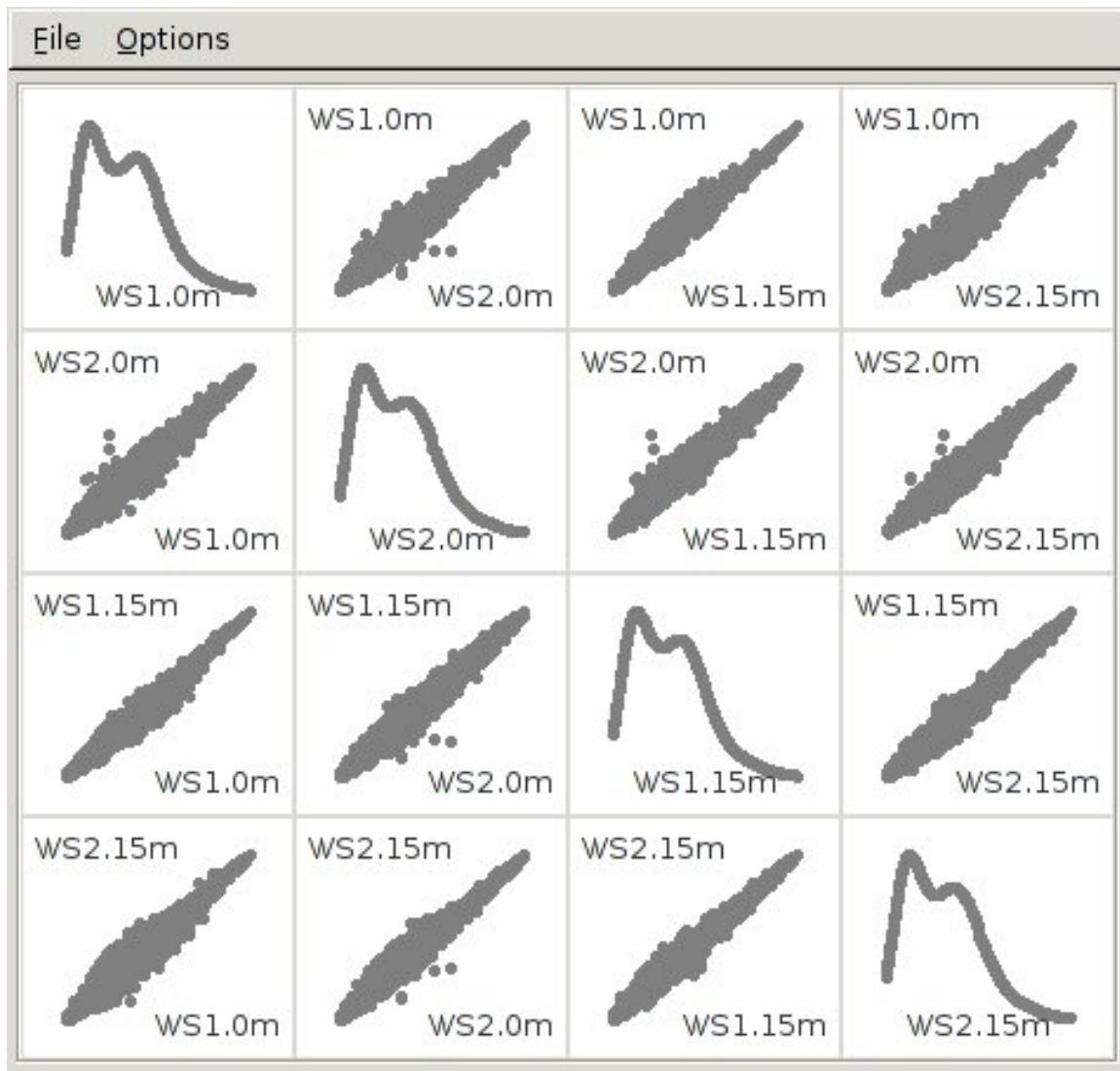


Figure A.1 Scatterplot matrix comparing the replicates at 0 and 15 minutes in the wildtype. The replicates generally agree, but there is also a high degree of similarity between the time points.

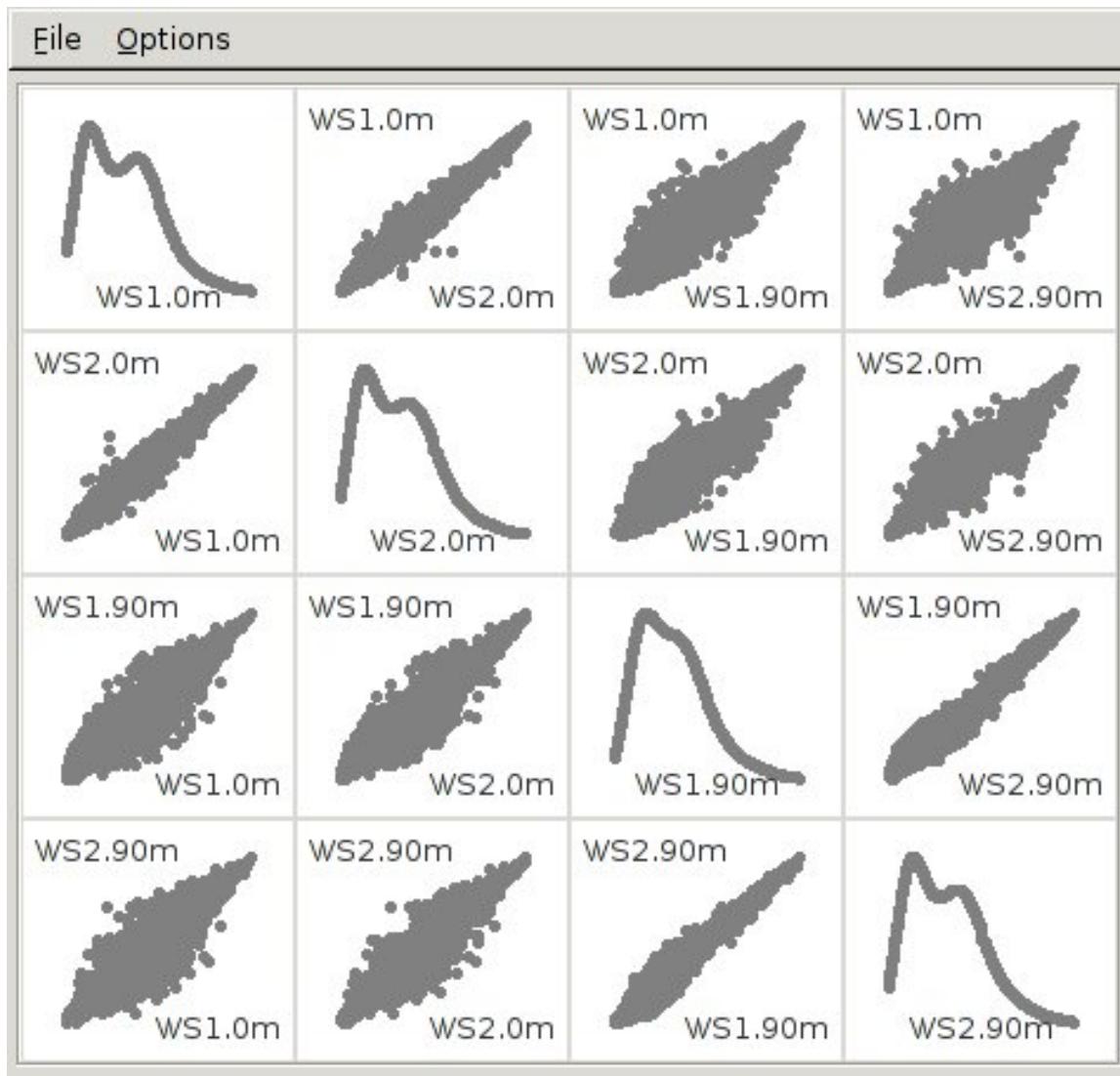


Figure A.2 Comparison of the replicates at 0 minutes and 90 minutes in the wildtype. Note the abnormal shape in the plots including the first 90 minute replicate.

are the strongest. The profiles have passed a quantile test, but there is little indication of the relative strength of the patterns. ExploRase allows the user to sort the pattern column by the magnitude the pattern, which is defined as the sum of the absolute values of the differences across the time transitions. The sorted results, shown in Figure 4, indicate that the constant rising pattern is not the only one of interest; there is an assortment of different patterns at the top.

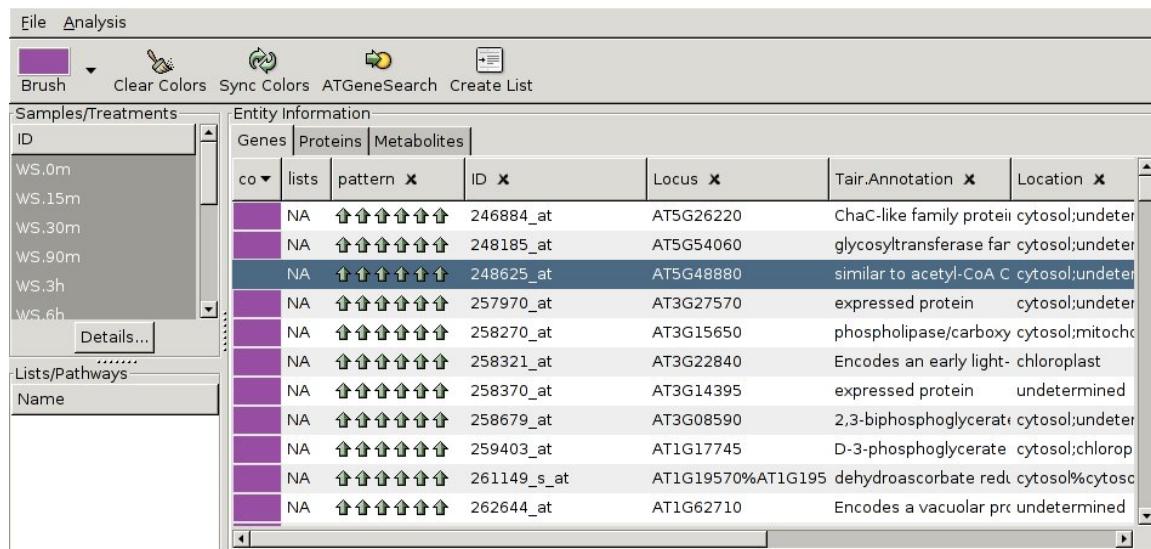


Figure A.3 A screenshot of exploRase with the profile patterns drawn using arrows in the “pattern” column in the large table. Each row in the table corresponds to a single probe on the Affymetrix chip. The table has been automatically sorted so that genes with the pattern of interest (constantly increasing) are at the top.

It is desirable to ask more sophisticated questions about the pattern-finder results. For example, under the assumption that entire pathways are being activated in response to stress, one might expect to see a disproportionate number of genes changing at a certain time. ExploRase allows searching for each pattern and counting the number found, but this is obviously time consuming and dull. Furthermore, by relying on the data averaged over the replicates, the exploRase pattern-finder ignores variance between replicates. If a profile has little agreement between replicates, the validity of the pattern derived from the means is cast in doubt.

A linear model may be one means of circumventing these shortcomings. The first step is

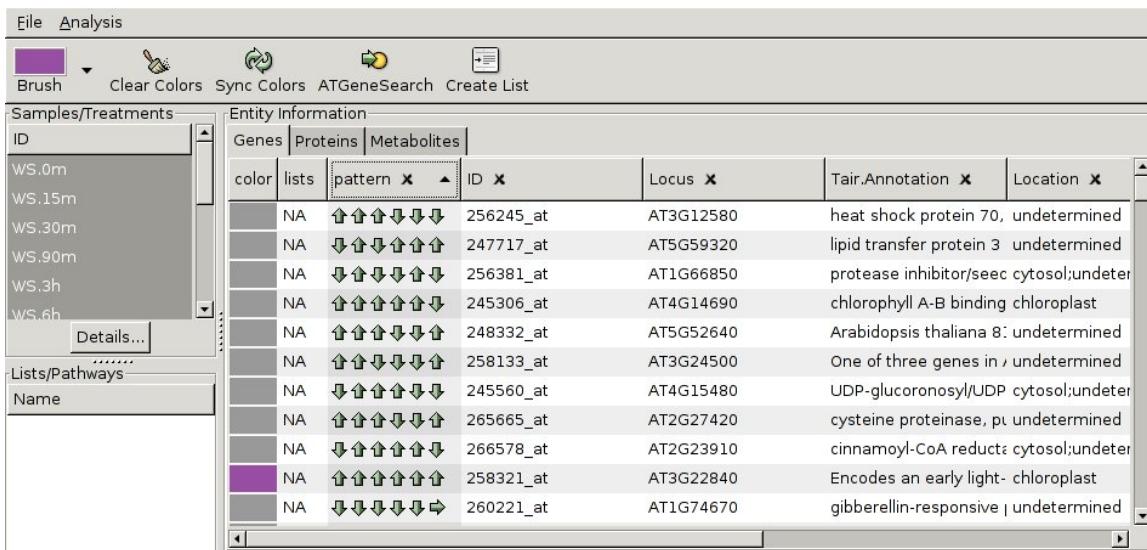


Figure A.4 A screenshot of exploRase where the gene table has been sorted according to the “magnitude” of the pattern, which is the sum of the absolute values of the differences between all the time points. Note that no pattern seems to stand out.

to choose a model formula that evaluates the time dependence of the expression values. A first order polynomial in time (a straight line) would not be sufficient, since it is assumed that many profiles will not respond immediately to the stress and that many trends will not continue throughout the time course. Many profiles will either reach a new steady state or return to their previous state. Thus, a third order polynomial seems a reasonable choice, yielding the formula  $y = b + it + jt^2 + kt^3$ , where  $b$  is the intercept, and  $i$ ,  $j$ , and  $k$  are the coefficients for the first, second, and third order time effects, respectively. The time variable  $t$  is treated as virtual time, meaning that its values belong to the set of integers  $\{1, 2, \dots, 7\}$  instead of the actual times, since we assume that the experiment was designed such that the amount of change between the initial time points is expected to be about the same as that between the later time points.

The model is first fit on the wildtype profiles, to see which genes (and pathways) are responding to the light stress. Only those profiles that have at least one significant time coefficient out of the three are of interest. Figure 5 contains a scatterplot of the coefficient vs its p-value for each of the three time coefficients. Those profiles with high magnitudes for the coefficients but low p-values are considered interesting and are brushed with colors

corresponding to the pattern of dependence across the three coefficients (see caption). It is clear that the coefficients have a  $+,-,+$  pattern, such that if  $i$  is positive,  $j$  is negative and  $k$  is positive. If  $i$  is negative, the opposite pattern occurs. Those profiles that are significant for all three coefficients are colored according to their pattern in the parallel coordinate plot shown in Figure 6. One might notice that the brown lines tend to rise to a higher steady state around 90 minutes and the green lines seem to drop around the same time. One expects a certain amount of state transitions to be occurring at all the time points, but it seems that an abnormally high number of profiles are changing state at 90 minutes. It also appears that the system is returning to its original state at 24 hours.

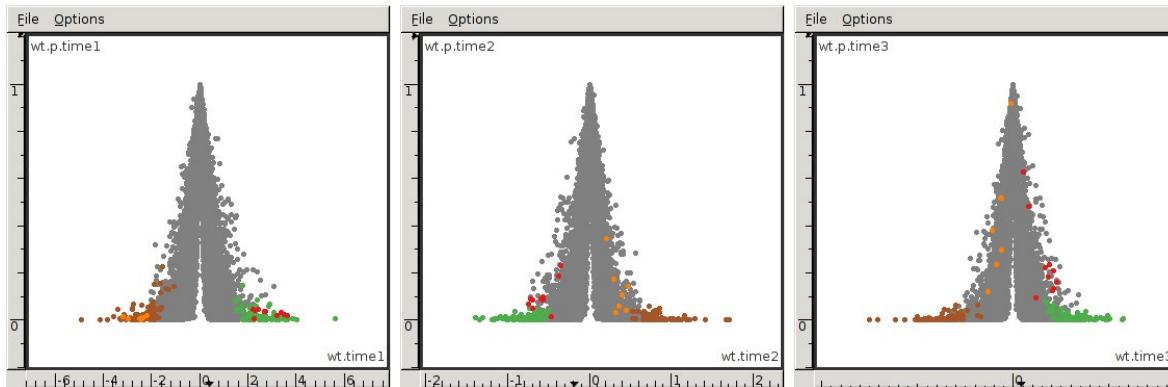


Figure A.5 Scatterplots of each of the time coefficients ( $i$ ,  $j$ , and  $k$ ) vs their respective p-values, in order of increasing degree of time. The gene colors indicate the sign pattern for the coefficients as well as whether the gene has significant effects for all three orders of time. Specifically, the red and orange genes quickly lose significance as the order of the time term increases, while the green and brown genes remain significant.

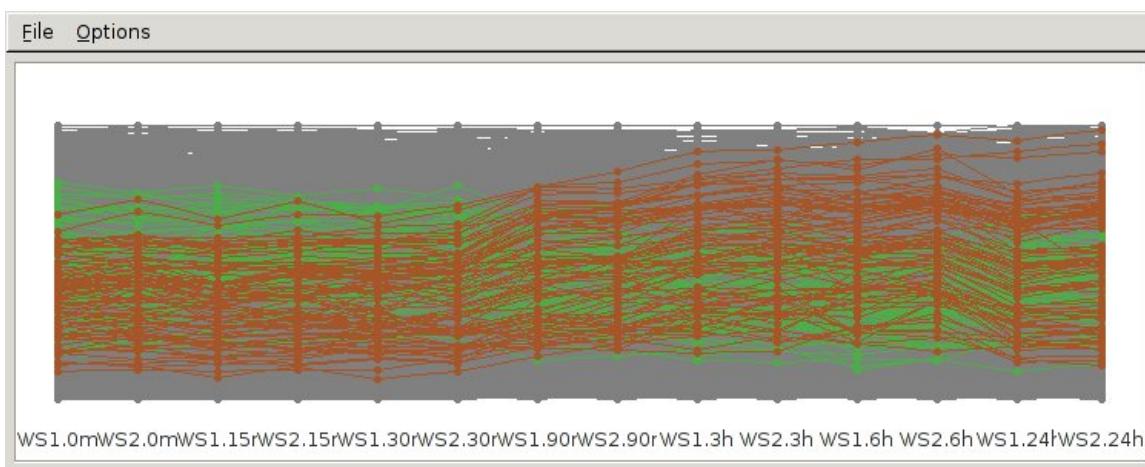


Figure A.6 A parallel coordinate plot of the genes with significant effects across all orders of time, colored by the sign pattern of their coefficients. There seems to be a state shift around 90 minutes and indications of a reverse shift around 24 hours.

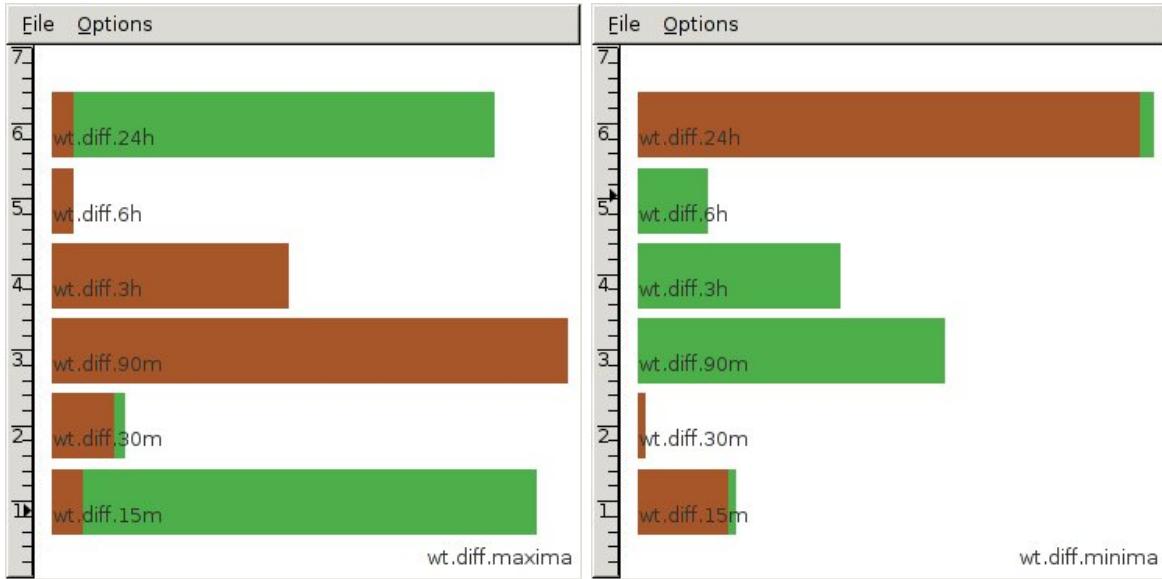


Figure A.7 Barcharts of the maxima and minima of the set of  $t$ -statistics for each gene. The brown genes seem to be increasing mostly at the 90 minute and 3 hour time point and decreasing at 24 hours. The green genes seem to have almost the opposite behavior.

This observation relies solely on the perceptive power of the brain, but it is possible to reinforce it numerically. We propose a method based on the  $t$ -statistic:

$$u_m = \frac{x_{1m} - x_{2m}}{\sqrt{\frac{s_{1m}^2}{2m} + \frac{s_{2m}^2}{2(7-m)}}}$$

where  $m \in \{1, 2, \dots, 6\}$  and represents the time transition across which sequential time points are grouped, so  $x_{1m}$  is the mean of  $\{y_1, \dots, y_m\}$  and  $x_{2m}$  the mean of  $\{y_{m+1}, \dots, y_7\}$ . Informally, for each gene we are splitting the time points into two sets at each transition and calculating the  $t$ -statistic between them. The  $t$ -statistic increases as the expression difference increases and variance within groups decreases, so a large value indicates a significant state transition. We define  $m_{max}$  and  $m_{min}$  to be the transitions at which the  $t$ -statistic is maximal and minimal. Figure 7 shows the barcharts of  $m_{max}$  and  $m_{min}$  for the green and brown profiles. As expected, the numerical method suggests that almost all of the brown profiles reach a higher state at 90 minutes or 3 hours and drop down again at 24 hours.

Table 1 lists the genes with the profiles that are shifting to a higher state at 90 minutes. Figure 8 contains a bar chart describing the distribution of the biological functions for those genes. Many of the genes are involved in flavonoid synthesis, which is likely leading to the production of anthocyanins to counteract the oxidative stress. This is possibly being stimulated by cytokinin, as the up-regulation of several cytokinin synthesis genes [Deikman and Hammer, 1995] is observed. There also appears to be an increase in gibberellin 2-oxidase transcription (at 30 minutes), which likely leads to negative regulation of gibberellin. Overall, a high percentage of the selected genes are thought to be involved in stress response. Most of the others are either involved with photosynthesis or have no known function.

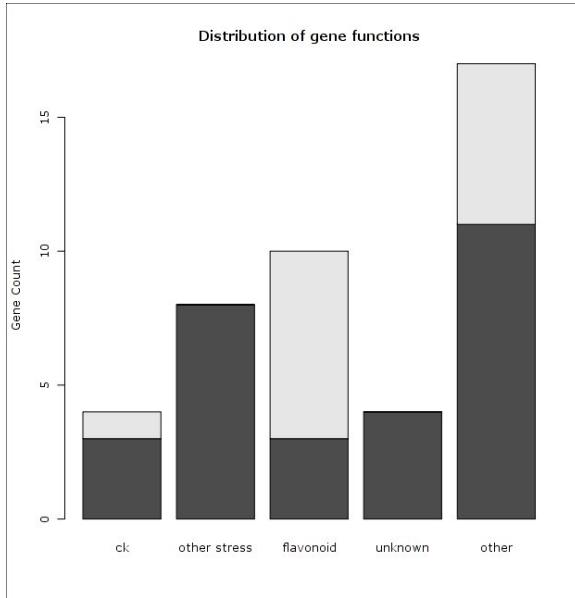
Table A.1 Identities and functions of the genes identified to be shifting to a higher state in the wildtype after 90 minutes of light stress. The last column indicates whether the gene responded to light stress in the mutant. Highlighted rows indicate those involved in stress. Genes with unknown function are omitted to save space. The genes listed here also passed a visual filter in which they were plotted individually in order to verify that their pattern was significant. The information was derived from AtGeneSearch [Wurtele et al., 2003].

Locus Name	Identity	Function/Pathways	KO-APX
AT4G14690	chlorophyll A-B binding protein	<i>light induced, photosynthesis</i>	Yes
AT4G15480	UDP-glucosyl transferase	<i>cytokinin synthesis</i>	No
AT1G56650	MYB75	<i>anthocyanin synthesis</i>	Yes
AT1G57590	pectinacetyl esterase (similar)	unknown	No
AT5G60540	glutamine amidotransferase complex	histidine biosynthesis	Yes
AT5G56090	cytochrome oxidase	unknown	No
AT5G13930	chalcone synthase, CHS	<i>anthocyanin synthesis, wounding, jasmoate</i>	Yes
AT5G08640	flavonol synthase, FLS	<i>flavonoid synthesis</i>	Yes
AT5G05270	chalcone-flavanone isomerase	<i>anthocyanin synthesis</i>	Yes
AT3G55120	chalcone-flavanone isomerase, TT5	<i>anthocyanin synthesis, UV</i>	No
AT3G53260	phenylalanine lyase, PAL2	<i>flavonoid synthesis, wounding, ROS</i>	No
AT3G1240	flavanone 3-hydroxylase, F3H	<i>anthocyanin synthesis</i>	No
AT4G39980	DHS1	chorismate, pathogen defense, wounding	Yes
AT4G31870	glutathione peroxidase	<i>oxidative stress</i>	No
AT4G27560	UDP-glucosyl transferase	<i>cytokinins</i>	No
AT4G23600	CORI3	<i>ABA, salt, jasmoate, toxin, wounding</i>	No
AT4G04840	methionine sulfoxide reductase	unknown	Yes
AT3G29200	chorismate mutase (CM1)	amino acid synthesis, <i>peroxidase</i>	No
AT3G19450	CAD4	<i>lignin biosynthesis</i>	No
AT3G22840	chlorophyll A-B binding protein	<i>cold</i>	No

Table A.1 (Continued)

Identities and functions of the genes identified to be shifting to a higher state in the wildtype after 90 minutes of light stress. The last column indicates whether the gene responded to light stress in the mutant. Highlighted rows indicate those involved in stress. Genes with unknown function are omitted to save space. The genes listed here also passed a visual filter in which they were plotted individually in order to verify that their pattern was significant. The information was derived from AtGeneSearch [Wurtele et al., 2003].

Locus Name	Identity	Function/Pathways	KO-APX
<i>AT3G22370</i>	<i>AOX1A</i>	<i>cold</i>	<i>No</i>
AT3G09650	HCF152	chloroplast mRNA processing	<i>No</i>
AT1G76790	O-methyltransferase	methyl transferase	<i>No</i>
AT1G06000	UDP-glucosyl transferase	<i>cytokinins</i>	Yes
AT1G65060	4CL3	<i>flavonoid, lignin synthesis</i>	Yes
AT1G06430	FtsH protease	chloroplast protease	<i>No</i>
AT1G28610	GDSL-motif lipase	TAG degradation	<i>No</i>
AT1G78580	ATTPS1	sugar metabolism	Yes
AT1G78580	phenylalanine ammonia-lyase	<i>flavonoid synth, wounding, ROS, defense</i>	Yes
AT1G22770	gigantea protein	<i>cold</i>	<i>No</i>
AT1G61800	glucose-6-phosphate translocator	sugar transport	<i>No</i>
AT1G10370	glutathione S-transferase	<i>toxins</i>	<i>No</i>
AT1G65560	allyl alcohol dehydrogenase	oxidoreductase	Yes
AT1G78580	multi-copper oxidase	unknown	Yes
AT2G36750	UDP-glucosyl transferase	<i>cytokinins</i>	<i>No</i>
AT2G02000	glutamate decarboxylase	glutamate to succinate	<i>No</i>
AT1G78580	zinc finger protein	transcription factor	<i>No</i>
AT2G41040	methyltransferase-related	carbon monoxide dehydrogenase	<i>No</i>
AT2G23000	serine carboxypeptidase	proteolysis	<i>No</i>
AT1G78580	cinnamate-4-hydroxylase	<i>flavonoids, wounding, light</i>	<i>No</i>



**Figure A.8** The distribution of biological functions for the genes found to be reaching a higher state at 90 minutes in the wildtype. The flavonoid genes seem to be directed towards anthocyanin synthesis. The sum of the flavonoid synthesis and other stress genes is greater than the “other” category. The more darkly shaded regions indicate the number of genes lost when APX1 is knocked out.

### Exploring the differences between wildtype and KO-APX

The focus of the analysis now shifts to see what, if any, are the effects of knocking out the APX gene. The same linear model described above is fit to the KO-APX levels. Figure 9 shows the scatterplots for the three time coefficients vs. their respective p-values for the mutant fit, with the gene colors the same as in the wildtype plots. Many of the genes that were significant (colored) in wildtype are no longer so in the mutant. Also, there are many gray genes that now appear significant. Interestingly, the red and orange colored genes that lost significance at higher time orders in wildtype now have a similar significance pattern relative to the green and brown genes. The parallel coordinate plot in Figure 10 results from coloring the gray genes on the “brown side” pink in the scatterplot. They appear to increase much earlier than the

majority of those in wildtype and fall off earlier as well. The *t*-statistic analysis was performed for the pink genes, yielding the barcharts shown in Figure 11.

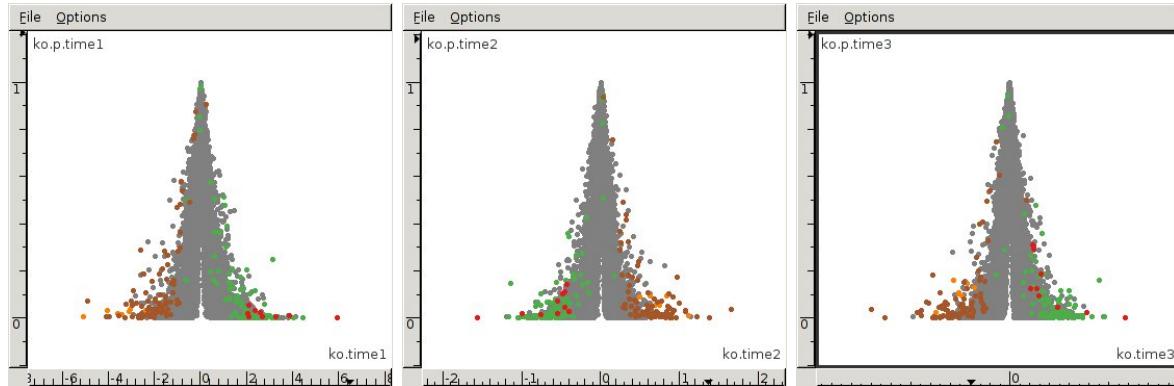


Figure A.9 Scatterplots of the time coefficients vs. their p-values for each gene. The colors are the same as in wildtype. Note that these plots bear only a slight resemblance to those for wildtype.

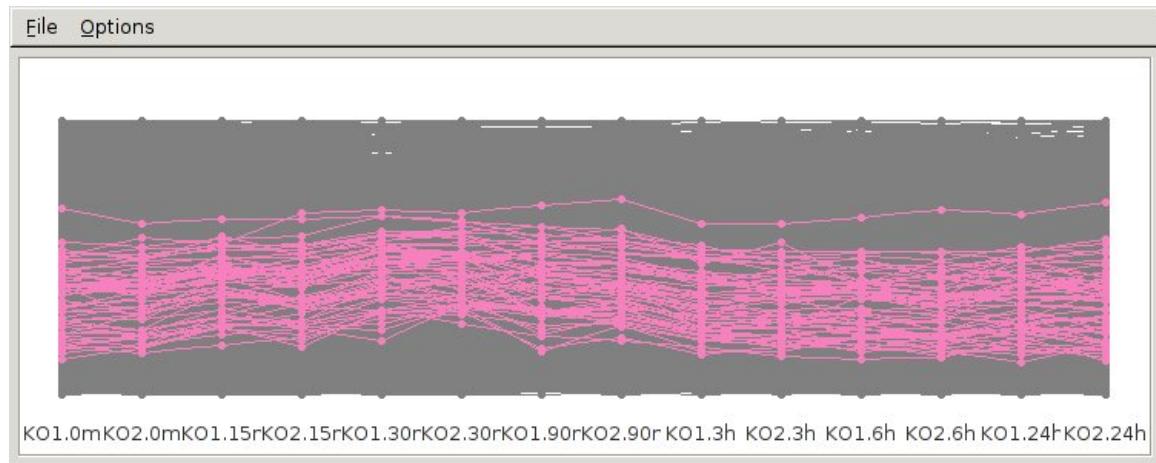


Figure A.10 Parallel coordinate plot of genes that were not significant with respect to time in the wildtype. There appears to be a slight increase around 15 or 30 minutes.

The identities and functions of the pink genes are listed in Table 2, with their functional distribution plotted in Figure 12. The prevalence of genes involved in the abscisic and jasmonic acid pathways, both directly related to stress response, is immediately recognizable. The quick jump in ROS levels resulting from the loss of APX1 may explain the immediate activation

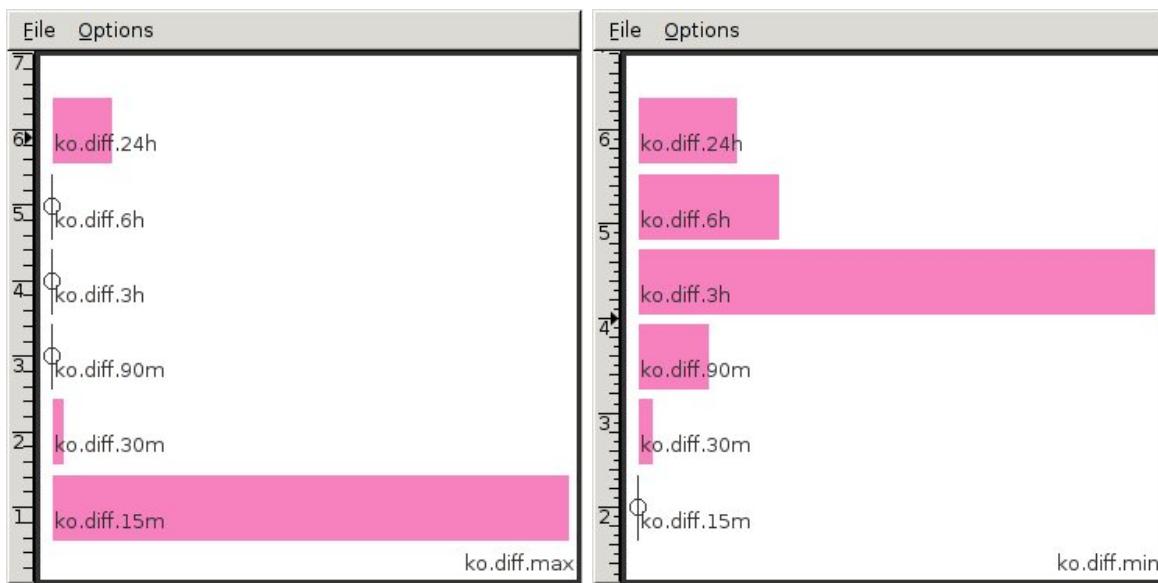


Figure A.11 Barcharts for the maxima and minima of the set of *t*-statistics calculated for each gene in the mutant. It appears that the great majority of the genes rise at 15 minutes and fall at 3 hours.

of these pathways, which might compensate for the lack of response by many genes that were activated in the wildtype (see Figure 8 and Table 1). The ABA and H<sub>2</sub>O<sub>2</sub> pathways have been shown to regulate many of the same targets [Wang et al., 2006b]. Interestingly, while most stress and cytokinin related genes responding in the wildtype do not respond in the mutant, most of those involved in flavonoid and anthocyanin biosynthesis still manage to become activated. Furthermore, two of the three anthocyanin genes that are non-responsive in the knockout (chalcone-flavonone isomerase and phenylalanine lyase) are redundant: two isomers of each were activated in wildtype. This suggests that there is an alternative pathway for anthocyanin response to light stress, perhaps independent of cytokinins. Another interesting observation is activation of many calcium-dependent proteins that are not associated with a specific pathway. These may be related to the ABA response, since its stomata closing mechanism and other effects are thought to depend on calcium transfer from the vacuole to the cytosol [Foreman et al., 2003]. Also of note is the apparent activation of the NADPH

oxidase RbohD, which has been suggested to amplify the ROS signal in response to stress [Davletova et al., 2005]. Perhaps it is up-regulated to compensate for the loss of H<sub>2</sub>O<sub>2</sub> signal transmission by APX1.

Table A.2 Genes responding in the mutant but not in wildtype. Specifically, these are the genes increasing at 15 minutes and decreasing at 3 hours. The genes were plotted individually to filter out those that did not fit the pattern as indicated by the *t*-statistics. Highlighted rows indicate those genes thought to be involved in stress. Unknowns are not included for brevity. The information was derived from AtGeneSearch [Wurtele et al., 2003].

Locus Name	Identity	Function/Pathways
<i>AT1G56600</i>	galactinol synthase	carbohydrate synthesis, heat stress
<i>AT5G04340</i>	c2h2 zinc finger protein	transcription factor
<i>AT5G27420</i>	RING-H2 zinc finger protein	<i>TF, ABA response</i>
<i>AT5G66210</i>	CPK28	<i>Ca-dependent ser/thr kinase</i>
<i>AT5G54490</i>	PBP1, <i>EF-hand</i> protein	auxin response
<i>AT5G54130</i>	<i>EF-hand</i> protein	unknown
<i>AT5G47910</i>	RbohD	<i>NADPH oxidase</i>
<i>AT5G39670</i>	<i>EF-hand</i> protein	unknown
<i>AT5G22380</i>	No Apical Meristem (NAM)	regulates development
<i>AT5G18470</i>	mannose-binding lectin protein	unknown
<i>AT5G13200</i>	<i>ABA responsive protein</i>	unknown
<i>AT5G01540</i>	lectin protein kinase	<i>ABA response</i>
<i>AT3G61890</i>	ATHB-12	<i>ABA response, osmotic stress</i>
<i>AT3G57530</i>	CPK32	<i>Ca-dependent, ABA, salt stress</i>
<i>AT3G50930</i>	AAA-type ATPase	ATP binding
<i>AT3G49530</i>	no apical meristem	regulation of development
<i>AT3G47420</i>	G3P transporter	sugar transporter
<i>AT3G23920</i>	beta-amylase	starch synthesis
<i>AT3G14440</i>	NCED3	<i>ABA synthesis</i>
<i>AT3G25780</i>	AOC2	<i>Jasmonic acid synthesis</i>

Table A.2 (Continued)

Genes responding in the mutant but not in wildtype. Specifically, these are the genes increasing at 15 minutes and decreasing at 3 hours. The genes were plotted individually to filter out those that did not fit the pattern as indicated by the *t*-statistics. Highlighted rows indicate those genes thought to be involved in stress. Unknowns are not included for brevity. The information was derived from AtGeneSearch [Wurtele et al., 2003].

Locus Name	Identity	Function/Pathways
<i>AT3G17800</i>	ATACP5	Ser/Thr kinase, <i>UV response</i>
AT3G01830	calmodulin-related protein	Ca binding
<i>AT3G11410</i>	protein phosphatase 2C	<i>ABA response</i>
AT1G01520	myb family transcription factor	transcription factor
AT1G77680	ribonuclease II family protein	ribonuclease
AT1G76650	<i>EF-hand</i> protein	unknown
<i>AT1G72520</i>	lipoxygenase	<i>defense, wounding, jasmonic acid</i>
AT1G53170	ATERF-8	ethylene response
AT1G28370	ERF	ethylene response
<i>AT1G01720</i>	ATAF1	<i>wounding</i>
<i>AT1G32640</i>	ATMYC2	<i>ABA, jasmonic acid, wounding</i>
AT1G80840	WRKY40	transcription factor
<i>AT1G72900</i>	disease resistance protein	<i>defense</i>
<i>AT1G72940</i>	disease resistance protein	<i>defense</i>
<i>AT1G49450</i>	transducin family protein	nucleotide binding
AT1G15100	RHA2A	RING-H2 zinc finger
<i>AT2G40140</i>	Zinc finger protein	<i>response to cold</i>
AT1G24140	matrixin family protein	proteolysis
AT2G20880	similar to RAP2.4	transcription factor

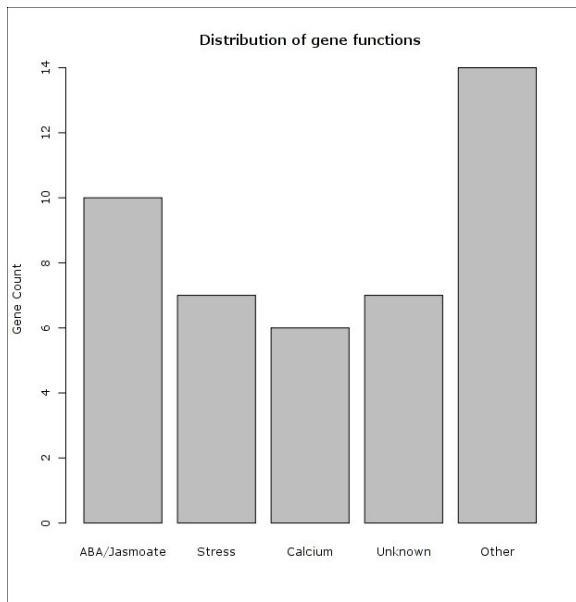


Figure A.12 Barchart of the biological function distribution of the genes listed in Table 2. Genes involved in ABA and jasmonic acid response were grouped together separately from the other stress response genes, which were placed in the “Stress” category. The “calcium” category contains genes that are dependent in some way on calcium.

## Discussion

### ROS and Anthocyanin synthesis

The high degree of anthocyanin activation in the wildtype is expected given that the plants are being exposed to light stress, which is directly counteracted by the anti-oxidant properties of anthocyanins. It may seem strange that the anthocyanins remain up-regulated in KO-APX despite the fact that every other stress related gene activated at 90 minutes in the wildtype does not respond in the mutant. However, there are plausible explanations for this. First, ROS signaling occurs in stresses besides excess light, but one would not expect anthocyanin synthesis to be activated to the same degree during other stresses. There is likely a light-dependent regulatory mechanism involved [Kubasek et al., 1992]. Another possibility is that the ABA

and jasmonic acid pathways are somehow compensating for the breakdown in H<sub>2</sub>O<sub>2</sub> signaling and activating anthocyanin synthesis, but one is still left with explaining how the activation occurs specifically in response to light. Thus, we favor the first suggestion. It is important to remember that more anthocyanin synthesis genes are activated when APX1 is functional, so anthocyanin production may be enhanced by but not dependent on ROS signaling. H<sub>2</sub>O<sub>2</sub> has been shown to stimulate anthocyanin synthesis [Vanderauwera et al., 2005].

### The H<sub>2</sub>O<sub>2</sub> signaling pathway

Davletova et al. [2005] proposed a model in which Heat Shock Factor 21 (HSF21) acts as an H<sub>2</sub>O<sub>2</sub> sensor, perhaps through its oxidation, and stimulates Zat12 transcription, which in turn activates the transcription of APX1. This signal may then be amplified by the NADPH oxidase RbohD. The response of HSF21 and Zat12 to light stress did not appear as significant as that of the genes noted in this study. It is quite possible that HSF21 and Zat12 are factors in the ROS signaling pathway, but it is difficult to make such a conclusion from the microarray data alone. We did, however, observe a very slight response from RbohD in KO-APX, which agrees with the findings of Davletova et al.. It may be that ABA is activating RbohD in an attempt to compensate for the loss of the H<sub>2</sub>O<sub>2</sub> signal from APX1.

### Conclusion

It should be noted that the results presented here are somewhat limited in scope. We have yet to investigate the genes up-regulated at time points other than 90 minutes in the wildtype and 15 minutes in the mutant. We have also completely ignored any down-regulation. Although many of the same genes respond in the two genotypes we have not investigated whether the expression profiles are similar. These are promising avenues for future work and might lead to further understanding of the H<sub>2</sub>O<sub>2</sub> signaling pathway and stress response as a whole in *Arabidopsis*.

## Cytosolic acetyl-CoA metabolism in *Arabidopsis*

Authors: Heather Babka, Michael Lawrence, Dianne Cook, Eve Wurtele

### Introduction

Acetyl-CoA is a central component of plant metabolism. Acetyl-CoA cannot cross membranes, thus it must be synthesized within the cell compartment in which it will be used. The cytosolic pool of acetyl-CoA is synthesized by the ATP dependent cleavage of citrate by the enzyme ATP-citrate lyase (ACL) .

Cytosolic acetyl-CoA can be metabolized via carboxylation, condensation, or acetylation. The products of carboxylation include elongated fatty acids, flavonoids, and malonyl-CoA derivatives. Condensation of cytosolic acetyl-CoA leads to mevalonate-derived isoprenoids like sterols and brassinosteroids. Acetylation can lead to the production isoprenoids, anthocyanins, and sugars.

*Arabidopsis* plants with reduced ACL activity were generated in order to better understand the importance of ACL generated acetyl-CoA. The resultant plants have an altered phenotype and have shortages of chemicals that are synthesized via the cytosolic pool of acetyl-CoA. ACL-deficient plants are rescued by treatment with exogenous malonate. We suspect this is due to malonate feeding the carboxylation pathway of cytosolic acetyl-CoA metabolism.

In order to better understand the consequences of a limited cytosolic acetyl-CoA pool and the subsequent rescue of these alterations with malonate, transcriptomic and metabolomic analyses were performed. Exploratory data analysis was utilized to analyze the transcriptomic data.

Abbreviations used: ACL, plants with decreased ACL activity; ACL-H<sub>2</sub>O, ACL plants treated with water; ACL-MA, ACL plants treated with malonate; WT-H<sub>2</sub>O, wildtype plants treated with water; WT-MA, wildtype plants treated with malonate;

## Methods

The microarray data was obtained from the Arabidopsis Genome ATH1 Array from Affymetrix. RMA normalization was preformed in R using the RMA package from Bioconductor. Exploratory analysis of the preprocessed data was carried out in exploRase. The scatter plot matrices of the biological replicates were compared using exploRase to check the data quality. Limma [Smyth, 2005] analysis was preformed through the exploRase limma frontend. The output consisted of F-statistics and p-values for the genotype, treatment, genotype\*treatment effects. The table of entities in the exploRase GUI was sorted and filtered based on these numbers.

For more in-depth analysis, the median for each genotype by treatment combination was calculated with exploRase and the gene profiles across the medians were shown in a parallel coordinate plot. Then, using the pattern finder, interesting patterns were detected. Each pattern of interest was highlighted in the parallel coordinate plot to check the significance of the pattern. When an interesting pattern was observed, similar entities were found via correlation of their patterns.

## Results

Initial analysis of the transcriptomic data showed few significantly differentially expressed genes in ACL water treated plants versus ACL malonate treated plants. This statistical observation did not mesh with the phenotypic observations of these plants. In order to better understand the transcription changes in ACL-H<sub>2</sub>O plants versus ACL-MA plants we resorted to exploratory data analysis. Using the methods described above, a long list of genes was obtained.

We discovered one particularly interesting gene, AT3G02310 (SEP2), which is more highly expressed in the ACL-H<sub>2</sub>O plants relative to the others, where its expression is approximately the same (Figure A.13). Also shown in Figure A.13 are genes that have a similar expression pattern.

exploRase also facilitated checking the transcript levels of two ACL genes (Figure A.14)

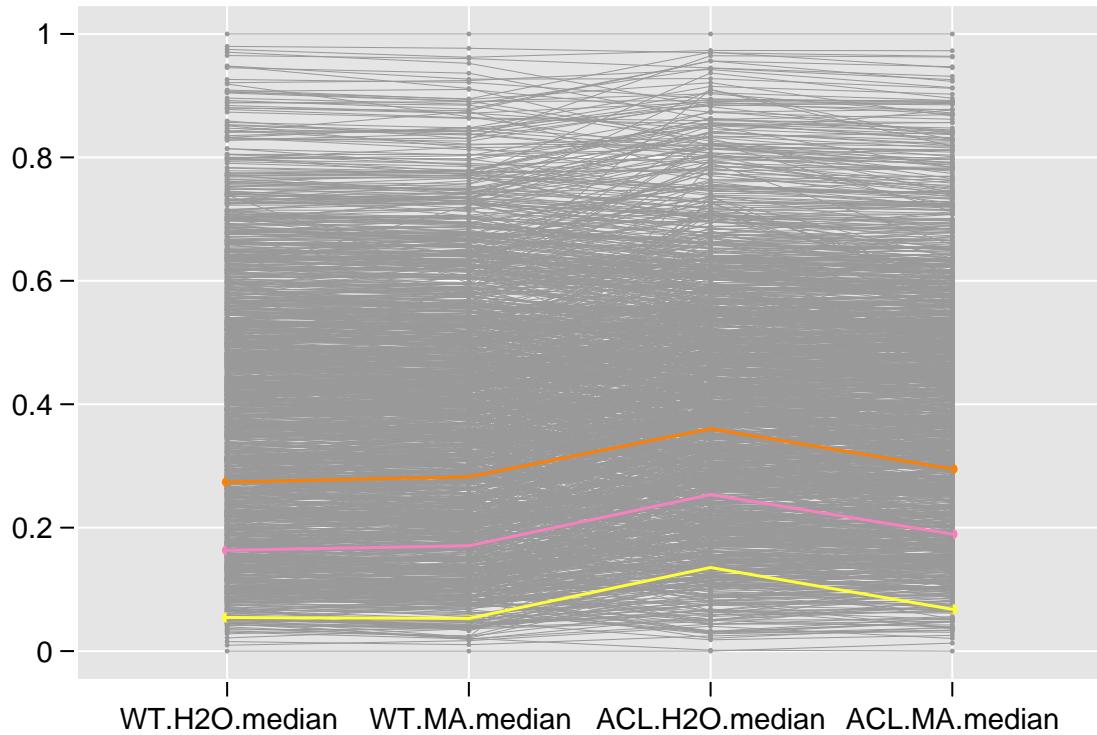


Figure A.13 Plot of genes correlated to AT3G02310 (SEP2) shown in yellow. These transcripts return to more normal levels in ACL plants treated with malonate. In orange is AT1G69600 (a ZF-HD homeobox family protein) and in pink is AT1G22170 (phosphoglycerate/bisphosphoglycerate mutase family protein).

and starch synthesis genes (Figure A.15). As expected, the transcript level of ACL in the ACL plants is lower than in wildtype (Figure A.14). The ACL plants hyper-accumulate starch and, as shown in Figure A.15, several starch synthesis genes are up-regulated.

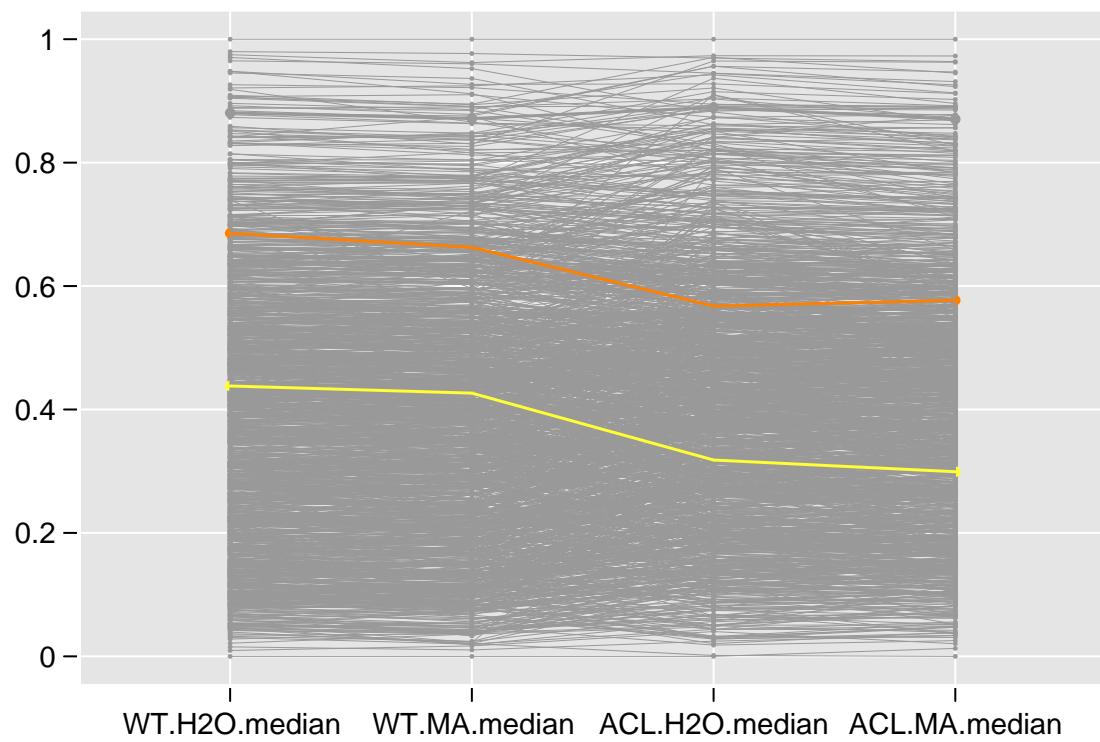


Figure A.14 ACLA-2 (AT1G60810) is plotted in yellow while ACLA-3 (AT1G09430) is shown in orange. This shows that the gene silencing worked.

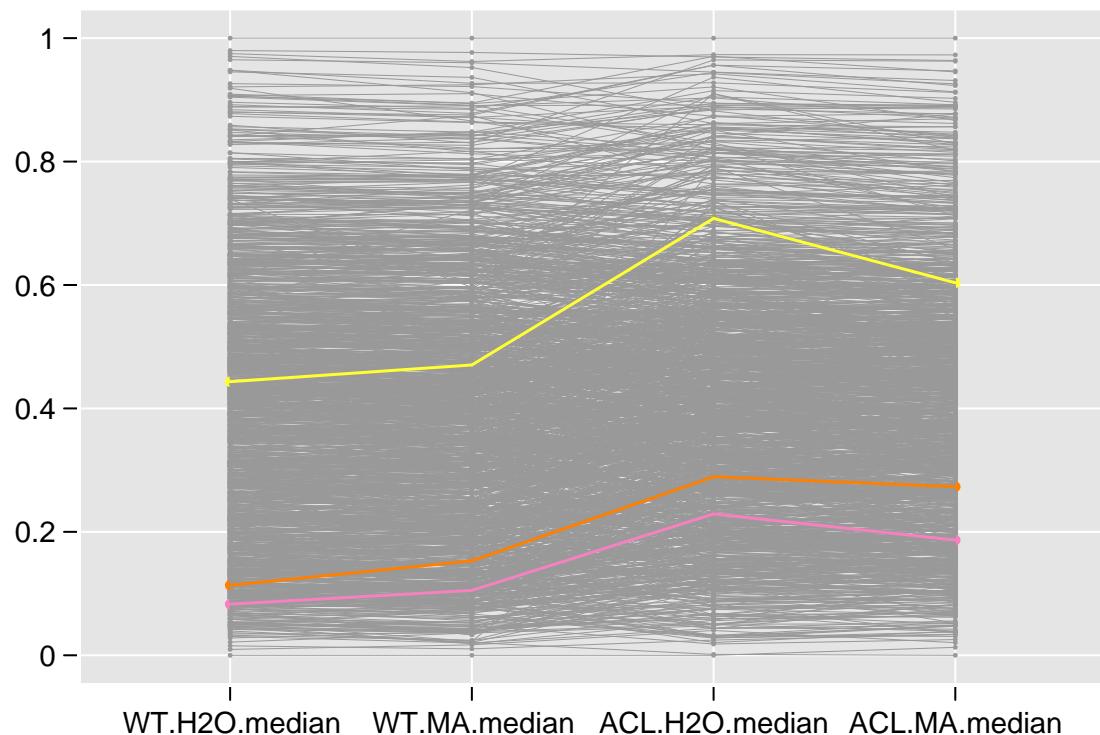


Figure A.15 This plot shows the coordinate expression of three genes related to starch biosynthesis, which up-regulated in ACL plants. AT1G32900 (AtGBSS1) is yellow, AT4G39210 (APL3) is orange, and AT2G21590 (APL2) is pink.

## Metabolomic and transcriptomic data analysis of PHB-producing Arabidopsis using R, exploRase and GGobi

Authors: Suh-Yeon Choi, Michael Lawrence, Dianne Cook, Heike Hofmann, Lauralynn Kourtz, Kristi Snell, Basil J. Nikolau and Eve Syrkin Wurtele

Plants can serve as solar-energy-powered biochemical factories. Rapid development in molecular biology during the past decades allows us to manipulate plant genes to produce what we need. Polyhydroxyalkanoates (PHAs) are a class of biodegradable polyesters (bioplastics) that could replace petroleum-based polymers. PHAs are naturally produced by many bacteria, and can also be produced in transgenic plants by genetic engineering. There have been efforts to produce this valuable product in plants for more than 10 years, since plant-based polymers are produced by solar rather than using petroleum-based energy; they are biodegradable; they utilize atmospheric carbon (carbon dioxide) rather than carbon from petroleum; and they do not require toxic chemicals in their synthesis as petroleum-derived plastic do [Poirier et al., 1992]. In sufficient yield, PHAs can economically compete with petroleum-based polymers. However, the introduction of the PHA-producing genes into plants can have detrimental effects on growth, thus reducing the economic viability of plants as sources of these biopolymers. To develop a high PHA-yielding plant, it is critical to systematically study the metabolic effects of addition of these new genes.

Polyhydroxybutyrate, a PHA, is produced from acetyl-CoA by the sequential action of three enzymes: 3-ketothiolase (*phaA*), acetoacetyl-CoA reductase (*phaB*), and PHA synthase (*phaC*). These three enzymes have been targeted to plastids of *Arabidopsis* using an inducible promoter system to minimize detrimental growth effects [Kourtz et al., 2005]. Two inducible lines of transgenic plant (IND7, IND11), constitutive expression plants (UBIQ), and control plants (CONT) were randomly assigned within the same flats. The plants were grown under three different illumination cycle; long day (16hr light, 8hr dark; LD), short day (8hr light, 16hr dark; SD) and constant illumination (24hr light; CI). Rosette leaf tissues were collected for PHB quantification, metabolomics analysis, and transcriptomic analysis.

PHB accumulation was dependent on the light regime, and was highest under LD. The

	Cont	UBIQ	IND7	IND11
Short day	4	4	6	3
Long day	4	4	10	10
Constant illum.	4	2(1)	5	2(1)

Table A.3 Number of replicates for each biological sample analyzed by metabolite profiling. Samples from constant illumination were small and thus resulted in fewer replicates.

mean PHB accumulation was up to 4-fold higher when plants were grown under long day light conditions compare to short day or constant illumination. The maximum accumulation of PHB was 18.7% (of dry weight), and detected in inducible plant grown under long day (LD) condition. Maximum PHB accumulation under short day condition and constant illumination were 2.32% and 6%, respectively.

Metabolite profiles were detected using non-targeted GC-MS analysis and the data sets were pre-processed using AMDIS. correction, peak detection, deconvolution, and peak matching across the samples. The chromatoplots software is an R package developed for GC-MS metabolomics preprocessing in conjunction with diagnostic visualizations. The result from chromatoplots is a data matrix which can be loaded into exploRase for further analysis. Using exploRase, metabolites that accumulate depending on the PHB-accumulation level in PHB-producing plants were found and recorded. To find metabolites altered by PHB production, we have calculated the correlation between the PHB level and each metabolite level. The student t-test was not suitable for detecting such correlations, because the variation of PHB production in each plant was large. So we decided to identify the metabolites with expression patterns that were highly correlated with PHB.

Three control plants and three inducible plants grown under the long day condition were also used for Affymetrix genechip analysis. Total RNA was extracted and analyzed using Arabidopsis Genome ATH1 Array from Affymetrix. The RMA (Robust Multichip Average) preprocessing algorithm, as implemented in the affy Bioconductor package, was used to calculate expression levels. For assessing differential expression, the data was analyzed in exploRase

Peak No.	Candidate ID	Correlation
p116	myo-Inositol	-.64
p62	(xylose)	-.62
p139	(D-turanose)	-.59
p148	maltose	-.58
p144	sucrose	-.55
p61	Glulonic acid, gamma lactone	-.52
p17	fumarate	-.50

Table A.4 List of metabolites negatively correlated with PHB in plants grown under long day condition.

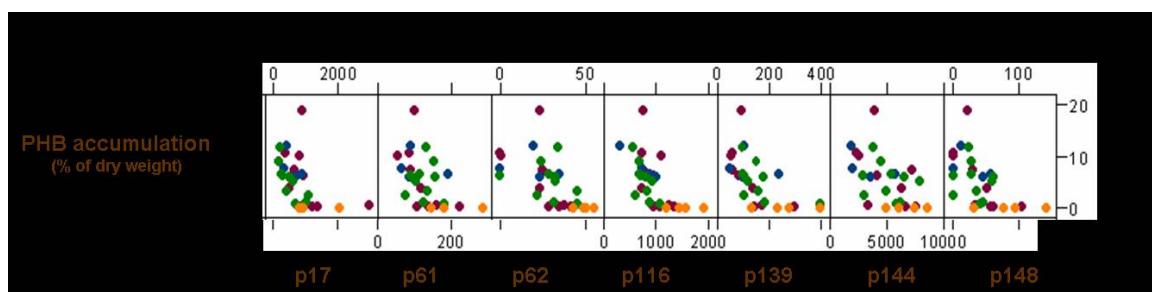


Figure A.16 Plots of relative intensity versus PHB accumulation for each of the compounds listed in Table A.4.

[Lawrence et al., 2006, 2007a]. The limma [Smyth, 2005] package, through its exploRase front-end, was used to find differentially expressed genes.

The list of differentially accumulated genes and metabolites in PHB-producing plants were exported from exploRase and visualized in the context of MetNetDB [Wurtele et al., 2003] Arabidopsis biological networks with Cytoscape to identify the connections and relations among gene products and metabolites.

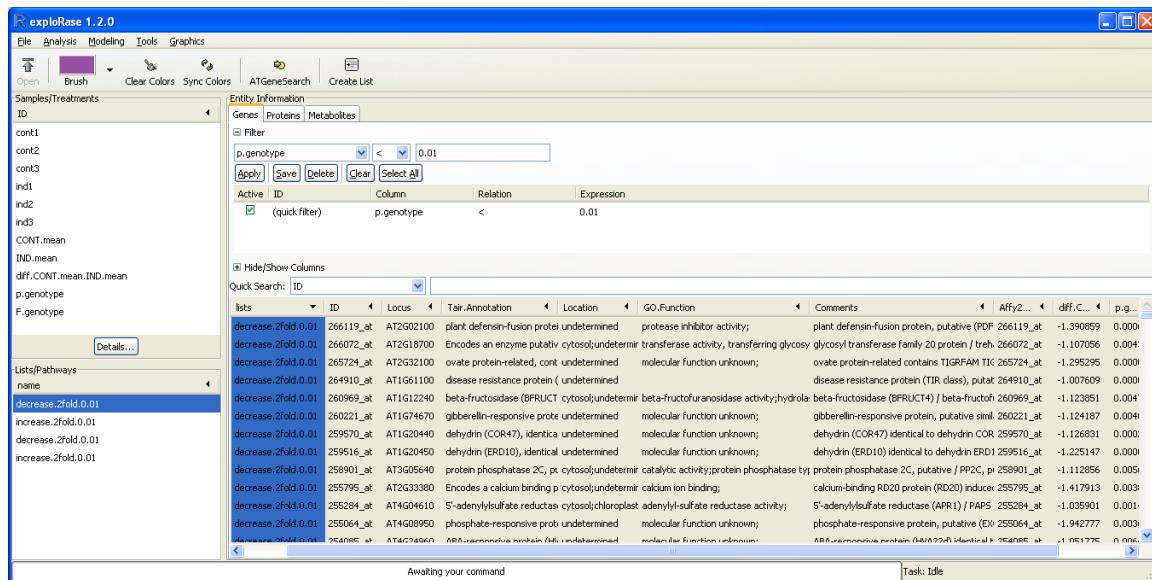


Figure A.17 Analysis of microarray data using exploRase. exploRase main window showing differentially expressed genes ( $p < 0.01$ ). genes decreased more than 2 fold blushed with blue color; genes increased more than 2 fold blushed with red.

It is hoped that the insights gained from analyzing this experiment with exploRase will lead to a better understanding of how Arabidopsis responds to the production of PHB. This will guide the optimization of bioplastic production in plants.

metabolomic and transcriptomic analysis were able to be analyzed with user's inspection. The understanding from this study will help to understand how plant metabolism would response to the production of foreign materials in plants, and these understanding will guide to optimization of production of bioplastic in plants.

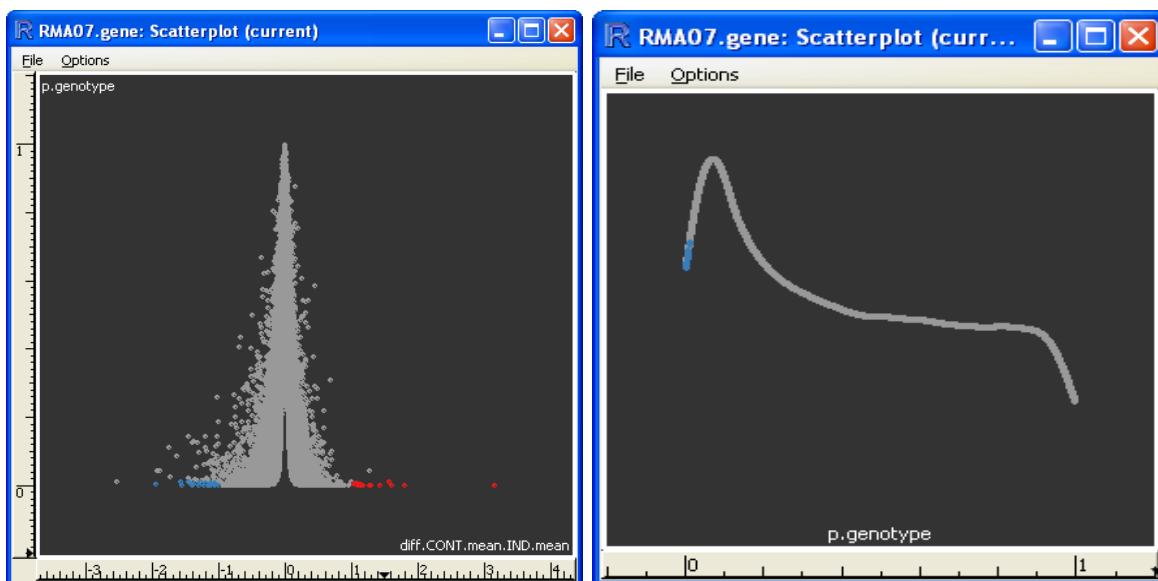


Figure A.18 Screenshots of exploRase plots of p-value vs. fold difference (left) and distribution of p-values (right) for the genotype contrast fit by limma.

## BIBLIOGRAPHY

- V.P. Andreev, T. Rejtar, H.S. Chen, E.V. Moskovets, A.R. Ivanov, and B.L. Karger. A universal denoising and peak picking algorithm for LC-MS based on matched filtration in the chromatographic time domain. *Anal Chem*, 75(22):6314–26, 2003.
- Felix Andrews. The playwith package, 2007. URL <http://cran.r-project.org/src/contrib/Descriptions/playwith.html>.
- Apache Software Foundation. Package org.apache.commons.pipeline, 2007. URL <http://jakarta.apache.org/commons/sandbox/pipeline/apidocs/org/apache/commons/pipeline/package-summary.html>.
- K. Apel and H. Hirt. Reactive oxygen species: Metabolism, oxidative stress, and signal transduction. *Annu. Rev. Plant Biol.*, 55:373–379, 2004.
- Mark Ardis, Kenneth Cox, Stacie Hibino, Lichan Hong, Audris Mockus, and Graham Wills. Building information visualizations: A commonality analysis. In *Information Visualization 2000*, 2000.
- T. Baier and E. Neuwirth. R (D)COM Server and RExcel packages, 2007. URL <http://sunsite.univie.ac.at/rcom/>.
- R. Baran, H. Kochi, N. Saito, M. Suematsu, T. Soga, T. Nishioka, M. Robert, and M. Tomita. MathDAMP: a package for differential analysis of metabolite profiles. *BMC Bioinformatics*, 7(1):530, 2006.
- M.Y. Becker and I. Rojas. A graph layout algorithm for drawing metabolic pathways. *Bioinformatics*, 17(5):461–467, 2001.

- R. A. Becker and W. S. Cleveland. Brushing Scatterplots. *Technometrics*, 29(2):127–142, 1987.
- M. Bellew, M. Coram, M. Fitzgibbon, M. Igra, T. Randolph, P. Wang, D. May, J. Eng, R. Fang, C.W. Lin, et al. A suite of algorithms for the comprehensive analysis of complex protein mixtures using high-resolution LC-MS. *Bioinformatics*, 22(15):1902, 2006.
- M. Bernstein Niel. Using the Gtk toolkit with Mono, 2004. URL [http://www.ondotnet.com/pub/a/dotnet/2004/08/09/gtk\\_mono.htm](http://www.ondotnet.com/pub/a/dotnet/2004/08/09/gtk_mono.htm).
- Jurg Billeter. Vala - Compiler for the GObject Type System, 2007. URL <http://live.gnome.org/Vala>.
- S. Bocker, M. Letzel, Z. Liptak, and A. Pervukhin. Decomposing metabolomic isotope patterns. In *Proceedings of the 6th Workshop on Algorithms in Bioinformatics WABI*, volume 3, 2006.
- K.F. Böhringer and F.N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people*, pages 43–51. ACM Press New York, NY, USA, 1990.
- Ben Bornstein. libsbml, 2006. URL <http://sbml.org/software/libsbml/>.
- S. Bridgeman and R. Tamassia. A User Study in Similarity Measures for Graph Drawing. *Journal of Graph Algorithms and Applications*, 6(3):225–254, 2002.
- C.D. Broeckling, I.R. Reddy, A.L. Duran, X. Zhao, and L.W. Sumner. MET-IDEA: data extraction tool for mass spectrometry-based metabolomics. *Anal Chem*, 78(13):4334–4341, 2006.
- A. Buja, D. Asimov, C. Hurley, and J. A. McDonald. Elements of a Viewing Pipeline for Data Analysis. In W. S. Cleveland and M. E. McGill, editors, *Dynamic Graphics for Statistics*, pages 277–308. Wadsworth, Monterey, CA, 1988a.
- Andreas Buja, Catherine Hurley, and John Alan McDonald. A data viewer for multivariate data. In *Computing Science and Statistics: Proceedings of the 18th Symposium on the Interface*, pages 171–174, Washington, DC, 1986. American Statistical Association.

Andreas Buja, Daniel Asimov, Catherine Hurley, and John A. McDonald. Elements of a viewing pipeline for data analysis. In *Dynamic Graphics for Statistics*. Wadsworth, Inc., 1988b.

Cairo. Cairo vector graphics library, 2007. URL <http://www.cairoglyphics.org>.

M. Chapman and B. Kelley. Examining the PyGtk Toolkit. *Dr. Dobb's Journal of Software Tools*, 25(4):82, 2000.

J.H. Christensen, J. Mortensen, O. Andersen, and A. Hansen. Chromatographic preprocessing of GC-MS data for analysis of complex chemical mixtures. *Journal of Chromatography A*, 1062(1):113–123, 2005.

D. Cook, H. Hofmann, E.-K. Lee, H. Yang, B. Nikolau, and E. Wurtele. Exploring Gene Expression Data, Using Plots. *Journal of Data Science*, 5:151–182, 2007.

Peter Dalgaard. A primer on the R-Tcl/Tk package. *R News*, 1(3):27–31, September 2001. URL <http://CRAN.R-project.org/doc/Rnews/>.

Peter Dalgaard. Changes to the R-Tcl/Tk package. *R News*, 2(3):25–27, December 2002. URL <http://CRAN.R-project.org/doc/Rnews/>.

R. Danielsson, D. Bylund, and KE Markides. Matched filtering with background suppression for improved quality of base peak chromatograms and mass spectra in liquid chromatography-mass spectrometry. *Analytica Chimica Acta*, 454(2):167–184, 2002.

S. Davletova, L. Rizhsky, H. Liang, Z. Shengqiang, D. Oliver, J. Coutu, V. Shulaev, K. Schlauch, and R. Mittler. Cytosolic ascorbate peroxidase 1 is a central component of the reactive oxygen gene network of Arabidopsis. *Plant Cell*, 17:268–281, 2005.

J. Deikman and P. E. Hammer. Induction of anthocyanin accumulation by cytokinins in Arabidopsis thaliana. *Plant Physiology*, 108:47–57, 1995.

V.B. Di Marco and G.G. Bombi. Mathematical functions for the representation of chromatographic peaks. *J. Chromatogr. A*, 931(1), 2001.

- Hugo A. D. do Nascimento. A framework for human-computer interaction in directed graph drawing. In Peter Eades and Tim Pattison, editors, *Conferences in Research and Practice in Information Technology*, volume 9, pages 63–69, 2001. URL <http://www.cs.usyd.edu.au/~hadn/thesis/gdhints/fullversion/>.
- Dogrusoz, Giral, Cetintas, Civril, and Demir. A compound graph layout algorithm for biological pathways. *Lecture Notes in Computer Science*, 3383:442–447, 2004.
- Punit R. Doshi, Elke A. Rundensteiner, Matthew O. Ward, and Daniel Stroe. Prefetching for visual data exploration. Technical Report WPI-CS-TR-02-07, Worcester Polytechnic Institute, 2007. URL <http://davis.wpi.edu/~xmvdv/docs/tr2002-Punit.pdf>.
- L. Drake, Plummer M., and Temple Lang D. gtkDevice: Loadable and embeddable gtk device driver for R, 2005. URL <http://cran.r-project.org/src/contrib/Descriptions/gtkDevice.html>.
- RG Dromey, M.J. Stefk, T.C. Rindfleisch, and A.M. Duffield. Extraction of mass spectra free of background and neighboring component contributions from gas chromatography/mass spectrometry data. *Analytical Chemistry*, 48(9):1368–1375, 1976.
- Pan Du, Warren A. Kibbe, and Simon M. Lin. Improved peak detection in mass spectrum by incorporating continuous wavelet transform-based pattern matching. *Bioinformatics*, 22(17):2059–2065, 2006.
- Tim Dwyer. adaptagrams library, 2007. URL <http://adaptagrams.sourceforge.net>.
- Tim Dwyer and Kim Marriott. IPsep-CoLa: An incremental procedure for separation constraint layout of graphs. *IEEE Trans Vis Comput Graph*, 12(5):821–828, 2006.
- Tim Dwyer, Yehuda Koren, and Kim Marriott. Drawing directed graphs using quadratic programming. *IEEE transactions on visualization and computer graphics*, 12(4):536–548, Jul/Aug 2006.
- P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

- P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics*, number 9 in 91, pages 24–33, 1991.
- P.H.C. Eilers. Parametric time warping. *Analytical Chemistry*, 76(2):404–411, 2004.
- M. A. Fisherkeller, J. H. Friedman, and J. W. Tukey. Prim-s, an interactive multidimensional data display and analysis system. In *Dynamic Graphics for Statistics*, pages 91–109. Wadsworth, 1975.
- J. Foreman, V. Demidchik, J.H. Bothwell, P. Mylona, H. Miedema, M.A. Torres, P. Linstead, S. Costa, C. Brownlee, and J.D. Jones. Reactive oxygen species produced by NADPH oxidase regulate plant cell growth. *Nature*, 422:442–446, 2003.
- Olivier Fourdan. Xfce : A lightweight desktop environment. In *Proceedings of the 4th Annual Linux Showcase, Atlanta*, 2000.
- J. Fox. Rcmdr: R commander, 2007. URL <http://cran.r-project.org/src/contrib/Descriptions/Rcmdr.html>.
- C.H. Foyer and G. Noctor. Redox sensing and signalling associated with reactive oxygen in chloroplasts, peroxisomes and mitochondria. *Physiologia Plantarum*, 119:355–364, 2003.
- A. Frick, A. Ludwig, and H. Mehldau. A Fast Adaptive Layout Algorithm for Undirected Graphs. In *Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 388–403. Springer-Verlag London, UK, 1994.
- Jeffrey Friedl. *Mastering Regular Expressions*. O'Reilly, 3 edition, 2006.
- T.M.J. Fruchterman and E.M. Reingold. Graph Drawing by Force-directed Placement. *Software - Practice and Experience*, 21(11):1129–1164, 1991.
- Gabouje and Zimanyi. A new compound graph layout algorithm for visualizing biochemical networks. In *Poster Proceedings Volume of the 4th International Workshop on Efficient and Experimental Algorithms*, Santorini Island, May 2005. URL <http://cs.ulb.ac.be/publications/P-05-05.pdf>.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

E. Gansner, Y. Koren, and S. North. Graph Drawing by Stress Majorization. *Lecture Notes in Computer Science, Proc. of 12th Int. Symp. Graph Drawing (GD'04)*, 3383:239–250, 2005.

Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

Ralph Gauges, Ursula Rost, Sven Sahle, and Katja Wegner. A model diagram layout extension for SBML. *Bioinformatics*, 22(15):1879–1885, 2006.

R. Gentleman, V. Carey, W. Huber, R. Irizarry, and S. Dudoit. *Bioinformatics and computational biology solutions using R and bioconductor*. Springer, 2005.

R. Gentleman, Elizabeth Whalen, W. Huber, and S. Falcon. *graph: graph: A package to handle graph data structures*, 2007. R package version 1.15.6.

GIDL. GIDL DTD, 2005. URL <http://svn.gnome.org/viewcvs/gobject-introspection/trunk/gidl.dtd>.

GOBJECT. The gObject reference manual, 2007. URL <http://developer.gnome.org/doc/API/2.0/gobject/index.html>.

F. Gong, Y.Z. Liang, Y.S. Fung, and F.T. Chau. Correction of retention time shifts for chromatographic fingerprints of herbal medicines. *Journal of Chromatography A*, 1029(1):173–183, 2004.

Alexander Gribov. Gauguin (grouping and using glyphs uncovering individual nuances), 2007. URL <http://rosuda.org/software/Gauguin/gauguin.html>.

Philippe Grosjean. tcltk2, 2006. URL <http://cran.r-project.org/src/contrib/Descriptions/tcltk2.html>.

GTK-Doc. GTK-Doc API Documentation Generator, 2007. URL <http://www.gtk.org/gtk-doc/>.

C.A. Hastings, S.M. Norton, and S. Roy. New algorithms for processing and peak detection in liquid chromatography/mass spectrometry data. *Rapid Communications in Mass Spectrometry*, 16(5):462–467, 2002.

He and Marriot. Constrained graph layout. *Constraints: An International Journal*, 3:289–314, 1998.

J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proc. of ACM Human Factors in Computing Systems (CHI'05)*, pages 421–430, Portland, OR, 2005.

Jeffrey Heer and Maneesh Agrawala. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5), 2006.

M. Helbig, S. Urbanek, and M. Theus. JGR: A Unified Interface to R. In *Proceedings of userR! 2004*, 2004. URL <http://www.ci.tuwien.ac.at/Conferences/useR-2004/>.

M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, , the rest of the SBML Forum:, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le Novere, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.

Moon Yul Huh and Kwangryeol Song. Davis: A java-based data visualization system. *Computational Statistics*, 17(3):411–423, 2002.

Catherine B. Hurley. The plot-data interface in statistical graphics. *Journal of Computational and Graphical Statistics*, 2(4):365–379, Dec. 1993.

C.B. Hurley and R.W. Oldford. Statistical graphics in quail: An overview. In *Biennial Meeting of the International Statistical Institute*, 1999.

H. Idborg, P.O. Edlund, and S.P. Jacobsson. Multivariate approaches for efficient detection of potential metabolites from liquid chromatography/mass spectrometry data. *Rapid Communications in Mass Spectrometry*, 18(9):944–954, 2004.

J.D. Jaffe, DR Mani, K.C. Leptos, G.M. Church, M.A. Gillette, and S.A. Carr. PEPPeR, a Platform for Experimental Proteomic Pattern Recognition. *Molecular & Cellular Proteomics*, 5(10):1927, 2006.

Ninad Jog and Ben Shneiderman. Starfield information visualization with interactive smooth zooming. In *Proceedings of IFIP 2.6 Visual Databases Systems*, College Park MD 20742-3255 USA, 1994.

K.J. Johnson, B.W. Wright, K.H. Jarman, and R.E. Synovec. High-speed peak matching algorithm for retention time alignment of gas chromatographic data for chemometric analysis. *Journal of Chromatography A*, 996(1):141–155, 2003.

R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis (5th ed)*. Prentice-Hall, Englewood Cliffs, NJ, 2002.

P. Jonsson, AI Johansson, J. Gullberg, J. Trygg, B. Grung, S. Marklund, M. Sjostrom, H. Antti, and T. Moritz. High-throughput data analysis for detecting and identifying differences between samples in GC/MS-based metabolomic analyses. *Anal Chem*, 77(17):5635–42, 2005.

T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

Tomihisa Kamada and Satoru Kawai. A general framework for visualizing abstract objects and relations. *ACM Trans. Graph.*, 10(1):1–39, 1991. ISSN 0730-0301. doi: <http://doi.acm.org/10.1145/99902.99903>.

Thomas Kamps, Joerg Kleinz, and John Read. Constraint-based spring-model algorithm for graph layout. In Franz J. Brandenburg, editor, *Graph Drawing, Passau, Germany, September 20-22*, pages pp. 349–360. Springer, 1996.

P.D. Karp and S. Paley. Automated drawing of metabolic pathways. In *Third International Conference on Bioinformatics and Genome Research*, pages 225–238, 1994.

S. Karpinski, H. Reynolds, B. Karpinska, G. Wingsle, G. Creissen, and P. Mullineaux. Systemic singaling and acclimation in response to excess excitation energy in *Arabidopsis*. *Science*, 284:654–657, 1999.

M. Katajamaa, J. Miettinen, and M. Oresic. MZmine: toolbox for processing and visualization of mass spectrometry based molecular profile data. *Bioinformatics*, 22(5):634, 2006.

M. Kirchner, B. Saussen, H. Steen, J.A.J. Steen, and F.A. Hamprecht. amsrpm: Robust Point Matching for Retention Time Alignment of LC/MS Data with R. *Journal of Statistical Software*, 18(4):o, 2007.

O. Kohlbacher, K. Reinert, C. Gropl, E. Lange, N. Pfeifer, O. Schulz-Trieglaff, and M. Sturm. TOPP—the OpenMS proteomics pipeline. *Bioinformatics*, 23(2):e191, 2007.

C. Kosak, J. Marks, and S. Shieber. Automating the layout of network diagrams with specified visual organization. *IEEE Transactions on Systems Man and Cybernetics*, 24(3):440–454, 1994.

Lauralynn Kourtz, Kevin Dillon, Sean Daughtry, Lara L. Madison, Oliver Peoples, and Kristi D. Snell. A novel thiolase-reductase gene fusion promotes the production of polyhydroxybutyrate in *arabidopsis*. *Plant Biotechnology Journal*, 3(4):435–447, 2005.

- Y. Kovtun, W.L. Chiu, G. Tena, and J. Sheen. Functional analysis of oxidative stress-activated mitogen-activated protein kinase cascade in plants. *Proc Natl Acad Sci USA*, 97:2940–2945, 2000.
- A. Krause. *Foundations of GTK+ Development*. Apress, 2007.
- M.D. Krebs, R.D. Tingley, J.E. Zeskind, M.E. Holmboe, J.M. Kang, and C.E. Davis. Alignment of Gas Chromatography-Mass Spectrometry Data by Landmark Selection from Complex Chemical Mixtures. *Chemometrics and Intelligent Laboratory Systems*, 81(1):74–81, 2006.
- W.L. Kubasek, B.W. Shirley, A. McKillop, H.M. Goodman, W. Briggs, and F.M. Ausubel. Regulation of flavonoid biosynthetic genes in germinating arabidopsis seedlings. *Plant Cell*, 4:1229–1236, 1992.
- J.M. Kwak, I.C. Mori, Z.M. Pei, N. Leonhardt, M.A. Torres, J.L. Dangl, R.E. Bloom, S. Bodde, JD Jones, and JI Schroeder. NADPH oxidase AtrbohD and AtrbohF genes function in ROS-dependent ABA signaling in Arabidopsis. *EMBO J*, 22:2623–2633, 2003.
- K. Lan and J.W. Jorgenson. A hybrid of exponential and gaussian functions as a simple model of asymmetric chromatographic peaks. *Journal of Chromatography A*, 915(1-2), 2001.
- Duncan Temple Lang. *REventLoop*, 2003. URL <http://www.omegahat.org/REventLoop/>. R package version 0.2-3.
- M. Lawrence. **rcola**: R binding to adaptograms implementation of IPSep-CoLa, 2007a. URL <http://www.ggobi.org/beta>.
- M. Lawrence and L. Drake. **cairoDevice**: Cairo-based cross-platform antialiased graphics device driver, 2007. URL <http://cran.r-project.org/src/contrib/Descriptions/cairoDevice.html>.
- M. Lawrence, E.K. Lee, D. Cook, H. Hofmann, and E. Wurtele. **exploRase**: Exploratory Data Analysis of Systems Biology Data. In *Proceedings of the Fourth International Conference on*

*Coordinated & Multiple Views in Exploratory Visualization*, pages 14–20. IEEE Computer Society Washington, DC, USA, 2006.

Michael Lawrence. RGtk2, 2007b. URL <http://www.ggobi.org/rgtk2>.

Michael Lawrence. Explorase website, 2007c. URL [http://www.metnetdb.org/MetNet\\_explorase.htm](http://www.metnetdb.org/MetNet_explorase.htm).

Michael Lawrence and Duncan Temple Lang. RGtk2: A graphical user interface toolkit for R. *Submitted to Journal of Statistical Software*, 2007. URL <http://www.ggobi.org/rgtk2>. Submitted.

Michael Lawrence, Dianne Cook, and Eun Kyung Lee. Explorase: Multivariate exploratory data analysis and visualization for systems biology. *Submitted to Journal of Statistical Software*, 2007a. URL [http://www.metnetdb.org/MetNet\\_explorase.htm](http://www.metnetdb.org/MetNet_explorase.htm). Submitted.

Michael Lawrence, Hadley Wickham, and Dianne Cook. GGobi Beta Homepage, 2007b. URL <http://www.ggobi.org/beta>.

Michael Lawrence, Hadley Wickham, Dianne Cook, Heike Hofmann, and Deborah Swayne. Extending the GGobi pipeline from R. *Submitted to Journal of Computational Statistics*, 2007c. URL <http://www.ggobi.org/beta>.

X. Li, R. Gentleman, X. Lu, Q. Shi, J.D. Iglehart, L. Harris, and A. Miron. *Bioinformatics and Computational Biology Solutions Using R and Bioconductor*, chapter SELDI-TOF Mass Spectrometry Protein Data, pages 91–109. Springer, 2005a.

X. Li, E.C. Yi, C.J. Kemp, H. Zhang, and R. Aebersold. A Software Suite for the Generation and Comparison of Peptide Arrays from Sets of Data Collected by Liquid Chromatography-Mass Spectrometry\* S. *Molecular & Cellular Proteomics*, 4(9):1328–1340, 2005b.

J. Listgarten, R.M. Neal, S.T. Roweis, and A. Emili. Multiple Alignment of Continuous Time Series. *Advances in Neural Information Processing Systems*, 17:817–824, 2005.

- J. Liu, A. Bell, J. Bergeron, C. Yanofsky, Carrillo B., C. Beaudrie, and R. Kearney. Methods for peptide identification by spectral comparison. *Proteome Science*, 5(3), 2007.
- Gunther Maier. simpleR: a Windows GUI for R. In *Proceedings of user! 2006*, 2006. URL <http://www-sre.wu-wien.ac.at/SimpleR/>.
- MathML. Mathematical Markup Language, 2006. URL <http://www.w3.org/Math/>.
- John A. McDonald. *Interactive graphics for data analysis*. PhD thesis, Stanford University, 1982.
- Microsoft Corporation. Distributed Component Object Model (DCOM) for Windows 98, 2007. URL <http://www.microsoft.com/com/default.mspx>.
- R. Mittler. Oxidative stress, antioxidants and stress tolerance. *Trends Plant Sci.*, 7:405–410, 2002.
- R. Mittler, S. Vanderauwera, M. Gollery, and F. Van Breusegem. Reactive oxygen gene network of plants. *Trends in Plant Science*, 9:10, 2004.
- A.W. Moore Jr and J.W. Jorgenson. Median filtering for removal of low-frequency background drift. *Analytical Chemistry*, 65(2):188–191, 1993.
- J.S. Morris, K.R. Coombes, J. Koomen, K.A. Baggerly, and R. Kobayashi. Feature extraction and quantification for mass spectrometry in biomedical applications using the mean spectrum. *Bioinformatics*, 21(9):1764, 2005.
- P. Mullineaux and S. Karpinski. Signal transduction in response to excess light: getting out of the chloroplast. *Curr. Opin. Plant Biol.*, 5:43–48, 2002.
- S. Neill, R. Desikan, and J. Hancock. Hydrogen peroxide signalling. *Curr. Opin. Plant Biol.*, 5:388–395, 2002.
- F.J. Newberry and W.F. Tichy. EDGE: An extendible graph editor. *Software Practice and Experience*, 20:63–88, 1990.

- N.P.V. Nielsen, J.M. Carstensen, and J. Smedsgaard. Aligning of single and multiple wavelength chromatographic profiles for chemometric data analysis using correlation optimised warping. *Journal of Chromatography A*, 805(1-2):17–35, 1998.
- J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley Reading, Mass, 1994.
- R. Penners. GTK-WIMP (Windows Impersonator), 2007. URL <http://gtk-wimp.sourceforge.net/>.
- L. Pnueli, H. Liang, M. Rozenberg, and R. Mittler. Growth suppression, altered stomatal responses, and augmented induction of heat shock proteins in cytosolic ascorbate peroxidase (Apx1)-deficient *Arabidopsis* plants. *Plant J*, 34:185–201, 2003.
- Y. Poirier, K. Klomparens, and C. Somerville. Polyhydrobutyrate, a biodegradable thermoplastic, produced in transgenic plants. *Science*, 256:520, 1992.
- A. Prakash, P. Mallick, J. Whiteaker, H. Zhang, A. Paulovich, M. Flory, H. Lee, R. Aebersold, and B. Schwikowski. Signal Maps for Mass Spectrometry-based Comparative Proteomics\*. *Molecular & Cellular Proteomics*, 5(3):423–432, 2006.
- Mono Project. Gapi, 2008. URL <http://www.mono-project.com/GAPI>.
- R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2005. URL <http://www.R-project.org>. ISBN 3-900051-07-0.
- Tomas Radivojevitch. A two-way interface between limited systems biology markup language and R. *BMC Bioinformatics*, 5(1):190, 2004. URL <http://www.biomedcentral.com/1471-2105/5/190>.
- R. Rew and G. Davis. NetCDF: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990.

A.F. Ruckstuhl, M.P. Jacobson, R.W. Field, and J.A. Dodd. Baseline subtraction using robust local regression estimation. *Journal of Quantitative Spectroscopy and Radiative Transfer*, 68(2):179–193, 2001.

Kathy Ryall, Joe Marks, and Stuart M. Shieber. An interactive constraint-based system for drawing graphs. In *ACM Symposium on User Interface Software and Technology*, pages 97–104, 1997.

SAS Institute. Jmp: Statistical discovery software, version 7, 2007. URL <http://www.jmp.com/>.

A.C. Sauve and T.P. Speed. Normalization, baseline correction and alignment of high-throughput mass spectrometry data. In *Proceedings of the Genomic Signal Processing and Statistics, 2004*, 2004.

Denise Scholtens. *SNAData: Social Networks Analysis Data Examples*, 2007. R package version 1.8.0.

F Schreiber. High quality visualization of biochemical pathways in biopath. *In silico Biology*, 2:6, 2002. URL <http://www.bioinfo.de/isb/2002/02/0006/>.

Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S. Baliga, Jonathan T. Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks. *Genome Res.*, 13(11):2498–2504, 2003.

Paul Shannon, David Reiss, Richard Bonneau, and Nitin Baliga. The gaggle: An open-source software system for integrating bioinformatics software and data sources. *BMC Bioinformatics*, 7(1):176, 2006. ISSN 1471-2105. doi: 10.1186/1471-2105-7-176. URL <http://www.biomedcentral.com/1471-2105/7/176>.

J. Siek, L.Q. Lee, and A. Lumsdaine. *The boost graph library: user guide and reference manual*. Addison-Wesley, 2002.

- J. Smart, K. Hock, and S. Csomor. *Cross-Platform GUI Programming with Wxwidgets*. Prentice Hall PTR, 2005.
- C.A. Smith, E.J. Want, G. O'Maille, R. Abagyan, and G. Siuzdak. XCMS: processing mass spectrometry data for metabolite profiling using nonlinear peak alignment, matching, and identification. *Anal. Chem.*, 78(3):779–787, 2006.
- G. K. Smyth. Limma: linear models for microarray data. In R. Gentleman, V. Carey, S. Dudoit, R. Irizarry, and W. Huber, editors, *Bioinformatics and Computational Biology Solutions using R and Bioconductor*, pages 397–420. Springer, 2005.
- SPSS Inc. *SPSS Base 16.0 for Windows User's Guide*. Chicago IL, 2007.
- SE Stein. An integrated method for spectrum extraction and compound identification from gas chromatography/mass spectrometry data. *J. Am. Soc. Mass Spectrom*, 10(8):770–781, 1999.
- SE Stein and DR Scott. Optimization and testing of mass spectral library search algorithms for compound identification. *J. Am. Soc. Mass Spectrom*, 5:859–866, 1994.
- Chris Stolte, Diane Tang, and Pat Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1), 2002.
- K. Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*. World Scientific Publishing Company, 2002.
- K. Sugiyama and K. Misue. Graph drawing by the magnetic spring model. *Journal of Visual Languages and Computing*, 6(3):217–231, 1995.
- K. Sugiyama, S. Tagawa, and M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Trans. on Systems, Man, and Cybernetics*, 11(2):109–125, 1981.
- Sun Microsystems. Java, 2007. URL <http://www.java.com/>.

- P. Sutherland, A. Rossini, T. Lumley, N. Lewin-Koh, J. Dickerson, Z. Cox, and D. Cook. Orca: A visualization toolkit for high-dimensional data. *Journal of Computational and Graphical Statistics*, 9(3):509–529, 2000.
- Deborah F. Swayne, Dianne Cook, and Andreas Buja. XGobi: Interactive Dynamic Graphics in the X Window System with a Link to S. In *American Statistical Association 1991 Proceedings of the Section on Statistical Graphics*, pages 1–8, Alexandria, VA, 1991. American Statistical Association.
- Deborah F. Swayne, Duncan Temple Lang, Andreas Buja, and Dianne Cook. GGobi: evolving from XGobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 43:423–444, 2003a.
- D.F. Swayne, A. Buja, and D.T. Lang. Exploratory Visual Analysis of Graphs in GGobi. *Proceedings of DSC*, 2:1, 2003b.
- D.L. Sweeney. Small Molecules as Mathematical Partitions. *Anal. Chem.*, 75(20):5362–5373, 2003.
- Marko Sysi-Aho, Mikko Katajamaa, Laxman Yetukuri, and Matej Oresic. Normalization method for metabolomics data using optimal selection of multiple internal standards. *BMC Bioinformatics*, 8(1):93, 2007. ISSN 1471-2105. URL <http://www.biomedcentral.com/1471-2105/8/93>.
- R. TAULER. Multivariate curve resolution applied to second order data. *Chemometrics and intelligent laboratory systems*, 30(1):133–146, 1995.
- R. Tautenhahn, C. Bottcher, and S. Neumann. Annotation of LC/ESI-MS Mass Signals. *Lecture Notes in Computer Science*, 4414:371, 2007.
- D. Temple Lang. RDCOM bundle of packages, 2005a. URL <http://www.omegahat.org/RDCOMBundle>.

- D. Temple Lang. *RDCOMEevents*, 2005b. URL <http://www.omegahat.org/RDCOMEevents/>. R package version 0.3-1.
- D. Temple Lang. Using XML for statistics: The XML package. *R News*, 1(1):24–27, January 2001a. URL <http://CRAN.R-project.org/doc/Rnews/>.
- Duncan Temple Lang. Rgcctranslationunit, 2006a. URL <http://www.omegahat.org/RGCCTranslationUnit/>.
- Duncan Temple Lang. RGtk, 2004. URL <http://www.omegahat.org/RGtk>.
- Duncan Temple Lang. RSPython, 2005c. URL <http://www.omegahat.org/RSPython>.
- Duncan Temple Lang. RxWidgets, 2007. URL <http://www.omegahat.org/RxWidgets/>.
- Duncan Temple Lang. GGobi meets R: an extensible environment for interactive dynamic data visualization. In *Proceedings of the 2nd International Workshop on Distributed Statistical Computing*, 2001b.
- Duncan Temple Lang. SJava, 2006b. URL <http://www.omegahat.org/RSJava/>.
- Martin Theus. Interactive data visualiating using Mondrian. *Journal of Statistical Software*, 7(11):1–9, 2003.
- L. Tierney. tkplot: TK Rplot, 2007. URL <http://cran.r-project.org/src/contrib/Descriptions/tkplot.html>.
- Tile. Tile extension to tcl/tk, 2007. URL <http://tktable.sourceforge.net/tile/>.
- K. Tobias and F. Oliver. Seven Golden Rules for heuristic filtering of molecular formulas obtained by accurate mass spectrometry. *BMC Bioinformatics*, 8(105), 2007.
- A. R. Unwin, G. Hawkins, H. Hofmann, and B. Siegl. Interactive Graphics for Data Sets with Missing Values - MANET. *Journal of Computational and Graphical Statistics*, 5(2):113–122, 1996.

Antony Unwin and Heike Hofmann. Gui and command-line - conflict or synergy? In K. Berk and M. Pourahmadi, editors, *Computing Science and Statistics*, 1999.

Simon Urbanek. rJava, 2006. URL <http://rosuda.org/rJava/>.

Simon Urbanek. *Exploratory Model Analysis. An Interactive Graphical Framework for Model Comparison and Selection*. PhD thesis, Universität Augsburg, 2004.

Simon Urbanek and Martin Theus. iPlots for R: Interactive java-based graphics. In *Proceedings of useR! 2003*, 2003.

AM van Nederkassel, M. Daszykowski, PHC Eilers, and Y. Vander Heyden. A comparison of three algorithms for chromatograms alignment. *Journal of Chromatography A*, 1118(2):199–210, 2006.

S. Vanderauwera, P. Zimmermann, S. Rombauts, S. Vandenabeele, C. Langebartels, W. Gruissem, D. Inze, and F. Van Breusegem. Genome-wide analysis of hydrogen peroxide-regulated gene expression in *Arabidopsis* reveals a high light-induced transcriptional cluster involved in anthocyanin biosynthesis. *Plant Physiology*, 139:806–821, 2005.

Paul F. Velleman. *Data desk. The New Power of Statistical Vision*. Data Description Inc, 1992.

John Verzani. The gWidgets toolkit, 2007a. URL <http://www.math.csi.cuny.edu/pmg>.

John Verzani. The Poor Man's GUI, 2007b. URL <http://www.math.csi.cuny.edu/pmg>.

B. Walczak and W. Wu. Fuzzy warping of chromatograms. *Chemometrics and Intelligent Laboratory Systems*, 77(1-2):173–180, 2005.

E. Walthinsen. GStreamer-GNOME Goes Multimedia. Technical report, GUADEC, April 2001.

P. Wang, Hua Tang, H. Zhang, J. Whiteaker, A.G. Paulovich, and M. McIntosh. Normalization regarding non-random missing values in high-throughput mass spectrometry data. In *Proc. Pac. Symp. Biocomput*, volume 11, pages 315–326, 2006a.

- P. Wang, H. Tang, M.P. Fitzgibbon, M. McIntosh, M. Coram, H. Zhang, E. Yi, and R. Aebersold. A statistical method for chromatographic alignment of LC-MS data. *Biostatistics*, 8(2):357, 2007.
- P.C. Wang, Y.Y. Du, G.Y. An, Y. Zhou, C. Miao, and C.P. Song. Analysis of global expression profiles of arabidopsis genes under abscisic acid and H<sub>2</sub>O<sub>2</sub> applications. *Journal of Integrative Plant Biology*, 48:62–74, 2006b.
- W. Wang, H. Zhou, Hua Lin, S. Roy, T.A. Shaler, L.R. Hill, S. Norton, P. Kumar, M. Anderle, and C.H. Becker. Quantification of proteins and metabolites by mass spectrometry without isotopic labeling or spiked standards. *Analytical chemistry*, 75(18):4818–4826, 2003.
- Matthew O. Ward. XmdvTool: Integrating multiple methods for visualizing multivariate data. In *IEEE Conf. on Visualization '94*, pages 326–333, Oct 1994.
- Chris Weaver. *Improvise: a user interface for interactive construction of highly-coordinated visualizations*. PhD thesis, University of Wisconsin-Madison, 2006.
- Chris Weaver. Patterns of coordination in improvise visualizations. In *Proceedings of the IS&T/SPIE Conference on Visualization and Data Analysis*, San Jose, CA, 2007.
- Katja Wegner and Ursula Kummer. A new dynamical layout algorithm for complex biochemical reaction networks. *BMC Bioinformatics*, 6:212, August 2005. URL <http://www.biomedcentral.com/1471-2105/6/212>.
- B.B. Welch. *Practical Programming in TCL and TK*. Prentice Hall PTR, 2003.
- Elizabeth Whalen. iSPlot: Linking plots, 2005. URL <http://bioconductor.org/packages/1.9/bioc/html/iSPlot.html>.
- Hadley Wickham. *classify: Explore classification models in high dimensions*, 2007a. URL <http://had.co.nz/classify>. R package version 0.2.3.
- Hadley Wickham. *ggplot2: An implementation of the Grammar of Graphics*, 2007b. R package version 0.5.6.

- A. Wilhelm. Interactive statistical graphics: the paradigm of linked views. In *Handbook of statistics 24: Data mining and data visualisation*, 2005.
- Leland Wilkinson. *The Grammar of Graphics*. Statistics and Computing. Springer, 2005.
- Graham Williams. Rattle: gnome R data mining, 2006. URL <http://rattle.togaware.com/>.
- Sylvia Winkler. *Parallele Koordinaten: Entwicklung einer interaktiven Software*. PhD thesis, Universität Augsburg, 2000.
- W.E. Wolski, M. Lalowski, P. Martus, R. Herwig, P. Giavalisco, J. Gobom, A. Sickmann, H. Lehrach, and K. Reinert. Transformation and other factors of the peptide mass spectrometry pairwise peak-list comparison process. *BMC Bioinformatics*, 6(1):285, 2005.
- E.S. Wurtele, J. Li, L. Diao, H. Zhang, C.M. Foster, B. Fatland, J. Dickerson, A. Brown, Z. Cox, D. Cook, E.K. Lee, and H. Hofmann. Metnet: Software to build and model the biogenetic lattice of arabidopsis. *Comp. Funct. Genom.*, 4:239–245, 2003.
- E.S. Wurtele, L. Ling, D. Berleant, D. Cook, J.A. Dickerson, J. Ding, H. Hofmann, M. Lawrence, E.K. Lee, J. Li, W. Mentzen, L. Miller, B.J. Nikolau, N. Ransom, and Y. Wang. *Concepts in Plant Metabolomics*, chapter MetNet: Systems Biology Software for Arabidopsis. Springer, 2007.
- W. Yu, X. Li, J. Liu, B. Wu, K.R. Williams, and H. Zhao. Multiple Peak Alignment in Sequential Data Analysis: A Scale-Space Based Approach. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 3(3):208–219, 2006.