RudolF: An Open-Source Sandbox for increasing the accessibility of Functional Programming to the Bioinformatics and Scientific Communities

Matthew Fenwick, Colbert Sesanker, Martin Schiller, Heidi JC Ellis, M. Lee Hinman, Jay Vyas, Michael Gryk

University of Connecticut Health Center, 263 Farmington Avenue Farmington, Connecticut 06030

Abstract

Scientists are continually faced with the need to express complex mathematical notions in code. The renaissance of functional languages such as LISP and Haskell is often credited to their ability to implement complex data operations and mathematical constructs in an expressive and natural idiom. The slow adoption of functional computing in the scientific community does not, however, reflect the congeniality of these fields. Unfortunately, the learning curve for adoption of functional programming techniques is steeper than that for more traditional languages in the scientific community, such as Python and Java, and this is partially due to the relative sparseness of available learning resources. To fill this gap, we demonstrate and provide applied, scientifically substantial examples of functional programming which we present as the multi-language RudolF source-code repository for software integration and algorithm development, which generally focuses on the fields of machine learning, data processing, bioinformatics. We encourage scientists who are interested in learning the basics of functional programming to adopt, reuse, and contribute to these examples. The source code is available at: https://github.com/jayunit100/RudolF (see also http://www.connjur.org).

Key Words: functional-programming; Clojure; Haskell; Java; bioinformatics; NMR; LISP

Introduction

Scientists express complex mathematical constructs and data translations in code. The modern age of science, wherein data sources are increasingly distributed and data types are increasingly complex, places unique demands on scientific programmers. The renewal of interest in functional programming languages in the past decade has enabled software engineering with increased modularity, composability, and programmer efficiency in many analytical domains, due to the fact that functional approaches are capable of "masking" the machine level implementation data editing which is necessary for complex data processing schemes [9]. The result is thus a natural, mathematical depiction of the way information flows through a particular problem domain.

The fields of protein bioinformatics and structural biology require integration of mathematical constructs, software libraries, and complex data structures that are difficult to express using conventional programming paradigms. Such repositories can easily span 1000s of lines of procedural code implementing complex operations matrix manipulation, combinatorial comparisons, and iteration through semantically dense data structures. Such endeavors are ideally suited to functional programming approaches: by hiding the details of the underlying data structures and machine-level data operations, functional approaches to scientific computing allows researchers to once again focus on the actual problem domain which they are interested in addressing, rather than any non-essential "accidental" computational complexity.

This manuscript introduces a sandbox for learning how these abstract concepts are applied to real world scientific analysis and visualization. We introduce and exemplify tidbits of the completely open-source RudolF source-code repository, which aims to increase the accessibility of practical functional programming constructs applied to real-world problems relevant to the bioinformatics community in a language neutral environment.

First, to demonstrate the power of functional abstraction for bioinformatics tasks, we describe its application to the definition of sample scheduling patterns of NMR for non-uniform data collection using Haskell (a purely functional language). The ability to rapidly define and integrate complex mathematical abstractions into the NMR sample scheduling workflow demonstrates the power of Haskell for implementing dynamic, experimental frameworks for mathematical programming, and provides a template to new researchers interested in abstracting the quantitative logic of their codebase so that it is both maintainable as well as easily modifiable.

Second, we also address one of the criticisms of functional programming, which is that, although it is powerful and expressive, there are libraries in other languages which do not readily port to different platforms. To address this concern, we exemplify the power and ease of integrating Clojure with pre-existing bioinformatics libraries for the standard Java programming langauge. The portable, tight JVM integration that Clojure provides is demonstrated as a data integration framework, interoperating with BioJava, a plotting tool via the Incanter toolset, and also by molecular visualization, with the popular Jmol library. In short, these provide scientists examples of how to accomplish powerful data analysis and visualization tasks in a practical, but functional, idiom using the Clojure language.

Active development of the RudolF sandbox is underway. There are examples of both prototypical scripts which briefly demonstrate important, less-known features of functional languages (such as GUI design, remote data integration, and language interoperability), as well as production ready code, which exemplify robust features of "real" source repositories – such as standardized unit tests, dependency management, and build automation.

We encourage contributions from programmers of different backgrounds, including developers from the Clojure, Haskell, Erlang, OCaml, and Ruby communities. Who are interested in learning more about problem-solving in functional programming, and applying their skills to complex problems.

There other existing examples, largely genome related, of functional programming work in the open-source bioinformatics community: The BioCaml project offers a variety of parsers for different genomic file formats (https://github.com/agarwal/biocaml), the official BioClojure project, deals with analysis and visualization of genomic data (https://github.com/jandot/bioclojure), and the BioHaskell project (http://biohaskell.org/) offers further analysis of genomic data. We expect that the RudolF project, which focuses on protein bioinformatics and machine learning, will instruct and prepare novice scientists for in the practical aspects of functional programming (unit testing, data integration, visualization) where didactic resources are laking.

Haskell Sample Scheduler

Background

NMR -- Nuclear Magnetic Resonance -- spectroscopy is a technique for collecting information about molecules. In the context of protein spectroscopy, it is applied to collect data which can be used to determine structure and dynamics. NMR experiments collect data in multidimensional (1 to 4 or more) grids, where each of the dimensions represents time or a pseudo-time dimension. Conventional experiments collect data evenly spaced, on-grid data points. The total amount of time to complete a data collection experiment depends on the number of points collected; thus, experimental time grows very quickly with the number of dimensions -- 4-dimensional experiments can take days to fully complete.

Not only is this expensive to run the spectrometer for such a long time, but there can also be negative consequences on the quality of the sample: samples may not be stable for the entire length of time required for the experiment, leading to detrimental effects on data quality. An alternative to uniform collection is non-uniform collection of on-grid data points. This reduces the time necessary to complete the experiment -- often, the time savings can be greater than 50% of the original experimental time. This technique has already been successfully applied. Additional proposals have shown the value of non-uniform collection in two other areas: quadrature units [1] and number of transients. Quadrature detection is used to distinguish between positive/negative data; 2n quadrature units are typically collected, where n is the number of dimensions. Non-uniform transient collection refers to the practice of collecting different points with different frequencies -- the values can then be combined in a way that reduces noise.

There have been a large number of published algorithms for generating various schemes of on-grid sampling. However, the need for an integrated platform which unifies these algorithms for the benefit of the programmer and user has become evident recently, with the advent of non-uniform quadrature detection and a renewed interest in non-uniform transient collection. Additionally, there are multiple spectrometer companies and tools for dealing with non-uniform data; typically, each of these has a special format -- another degree of complexity for the NMR spectroscopist to deal with. The goal of this project is to create such a platform which enables combination and reuse of the various algorithms by means of a coherent and complete data model.

Summary of Results

1) Data model of schedule

A data model was implemented in Haskell, as well as in MySQL, an open-source Relational Database Management System (RDBMS). A formal data model is necessary to enable computerized use and interpretation of sample schedules, as well as allowing researchers to easily share and describe schedules. The model we describe is a superset of existing, informal models; it provides additional descriptive power for specifying more rich and complex schedules and sampling schemes.

Specifically, the model includes support for non-uniform quadrature detection and transient selection. These two areas have been largely unexplored by NMR researchers. However, the richness of our model allows a user of the code to easily switch between uniform and non-uniform settings for both quadrature and transients -- while allowing continued use of traditional sampling schemes.

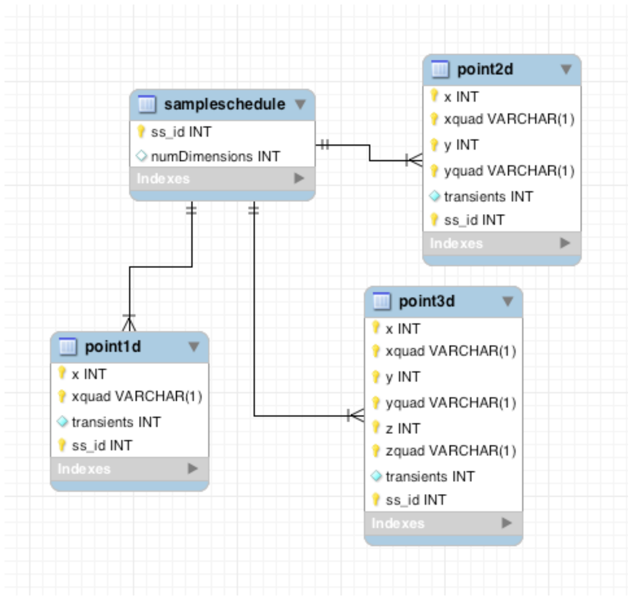


Figure 1: a MySQLWorkbench data model of a non-uniform sample schedule.

Basically, a sample schedule has an associated number of dimensions which determines the dimensionality of the grid points and quadrature units -- for example, a 1D schedule has 1D grid points and quadrature units, a 2D schedule has 2D points, and so on. Each unique combination of grid point and quadrature unit is termed a "point"; the number of times each point appears in the schedule is the number of "transients". In database terms, "point" + sample schedule identifier is the primary key of the point table.

2) Model of schedule creation workflow

We break down schedule creation into three distinct steps: schedule generation, point selection, and schedule modification. Many schedules can be created solely with a 'generation' step. However, more complicated schedules can not be expressed so simply; these are then expressed with combinations of generation, selection, and modification steps.

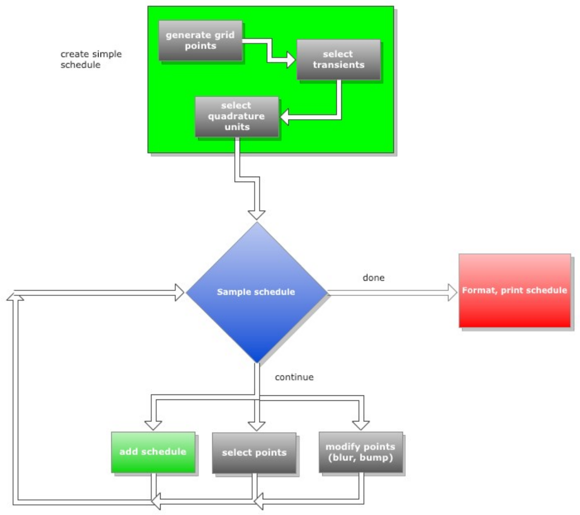


Figure 2: the process of building complex sample schedules.

Generation: points from a grid and quadrature units are combined to create a simple schedule. Also, the number of transients is selected for each point; the current default is to have one transient for each point, as non-uniform transient selection can be achieved by different means. However, this can easily be extended in future versions if desired.

Selection: points from a schedule are selected; this can involve selection with or without replacement. If without replacement, this operation will simply filter out some points. If with replacement, this operation results in non-uniformly transient detection. Additionally, there are options, during the selection stage, to select all transients together, all quadrature units of a grid point together, or all separate, allowing maximum flexibility.

Modification: many schedules can be improved by adding random noise, such as Gaussian blurring or bursty selection. Such an operation would typically adjust all of the points in a schedule in some way without removing any directly [2, 3].

Such a conceptualization of schedule creation provides maximum expressiveness, allowing a user to easily create schedules that would otherwise involve a large amount of work. (give examples where this is useful -- for instance, allLowerBounds)

3) Integrated, interactive environment for creation, evaluation of schedules

Loading the code into the Haskell interpreter (GHCi) allows the user to interactively create, view, and analyze sample schedules. The combination of data model and useful functions allows analysis of schedules at a very high level, while the interpreter allows flexible experimentation and combination with a variety of algorithms, allowing a user to identify potentially useful workflows, which can then be more robustly coded at a later time. The very low barrier of entry to creating and examining various sample schedules saves the NMR spectroscopist time.

4) Example schedules

Bundled with the code are a number of example sample schedules of practical interest to an NMR spectroscopist, including schedules demonstrating the Halton sub-random sampling scheme, exponential sampling scheme, non-uniform transients selection, non-uniform quadrature selection, and uniform grid sampling scheme. In addition to these schedules, the code/data used to create them is given, both as code, and as JSON configuration files.

5) Flexible export options -- schedule formatting

The code includes many options for schedule output, including all those used by common spectrometer companies, as well as the Rowland Toolkit, and a custom column-based format including transients, and a custom, unambiguous JSON format for explicit automated data transfer and sharing, possibly with other programs or through web services.

6) Implementation

This project was implemented using a purely functional programming language for a number of reasons. Haskell's rich type system facilitates type-safe programming at a very high level, which results in code that is more general, easier to comprehend, and much shorter. In many cases, complex algorithms were easy to implement and easy to read and understand once the project reached the maintenance phase. Additionally, the richness of its type system catches a whole host of typing errors at compile-time rather than at run-time -- allowing the programmer to have much more confidence in the quality of his/her code. Due to its nature as a functional language, Haskell code is inherently testable; additionally, an easily available Haskell library known as QuickCheck provides a very robust means of testing properties of code.

7) Future goals

The main future goal of this project is an application for creating sample schedules, usable from the command line or a Graphical User Interface (GUI). User-defined parameters, in a JSON-formatted text file or from the GUI, would be passed in to the program, which then uses the parameters to create and execute a schedule workflow. Errors could occur either when reading the parameters, or when executing the workflow; they would then be reported to the user in place of a schedule. The schedule may be output in any of a number of formats, including Varian and Bruker. The major advantage of using JSON parameter files is the ease of sharing, creating, and analyzing the parameters, due to the prevalence of JSON as a data format.

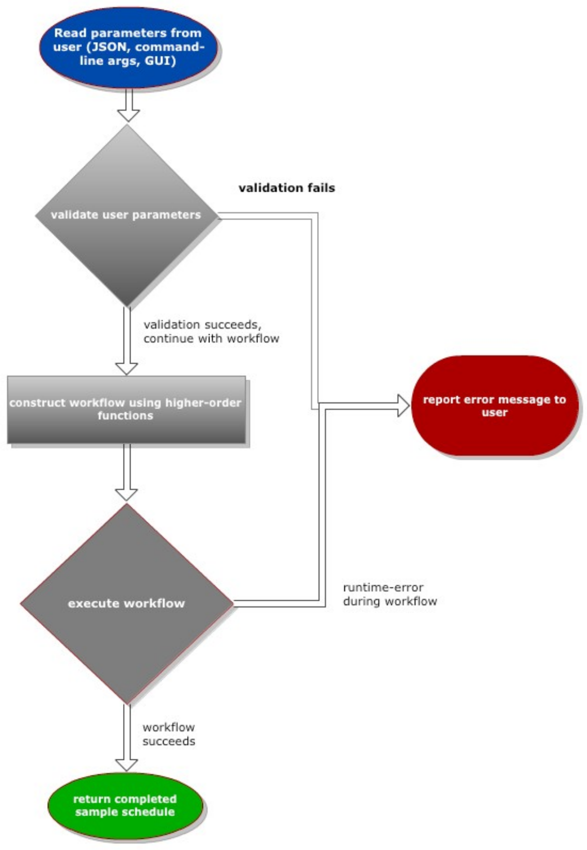

Read parameters from user (JSON, command-line args, GUI)
validation fails
validate user parameters
validation succeeds, continue with workflow
construct workflow using higher-order functions
report error message to user
execute workflow
runtime-error during workflow
workflow succeeds
return completed sample schedule

Figure 3: the flow of execution in the proposed sample scheduler functional program.

Java, LISP, and Bioinformatics : Three worlds Collide

The flexible and expressive nature of LISP languages have engendered support from theoretical computer scientists for several decades. In particular, the ability to customize LISP syntax for particular, domain specific problems has been leveraged in several domains. However, the lack of a standard platform for installing, learning, and using LISP has been a barrier to adoption in other areas. The emergence of the Clojure programming language, which is free, open source, and well supported, has changed this paradigm, enabling a JVM ready platform for building applications which can leverage the expressiveness of LISP with a lower barrier to entry.

One particular area of interest in our group has been integrated protein bioinformatics. We have previously deployed applications [4] which leverage the CONNJUR framework [5] for data integration, the BioJava library [6] for analyzing proteins sequences and structures, as well as the Jmol tool [7] for molecular visualization. One of the concerns that small laboratories face in designing large, data driven applications is the maintenance of their code base, which can grow quite rapidly. Such applications can benefit from not only the wide variety of numerical and scientific libraries available in mainstream languages (such as Java, C++, Perl, and Python), but also the conciseness and abstractness of functional code. Clojure provides a unique combination between availability of libraries and accessibility of function paradigms.

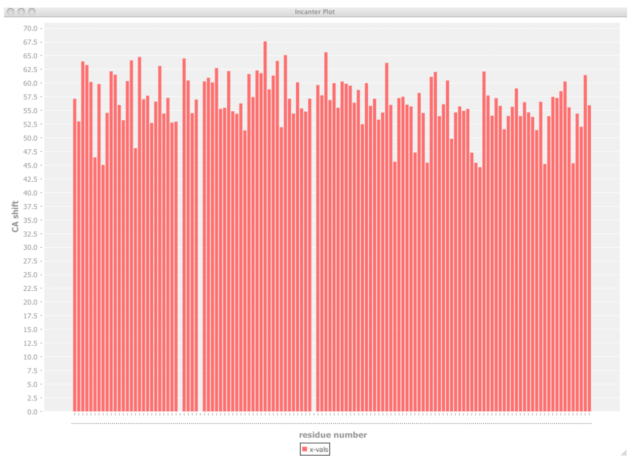
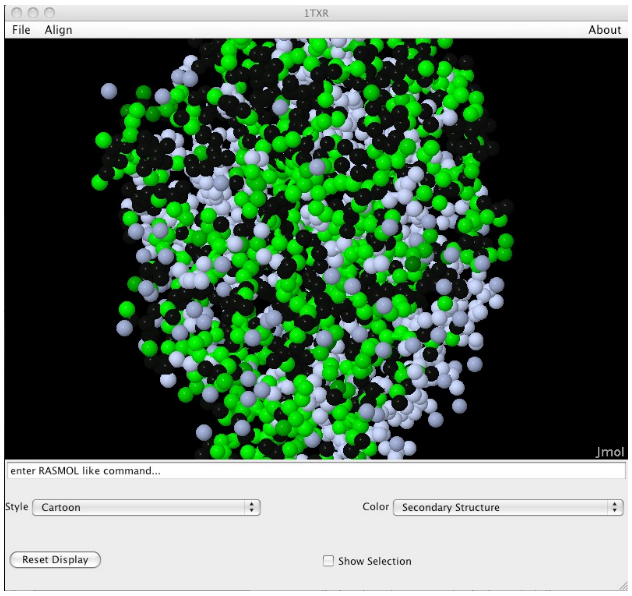


Figure 4: a plot of chemical shift of CA atom vs. residue number, created with Clojure, Incanter.

In order to exemplify the Clojure language for solving and integrating domain specific problems, we have deployed several examples of the integration of Clojure with the popular BioJava library for protein bioinformatics. For example, the RudolF source tree includes examples of how to remotely and integrate and test the functionality of BioJava into a Clojure project, examples of how to load, parse, and visualize protein structures from the PDB in just a few lines of code using the LISP-like syntax of Clojure, and the basic elements of data visualization for NMR-derived protein chemical shift data, by virtue of the Incanter library for data plotting.

Included are screen shots, and code snippets, which exemplify the expressive and functional nature of these operations - along with the robustness of the Clojure platform for integrating with real, computationally intensive Java processes at the API level. The ability to utilize Clojure in this context will open up new venues for the penetration of functional programming concepts into the Bioinformatics community, while also showcasing the ability of modern LISP dialects to satisfy the multidimensional requirements of the modern scientific programmer - who must not only design new algorithms, but also engineer solutions for maintaining and visualizing the implementation of such constructs.



Style Cartoon
Color Secondary Structure

Figure 5: a plot of color-mapped chemical shift values, created with Clojure, BioJava and Jmol.

Since Clojure is Java-based, we are able to quickly and easily package the source code into executable .jar files, which can be run by any computer equipped with the Java Virtual Machine (JVM) 1.6 or higher. This was accomplished using the 'leiningen' build tool for dependency management and project builds.

A Clojure Amino Acid Predictor

Background

The Clojure Amino Acid Predictor is a machine learning algorithm based on a binary Support Vector Machine (SVM) . In binary SVM algorithms, examples are represented by sparse vectors, in $\mathbf{R}^n$,where each entry of the vector holds the *value* of a *feature*, corresponding to the index of the entry. For example, if one is trying to categorize a set of journal abstracts into one of two classes, biology and astronomy, the abstracts represent examples and selected keywords in the abstract, corresponding to each index of the sparse vector, could represent features. The frequency of keywords in the abstracts represent the values at the feature index in the sparse example vector. The learning model, $\mathbf{f}$, learns to associate each sparse vector, $\mathbf{X = (x_1 , x_2 , . . . x_n)}$, with either the positive class, $\mathbf{f(X)} > 0$, or the negative class, $\mathbf{f(X)} < 0$, via the mapping $\mathbf{f : X} \rightarrow \mathbf{Y}$, $\mathbf{Y} = \{-1 , +1\}$, where elements in the negative class are assigned to -1 and elements in the positive class are assigned to +1. The model learns by maximizing its projection (minimizing the L_2 norm) onto elements in the positive class via the inner product. It gets closer to elements in the positive class, and further from elements in to negative class, gradually partitioning $\mathbf{R}^n$, into two disjoint sets bounded by a hyperplane;

positive examples are on one side of the hyperplane and negative examples on the other [8].

Design

The essence of the Clojure Amino Acid Predictor is to treat a *neighborhood*, e.g., "GLAMS", centered about an amino acid of interest, in this case A, as an example and define features about A in that neighborhood. Neighborhoods represent the decomposition of a peptide sequence into local examples and know nothing about other examples extracted from the sequence. We teach our model to guess the center of a neighborhood given inner, LM, and outer, GS, neighbors (in the example neighborhood of "GLAMS"). To distinguish inner from outer neighbors a "1" is appended to inner neighbors, LM → LM1, and a "2" is appended to outer neighbors, GS → GS2. Neighborhoods capture local information about each amino-acid, and we train our SVM classifier to recognize this information. We experiment with defining different features and neighborhoods; we aspire to develop more sophisticated neighborhoods and features for improved classification.

This project makes liberal use of Clojure's lazy sequence evaluation, combining infinite and finite lists as exemplified in this short and expressive code snippet:

```
(def protein-neighborhood (let [ s (stringseq-tuple protein-neighbors)]
  (zipmap (into (stringseq s (repeat "1")) (stringseq s (repeat "2")))
         (iterate inc 1))))
```

In these three lines of code, we create a map, `protein-neighborhood,` of all 21*21*2 = 882 possible inner and outer neighbors given a 21 letter alphabet of amino acids; each inner/outer neighbor is indexed from 0-881. `protein-neighbors` is a list of all ordered, 21*21= 441, pairs of amino acids in tuple format. The function **`stringseq-tuple`** turns this list of tuples into a list of ordered strings while **`stringseq`** lazily concatenates each element of the ordered string list with an infinite sequence of ones. The map, `protein-neighborhood,` can also be used a function that returns the index of an inner/outer neighbor:

```
(defn get-protein-neighbor-index [protein-neighbor] "String -> Int"
       (protein-neighborhood protein-neighbor))
```

(`get-protein-neighbor-index` "LS1") returns 783, the index for the inner neighbor LS1. Now, the idea is to take a peptide sequence and find all the neighbors, a straightforward implementation of regular expressions in Clojure:

```
(defn target-neighbors [string] ;; returns neighborhoods about desired amino acid
 (let [ matches (re-seq #"..[A].." string) ]
   matches))
```

This function returns all the neighborhoods centered about A in a peptide sequence. For Instance, calling the function `target-neighbors` on the sequence `"LMAGSAPW. . ."` yields the sequence of neighborhoods `("LMAGS" "GSAPW" . . .)`. In addition, negative neighborhoods, examples in the negative class, are generated identically to `target-neighbors` except the regular expression `"..[^A].."` is used in place of `"..[A].."` to match all the neighborhoods centered around every amino acid except A. These error neighborhoods train the model to avoid examples not associated with features centered around A.

Rationale and Future Work

The use of Clojure in a computationally extensive machine learning task evidences its potential as an alternative to mainstream languages such as C++ for the computational science community. Clojure's core library of functions, lazy sequences, expressiveness, and emphasis on functional code allow for easy parsing of files, implementation of numerical procedures and parallel computation. We are currently devising strategies for exploiting Clojure's support for parallelism to transform the Amino Acid Predictor into a multi-classifier by running multiple binary classification problems in parallel for different amino acids. The Hadoop framework offers tools to make this parallelism easier and more efficient. We are also investigating feature extraction techniques; in particular, Non-Negative Matrix factorization, to reduce the dimensionality of the example space far below all possible orderings of the features we choose to define in our neighborhood. The end goal of the of this project is to create an application that reads a training set of FASTA peptide sequence files and learns to predict amino acid gaps in new proteins related to the training set proteins.

RudolF on the web

The source code is free and open source, and available on our github page at https://github.com/jayunit100/RudolF. Contributions of any kind, including suggestions, documentation support, testing, and new languages or algorithm sandbox implementations, are welcomed and encouraged.

Methods

We used the command-line tool `git`, to provide local source control capabilities, together with github for remote, shared, and distributed source control between members of the group. For java dependency management and project builds, we have employed `leiningen`, a script specifically targeted at Clojure, and built on top of maven, for such tasks.

The Future of Bioinformatics and Functional Programming

Through this paper, we hope to establish the practicality, value and usefulness of functional programming to the bioinformatics and NMR communities and to the scientific programming community at large. It is our belief that the inherent advantages of functional programming will lead it to continue to grow in popularity in the coming years; we hope that project RudolF will provide guidance, motivation, and a place of learning for computing scientists interested in learning about and applying the benefits of functional programming to biologically relevant problems.

RudolF is named after the popular reindeer RudolF, who, although initially mocked by his peers, was ultimately capable of guiding Santa's sleigh through the dark, wintery skies on Christmas eve. Functional programming enthusiasts will certainly identify with this metaphor.

References

1) Mark W. Maciejewski, Matthew Fenwick, Adam D. Schuyler, Alan S. Stern, Vitaliy Gorbatyuk, and Jeffrey C. Hoch. Random phase detection in multidimensional NMR. *PNAS* vol. 108 no. 40, 2011. p. 16640 – 16644.

2) Jeffrey C. Hoch, Mark W. Maciejewski, Blagoje Filipovic. Randomization improves sparse sampling in multidimensional NMR. *Journal of Magnetic Resonance*, vol 183, no. 2, 2008, p. 317-320.

3) Mark W. Maciejewski, Harry Z. Qui, Iulian Rujan, Mehdi Mobli, Jeffrey C. Hoch. Nonuniform sampling and spectral aliasing. *Journal of Magnetic Resonance*, vol 199, no. 1, 2009, p. 88-93.

4) Jay Vyas, Michael R. Gryk and Martin R. Schiller. VENN, a tool for titrating sequence conservation onto protein structures. *Nucleic Acids Research*, vol. 37, no. 18, 2009

5) Ronald J. Nowling, Jay Vyas, Gerard Weatherby, Matthew W. Fenwick, Heidi J. C. Ellis and Michael R. Gryk. CONNJUR spectrum translator: an open source application for reformatting NMR spectral data. *Journal of Biomolecular NMR* vol 50, no. 1, 2011, p. 83-89

6) R. C. G. Holland, T. A. Down, M. Pocock, A. Prlic, D. Huen, K. James, S. Foisy, A. Drager, A. Yates, M. Heuer and M. J. Schreiber. BioJava: an open-source framework for bioinformatics. *Bioinformatics* vol 24, no. 18, 2008

7) Jmol: an open-source Java viewer for chemical structures in 3D. http://www.jmol.org/

8) Shai Shalev-Shwartz, Yoram Singer, Nathan Srebro. Pegasos: Primal Estimated sub-GrAdient Solver for SVM. 24th *International Conference on Machine Learning, Corvallis, OR, 2007.*

9) Konrad Hinsen. The Promises of Functional Programing. *Computing in Science & Engineering, July/August 2009, pp. 86 –90.*