Short Specialist Review

Perl in bioinformatics

R. Hannes Niedner and T. Murlidharan Nair
University of California, San Diego, CA, USA

Michael Gribskov
Purdue University, West Lafayette, IN, USA

1. Introduction

Advances in biology have generated an ocean of data that needs careful computational analysis to extract useful information. Bioinformatics uses computation and mathematics to interpret biological data, such as DNA sequence, protein sequence, or three-dimensional structures, and depends on the ability to integrate data of various types. The Perl language is well suited to this purpose.

An important practical skill in bioinformatics is the ability to quickly develop scripts (short programs) for scanning or transforming large amounts of data. Perl is an excellent language for scripting, because of its compact syntax, broad array of functions, and data orientation. The following are a few of the attributes of Perl that make it an attractive choice.

- Perl provides powerful ways to match and manipulate strings through the use of regular expressions. Changing file formats from one to another is a matter of contorting strings as required.
- Modularity that makes it easy to write programs as libraries (called modules).
- Perl system calls and pipes can be used to incorporate external programs.
- Dynamic loaders of Perl that help extend Perl with programs written in C as well as create compiled libraries that can be interpreted by the Perl interpreter.
- Perl is a good prototyping language and is easy to code. New algorithms can be easily tested in Perl before using a rigorous language.
- Perl is excellent for writing CGI scripts to interface with the Web.
- Perl provides support for object-oriented program development.

Projects such as the sequencing of the human genome produce vast amounts of textual data. In its early stages, the human genome project faced issues with data interchange between groups that were developing software, and Lincoln Stein has noted how Perl came to the rescue (Stein, 1996). Perl is certainly not the only language that possesses these positive features – many of them are found, as well, in other scripting languages such as Python (*see* Article 103, **Using the Python programming language for bioinformatics**, Volume 8).

In the following section, we discuss some of the important Perl features that make it attractive to bioinformaticians. We refer in this article to Perl version 5.6–5.8, but encourage you to look into the exciting new features of Perl 6, which has been reengineered from the ground up (**http://dev.perl.org/perl6/**, 2005).

2. Why use Perl?

Looking at today's landscape of programming languages, one is offered a wide variety from which to choose. The difference between compiled and interpreted languages is explained sufficiently in other articles in this series (Sanner, 2004), thus it suffices here to say that compiled languages are usually selected for their superior performance, as reflected by the C/C++ implementation of computationally demanding algorithms such as BLAST, ClustalW, or HMMer, or for their specific capabilities such as comprehensive user interfaces or graphical libraries. Scripting languages (often used synonymously for interpreted languages) such as Perl, Python, PHP, Ruby, Tcl, and so on, on the other hand, are often favored for fast prototyping since the development cycle does not require code recompilation. With the ever-increasing computational power (Moore's Law) (Moore, 1965) available for bioinformatics tasks, performance issues lose their importance in contrast to ease of development and code maintainability. In the search for the right programming language, one might also encounter the distinction of object-oriented and procedural programming accompanied with evaluations of which languages to choose for each of these programming approaches. Again, detailed publications exist on this topic (Valdes, 1994; Kindler, 1988; **http://www.archwing.com/technet/technet_OO.html**, 2001; **http://search.cpan.org/~nwclark/perl-5.8.6/pod/perltoot.pod**, 2005); thus, we restrict ourselves to some brief explanations here.

Until recently, the most popular programming languages (e.g., FORTRAN, BASIC, and C) were procedural – that is, they focused on the ordered series of steps needed to produce a result. Object-oriented programming (OOP) differs by combining data and code into indivisible components, called objects, which are abstract representations of "real-life" items. These objects model all information and functionality as required by the application. OOP languages include features such as "class", "instance", "inheritance", and "polymorphism" that increase the power and flexibility of an object.

In bioinformatics, a DNA sequence might be represented as an object inheriting from a more general implementation that covers the properties of all biological sequences. OOP would code this object by describing its properties such as length, checksum, and certainly the string of letters that comprises the sequence itself. Then one would implement accessor methods to retrieve or set these properties, and also more complex functions such as transcribe() that would take as an argument an organism-specific codon matrix and transform the DNA object to an RNA object. In procedural (or sequential) programming, the code flow is driven by the "natural" sequence of events. OOP is generally regarded as producing more reusable software libraries and better APIs (application programming interfaces) (**http://www.archwing.com/technet/technet_OO.html**; **http://search.cpan.org/∼nwclark/perl-5.8.6/pod/perltoot.pod**, 2005).

While all these aspects certainly matter in one's choice of a language, our experience is that bioinformatics programming frequently is not done by computer scientists or trained programmers, but rather by researchers in bioscience who are trying to make sense out of their experiments. The priority is not the production of elegant and maintainable code but rather speed and ease of programming. As Larry Wall puts it, "... you can get your job done with any 'complete' computer language, theoretically speaking. But we know from experience that computer languages differ not so much in what they make possible, but in what they make easy." In that regard, "Perl is designed to make the easy jobs easy, without making the hard jobs impossible" (Wall *et al*., 2000).

The high-level data structures built into Perl scale to any size and are merely restricted by the boundaries of the operating system and the amount of memory available on the host machine. Perl supports both procedural and object-oriented programming approaches and runs on virtually on all flavors of Unix and Linux. In addition, ActivePerl from ActiveState (**http://www.activestate.com/Products/ActivePerl/**, 2005) allows Perl to run even on Windows computers, and MacPerl (**http://dev.macperl.org/**, 2005) on Apple computers running System 9 and lower.

3. But Perl code is cryptic?!

At its simplest, Perl requires no declaration of variables. Variables belong to one of three classes: scalars (the scalar class supports strings, integers and floating point numbers), arrays, and associative arrays (arrays indexed by strings). Each type of variable is implicitly identified by beginning with a specific character: "$" for scalars, "@" for arrays, and "%" for associative arrays. Storage for variables is created when they are first referenced and released automatically when they pass out of scope. This automatic garbage collection is one of the high-level language features that makes Perl very convenient. Strings, integers, and floating point numbers are automatically identified and converted from one type to another as required. This is another powerful simplifying feature of Perl.

As with any high-level language, one can write very cryptic code in Perl – the distinction is that Perl has frequently been criticized for this. While the lack of a requirement for declaring variables exacerbates this problem and the automatic creation and initiation of variables can be a headache when typographical errors occur, these Perl-specific criticisms are, in our view, largely a red herring. Many style guides (e.g., perlstyle, (**http://search.cpan.org/∼krishpl/pod2texi-0.1/perlstyle.pod**, 2005)) point out some common practices, not necessarily specific to Perl, that lead to "cryptic" code:

- usage of meaningless variable and function names (e.g., x1);
- removing extra (but structuring) white spaces and parentheses;
- writing more than one instruction (ended by a semicolon) on one line; or

- active attempts to obfuscate your code such as substitution of numeric values with the arithmetic expressions or usage of hexadecimal character codes for all characters in strings to name just a few.

Perl provides support for better programming style (for instance, the "strict" pragma that requires explicit declaration of variables) and for internal documentation (via the pod mechanism). Perl also supports high-level constructs such as subroutines (with support for both call-by-reference and call-by-value), libraries (called modules in Perl), and classes.

Perl is not strict about the completeness of the data. Missing or odd characters are not penalized. This can be thought as a downside of Perl, since Perl will perform operations on mixed types without complaining. This can come back to haunt you when spelling mistakes lead to logical errors, but can be easily circumvented by using the strict pragma and taking advantage of the many debugging features built into Perl, such as the using the -w switch to increase the verbosity of warnings and messages (**http://search.cpan.org/∼nwclark/perl-5.8.6/pod/perlmodlib.pod#Pragmatic_Modules**, 2005; Gutschmidt, 2005).

4. Numerical calculations

With all the characteristics of a loosely typed language discussed so far, one might conclude that Perl cannot be used for numerical computations, but this is certainly not the case. Perl performs numerical calculations as double-precision floating point operations, and thus can be used for most computations that do not involve very high numerical precision. The Math::BigInt (**http://search.cpan.org/∼tels/Math-BigInt-1.77/lib/Math/BigInt.pm**, 2005) and Math::BigFloat (**http://search.cpan.org/∼tels/Math-BigInt-1.77/lib/Math/BigFloat.pm**, 2005) provide high-precision arithmetic for integers and floating-point numbers at the cost of somewhat slower execution in case you require more precision.

One can thus comfortably implement the classical algorithms such as learning by back-propagation of errors or Cleveland's locally weighted regression (LOWESS) in Perl. These are very frequently used algorithms in Bioinformatics for pattern recognition and microarray data normalization respectively. Having said this, however, Perl is nearly always a poor choice for large "number crunching". This is because Perl represents floating point numbers internally as strings and every numerical operation requires a conversion of string to floating point and back again. This can make Perl programs very slow if extensive calculations are required.

5. Regular expressions – Perl's most famous strong point

Pattern matching in text is simple and straightforward in Perl. Our experience is that code that can extend over a half a page in C or C++ can be easily written in one or two lines of Perl. Frequently, bioinformatics programs screen text documents for tokens that are identified by complex patterns. Regular expressions ("regex's" for short) are sets of symbols and syntactic elements used to match patterns of text.

They make it easy to represent complex patterns. For instance, searching for the pattern “ATT(C-or-G)(C-or-G)AAT” within 500 bases upstream of a promoter can be easily accomplished using this regular expression:

```
ATT[CG][CG]AAT.{0,500}TATAA
```

(the promoter is characterized by the Goldstein-Hogness or TATA box (Sudhof *et al*., 1987)).

Perl’s built-in support for regular expressions is second to no other language, and there are three typical operations done with regular expressions:

```
Match:   m/PATTERN/;
Replace: s/PATTERN/REPLACEMENT/;
Split:   @tokenlist=split /PATTERN/, $string;
```

Additional facilities allow the pattern matching to be very flexible, for instance to be case-insensitive, to apply to every occurrence (global matching), or to translate sets of characters (useful for changing case or converting DNA Ts to RNA Us). It is simple to transform genetic sequences formatted with spaces and numbers into plain strings of letters as often required for further analyses:

$$\$\text{sequence} = \text{s/[\^{}ACGT]//g} \tag{1}$$

The above expression replaces every character that is NOT a capital A, C, G, or T with an empty string (well nothing), thus stripping all nonsequence characters (including carriage returns), lower case letters, and numbers from the sequence string. It is outside the scope of this article to provide a tutorial on regular expression, but several on-line and printed publications on this topic are available (Friedl, 2002; **http://www.regular-expressions.info/**, 2005).

Regular expressions also support bioinformatics resources such as, for example, Prosite (Sigrist *et al*., 2002) that provide access to organized collections of sequence motifs expressed as patterns.

6. File handling and databases

“Bioinformation”, that is, data produced within the biosciences, is often produced as flat files (tag-field formatted text files). File formats developed by the Protein Data Bank (PDB) (Berman *et al*., 2002), Swissprot (Bairoch and Boeckmann, 1991), and Genbank (Benson *et al*., 2003) have become quasi-standards within the bioinformatics community. Perl makes it very easy to traverse large file systems, reading from and writing to files, and using its rich regular expression syntax to parse and transform textual file content. An example would be extracting the sequence string out of hundreds of individual files in Genbank format, while leaving out the information describing taxonomy, sequence features, bibliographic references, and other annotations. With these features, it is easy to understand why Perl naturally became the number one choice for the bioinformatics pioneers.

With the advent of relational databases, many bioinformatics repositories have been undergoing major reengineering efforts to reorganize their data storage technology from file repositories to relational databases. In addition, many labs and companies use relational database management systems (RDBMS) to store and organize annotation and experimental data. In Perl the Database Interface module (DBI) (**http://dbi.perl.org/**, 2005) provides a standardized and common interface for writing scripts that reference RDBMSs. Specific database drivers (DBDs) are called by DBI to communicate with particular RDBMS. The following code sample establishes a database connection ($dbh), which can be used to execute SQL commands:

```
use DBI;
my $dsn = 'DBI:mysql:my_database:localhost';
my $db_user_name = 'admin';
my $db_password = 'secret';
my $dbh = DBI->connect($dsn, $db_user_name,$db_password);
```

This modular architecture allows programmers to clearly separate common and RDBMS-specific code. While facilitating the production of database-independent code, DBI allows one to take full advantage of the features of a particular RDBMS via the specific functionality implemented in the DBD, or by handing off vendor-specific SQL statements directly to the RDBMS.

7. CPAN – the on-line Perl code library

Perl has been widely used not only by bioinformaticians but also by programmers in many other areas. Consequentially, there is an enormous wealth of Perl code available in the form of reusable software libraries, also called Perl modules (e.g., the DBI module mentioned above). The open repository for these Perl modules is the Comprehensive Perl Archive Network or CPAN for short. CPAN can be accessed not only at **http://www.cpan.org** (or many mirror sites) but also from within the standard Perl installation using the CPAN.pm module (**http://search.cpan.org/~andk/CPAN-1.76/lib/CPAN.pm**, 2005). When using Perl, you are part of a huge community – take advantage of it. Check CPAN first before you start implementing any major programming task since there is a very good chance that at least the core functionality is already available there.

8. Perl and the World Wide Web

Perl is widely used to develop web applications, and was adopted as the de facto language for creating content on the World Wide Web. Perl's powerful text manipulation facilities have made it an obvious choice for writing Common Gateway Interface (CGI) scripts. CGI is a standard for external gateway programs to interface with information servers such as http (or web) servers. An HTML document that the web server retrieves is static, a CGI script, on the other hand, is executed in real time upon user request (via the web client) and helps to dynamically create a webpage. The standard Perl installation comes with the CGI module (CGI.pm), which can be used to streamline the creation of web pages. CGI.pm provides functionality to handle the GET and POST protocols, to receive and parse CGI-queries, to create Web fill-out forms on the fly, and to parse their contents and to generate HTML elements (e.g., lists and buttons) as a series of Perl functions, thus sparing you from the need to incorporate HTML syntax into your code (**http://stein.cshl.org/WWW/software/CGI/**, 2005).

The results of data analyses often require display in the form of charts and graphics. The GD module provides basic tools necessary for this. GD.pm is a Perl interface to Thomas Boutell's C-based gd graphics library that enables Perl programs to create color drawings using a large number of graphics primitives, and to format the images in PNG (**http://stein.cshl.org/WWW/software/GD/**, 2005) and other formats.

Perl also has a set of ready-to-use libraries to implement conventional http-based as well as the more recently developed RSS-, RPC-, and SOAP-based web services. This is important when one implements fully automated data retrieval from online services. For example, SOAP::Lite for Perl is a collection of Perl modules that provides a simple and lightweight interface to the Simple Object Access Protocol (SOAP) both on client and server side (**http://www.soaplite.com/**, 2005).

The LWP (Library of WWW access in Perl) modules provide the core of functionality for web programming in Perl. It contains the foundations for networking applications, protocol implementations, media type definitions, and debugging ability. Most notably, with LWP::UserAgent you can build "robots" to access remote websites as part of a program or even build your own robust web client (**http://search.cpan.org/∼gaas/libwww-perl-5.803/lib/LWP.pm**, 2005).

9. XML processing

Perl's strong text parsing abilities make it no wonder that a host of modules have been developed to apply the power of Perl (and especially its regular expression syntax) to XML. The available modules cover the full range of XML standards, such as SAX and DOM, and are implemented either in Perl or provide a Perl interface to C-based libraries such as xerces, expat, or libxml/libxslt (**http://perl-xml.sourceforge.net/**, 2005).

10. BioPerl

BioPerl, the first in the series of Bio* projects, is an international association of developers of open source Perl tools for bioinformatics, genomics, and life science research formed in the early nineties and officially organized in 1995. BioPerl is a coordinated effort to convert and collect computational methods routinely used in bioinformatics and life science research into a set of standard CPAN-style, well-documented, and freely available Perl modules (**http://bioperl.org/**, 2005). BioPerl provides you with a rather complete set of sequence analysis functions and more, thus the following is by no means a comprehensive listing:

- accessing the major biological databases for data retrieval;
- reading and converting all major sequence file formats;
- extracting sequence, parameters, and annotation from flat files (source data as well as program outputs);
- providing an interface to ClustalW, HMMer, BLAST, FastA, and other standard bioinformatics applications;
- parsing/representing protein structure (PDB) data;
- traversal of phylogenetic trees.

Usually, there is a steep learning curve before a novice programmer gets around to actually using BioPerl. This is because it is a complex collection of modules, it is not trivial to install, and requires an understanding of object-oriented programming. It is worth the effort required to work through the comprehensive documentation and examples provided on the World Wide Web since BioPerl can tremendously accelerate the development of complex and feature-rich applications after one masters the initial hurdles. In the event that you decide to go with your own implementation, keep in mind that you do not have to install BioPerl to use it – the BioPerl developers recommend "you just steal the routines in there if you find any of them useful" (**http://bio.cc/Bioperl/index_bioperl_original.html**, 2005).

11. Where to learn more

A tutorial on using Perl in Bioinformatics is presented in Article 112, **A brief Perl tutorial for bioinformatics**, Volume 8. Additional sources include:

Books:

- "Beginning Perl for Bioinformatics" by James Tisdall and published by O'Reilly and Associates Inc.
- "Sequence Analysis in a Nutshell" by Darryl León, Scott Markel and published by O'Reilly and Associates Inc.
- "Learning Perl" by Randal L. Schwartz, Tom Christiansen and published by O'Reilly and Associates Inc.
- "Programming Perl" by Larry Wall, Tom Christiansen and Randal L. Schwartz and published by O'Reilly and Associates Inc. (the Camel Book)
- "Perl Cookbook" by Tom Christiansen and Nathan Torkinton and published by O'Reilly and Associates Inc.
- "Mastering Algorithms with Perl" Jon Orwant, Jarkko Hietaniemi & John Macdonald and published by O'Reilly and Associates Inc.
- "Data Munging with Perl" by David Cross and published by Manning.
- "Object Oriented Perl" by Damian Conway and published by Manning.
- "Mastering Regular Expressions" by Jeffrey E. F. Friedl and published by O'Reilly and Associates Inc.

Websites:

- **http://www.cpan.org/** – the main CPAN site
- **http://www.bioperl.org/** – the BioPerl home
- **http://www.tpj.com/** – the Perl Journal
- **http://www.perl.com/** – the O'Reilly Perl site
- **http://bio.oreilly.com/** – the O'Reilly Bioinformatics site

12. Conclusion

Bioinformatics is surging ahead at an increasing pace. The demand for the development of tools needed to analyze biological data is ever increasing. Developing scripts in Perl for prototyping, or suites of reusable Perl modules, allows one to contribute to a large research community developing and sharing Perl programs. As with any other programming language, fluency in the language comes with experience; clarity of the code largely depends on the programmer and his or her use of good programming practices. With the wealth of code already freely available, Perl is and will remain a language of significant impact in computational biology for many years to come.

Perl is a "feel-good" language that does not impose a particular style on you but rather serves your way of doing things. Perl therefore promotes the three virtues of a programmer as expressed in the editorial of the Camel book (Wall *et al*., 2000): Laziness, Impatience, and Hubris (as explained at **http://c2.com/cgi/wiki?LazinessImpatienceHubris**).

References

ActiveState (2005) *ActivePerl – The industry-standard Perl distribution for Linux, Solaris and Windows* @ **http://www.activestate.com/Products/ActivePerl/**.

archwing.com (2001) *Object-Oriented Programming Overview* @ **http://www.archwing.com/technet/technet_OO.html**.

Bairoch A and Boeckmann B (1991) The SWISS-PROT protein sequence data bank. *Nucleic Acids Research*, **19**(Suppl), 2247–2249.

Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J and Wheeler DL (2003) GenBank. *Nucleic Acids Research*, **31**(1), 23–27.

Berman HM, Battistuz T, Bhat TN, Bluhm WF, Bourne PE, Burkhardt K, Feng Z, Gilliland GL, Iype L, Jain S, *et al*. (2002) The Protein Data Bank. *Acta Crystallographica. Section D, Biological Crystallography*, **58**(Pt 6 1), 899–907.

bio.cc (2005) *Bio Perl* @ **http://bio.cc/Bioperl/index_bioperl_original.html**.

bioperl.org (2005) *bioperl.org* @ **http://bioperl.org/**.

cpan.org (2005) *CPAN – Comprehensive Perl Archive Network* @ **http://cpan.org/**.

dbi.perl.org (2005) *DBI – a database interface module for Perl* @ **http://dbi.perl.org/**.

Friedl JEF (2002) *Mastering Regular Expressions*, *Second Edition*, O'Reilly.

Gutschmidt T (2005) *Perl: Strict, Warnings, and Taint* @ **http://www.developer.com/lang/perl/article.php/1478301**.

Kindler E (1988) Object oriented programming and general principles of modelling complex biological systems. *Acta Universitatis Carolinae. Medica*, **34**(3–4), 123–147.

macperl.org (2005) *MacPerl Development* @ **http://dev.macperl.org/**.

Moore GE (1965) Cramming more components onto integrated circuits. *Electronics*, **38**(8), 114–117.

perl.org (2005) *Perl6* @ **http://dev.perl.org/perl6/**.

perldoc.com (2005) *CPAN.pm* @ **http://search.cpan.org/~andk/CPAN-1.76/lib/CPAN.pm**.

perldoc.com (2005) *LWP – Library for WWW access in Perl* @ **http://search.cpan.org/~gaas/libwww-perl-5.803/lib/LWP.pm**.

perldoc.com (2005) *Math::BigFloat – Arbitrary length float math package* @ **http://search.cpan.org/~tels/Math-BigInt-1.77/lib/Math/BigFloat.pm**.

perldoc.com (2005) *Math::BigInt – Arbitrary size integer math package* @ **http://search.cpan.org/~tels/Math-BigInt-1.77/lib/Math/BigInt.pm**.

perldoc.com (2005) *perlstyle – Perl style guide* @ **http://search.cpan.org/~krishpl/pod2texi-0.1/perlstyle.pod**.

perldoc.com (2005) *perltoot – Tom's object-oriented tutorial for perl* @ **http://search.cpan.org/~nwclark/perl-5.8.6/pod/perltoot.pod**.

perldoc.com (2005) *strict – Perl pragma to restrict unsafe constructs* @ **http://search.cpan.org/~nwclark/perl-5.8.6/pod/perlmodlib.pod#Pragmatic_Modules**.

regular-expressions.info (2005) *Regular Expressions* @ **http://www.regular-expressions.info/**.

Sanner MF (2004) *Using the Python Programming Language for Bioinformatics*, pp. 5–9.

Sigrist CJ, Cerutti L, Hulo N, Gattiker A, Falquet L, Pagni M, Bairoch A and Bucher P (2002) PROSITE: a documented database using patterns and profiles as motif descriptors. *Briefings in Bioinformatic*, **3**(3), 265–274.

soaplite.com (2005) *SOAP::Lite for Perl* @ **http://www.soaplite.com/**.

sourceforge.net (2005) *Perl XML Project Home Page* @ **http://perl-xml.sourceforge.net/**.

Stein L (1996) How Perl saved the human genome project. *The Perl Journal*, **1**(2).

stein.cshl.org (2005) *CGI.pm – a Perl5 CGI Library* @ **http://stein.cshl.org/WWW/software/CGI/**.

stein.cshl.org (2005) *GD.pm – Interface to Gd Graphics Library* @ **http://stein.cshl.org/WWW/software/GD/**.

Sudhof TC, Van der Westhuyzen DR, Goldstein JL, Brown MS and Russell DW (1987) Three direct repeats and a TATA-like sequence are required for regulated expression of the human low density lipoprotein receptor gene. *The Journal of Biological Chemistry*, **262**(22), 10773–10779.

Valdes IH (1994) Advantages of object-oriented programming. *M.D. Computing*, **11**(5), 282–283.

Wall L, Christiansen T and Orwan J (2000) *Programming Perl*, *Third Edition*, O'Reilly.