Software review

The Bio* toolkits – a brief overview

Abstract

Bioinformatics research is often difficult to do with commercial software. The Open Source BioPerl, BioPython and BioJava projects provide toolkits with multiple functionality that make it easier to create customised pipelines or analysis. This review briefly compares the quirks of the underlying languages and the functionality, documentation, utility and relative advantages of the Bio counterparts, particularly from the point of view of the beginning biologist programmer.

Keywords: *Bioperl, BioPython, BioJava, bioinformatics, comparison, open source*

This article is directed to the beginning bioinformaticist or biologist thinking of learning a programming language to help with their work. If you are familiar with Perl, Python or Java, your decision is probably already made, based on your current preferred language. However, if you have not already passed that developmental checkpoint, this overview may help you decide which one to pursue.

Bioinformatics is a young science and while there are a number of commercial applications aimed at researchers in biology, these are often not sufficient for the level of data analysis required in bioinformatics research. It was partly the frustration with commercial suites that drove the founding of the Bio* groups. (The Bio* name uses the regular expression * operator to denote all characters, shorthand for BioPerl, BioJava, BioPython, etc.)

The Bio* group (formally the Open Bioinformatics Foundation[1]) was formed by a group of self-described Perl hackers who got together in 1995 to pool resources for writing bioinformatics software. The group saw that there was much fine-grained functionality that was extremely useful and if the program source code could be shared, it could be easily worked into functional programs. The same idea gave birth to the BioPython and BioJava groups in 1999 and the BioCORBA and BioDAS have been added since. The BioRuby,[2] BioLisp[3] and Bioinformatics.org[4] groups share a similar vision and are worth investigating for useful perspective and resources, but are officially unaffiliated.

Before you dive into a long-term commitment to a language, and its Bioderivative, it is useful to see how it is perceived by the various stakeholders. Table 1 shows a quick and simple survey based on scanning the Usenet newsgroups, Google and Amazon.com as a crude measure of languages' popularity and support.

Over the last 10 month period, the membership of each group has grown by about 50 per cent. By far the largest number of posts is in the BioPerl group; the BioJava group is gaining steam, and the BioPython group tends to be quite a bit lower, reflecting its lower membership (see Table 2).

All the Bio* projects described here use the eponymous base language and endow it with 'Bio' features via additional modules or libraries. A brief overview of the base languages follows.

Perl, Python and Java are all interpreted, which means that they are slower than a compiled language such as

Table 1: Popularity of base languages and Bioderivatives on 6.29.02 on Amazon (number of titles found at Amazon.com based on search for '[Language] programming', Usenet news (total number of posts in all of comp.lang. and subgroups), Google (number of hits based on single word query for [Language] or Bio derivative)

Language	Amazon	Usenet	Google	Google	Bio*
Perl	292	13,279	9,440,000	406,000	BioPerl
Java	1,161	33,296	23,300,000	92,000	BioJava
Python	39	11,880	3,590,000	44,600	BioPython
Lisp	126	5,190	1,630,000	190	BioLisp
Vbasic	983	?	2,370,000	–	–
C++	1,188	17,313	5,730,000	–	–

Table 2: Traffic on Bio* lists. The two dates indicate the number of subscribers to each of the lists on that date. 'Total Posts' is the aggregate total number of posts to the lists since the 'Since' date. Posters is a crude estimate of the number of people posting to the list

List	28th August 2001	21st June 2002	Total posts	Posters	Since
BioPerl-l	643	922	5,938	989	August 1996
BioJava-l	365	575	3,037	531	September 1999
BioPython	168	242	1,024	187	September 1999

C or C++ (typically three-quarters to one-tenth as fast, depending on the type of logic being implemented), but they are hardly slouches. If speed is an issue, all of them can be made to link to compiled libraries via the Java Native Interface, Python's C interface and Perl's XS routines. The latter two can also make use of SWIG,[5] a more portable way of interfacing polyglot code. Two examples of how an interpreted language can be used in high-performance computing are PyMol,[6] a molecular visualisation and modelling application, and the Perl Data Language[7] which uses libraries of compiled code and an object oriented (OO) approach to allow very fast computation on *N*-dimensional arrays.

In giving up the speed of compiled code, all these languages are considerably easier to program with. Mercifully, none of them requires that you manually track and manipulate memory allocation and none requires (or even permits) use of the much-hated memory pointer. As well, many programming features or niceties that in C you have to program yourself, are provided for you, such as associative arrays, numeric interconversions, easy input/output handling, string manipulations and large numbers of oft-used programming expressions.

All have very good support for network functionality and all support regular expression (regex) pattern matching[8] although regex support is integrated throughout Perl's structure but must be explicitly requested in Java and Python. All provide extensive libraries to connect to many relational databases. Perl's database independent module allows nearly identical access to most relational database management systems (RDBMSs). Python has a similar approach, using database-specific drivers that present an identical application programming interface (API) to the programmer, and while it is less well developed than Perl's, it supports most of the popular commercial and open source RDBMSs. The Java DataBase Connectivity (JDBC) is now a standard part of the language that provides nearly identical functionality and database

support to the ODBC drivers that Windows uses to provide RDBMS connectivity.

All three languages are multiplatform – they run similarly on the most current versions of Unix, Linux, Windows and the Mac. In addition, Perl and Python qualify for the open source definition – they are freely available in source code and while hundreds contribute to their continued development, a single person wrote the first few implementations and remains the lead technical Godfather of the project. Java, while made freely available, is owned and defined by Sun Microsystems, whose technical committees decide what goes into Java and when.

Java and Python have good support for creating graphical user interfaces (GUIs). Java uses its native Swing libraries; Python uses a variety of multiplatform widgets sets including Tk[9] (bundled with Python), wxWindows[10] and Qt.[11] While Perl can be used to create GUIs (most easily with Tk), it is a failing that is not nearly as well supported as it is in Java or Python.

Perl does, however, have a non-trivial advantage over Python and Java in that it can be automatically upgraded and enhanced using the CPAN module (for Comprehensive Perl Archive Network), included in the default installation. This allows a user to request an additional module to be retrieved, checked for dependencies, have those dependencies resolved automatically, and the entire tree of dependencies automatically downloaded, tested and installed all in a single line of code. All this is available without requiring the user know where the files are archived. For example, to install the BioPerl module and all of its documentation (once the CPAN module is easily configured), this is all you need to do:

```
$ perl -MCPAN -e 'install "Bio::Perl"'
```

The BioPerl installation will prompt you about additional Perl libraries it needs for some methods; more detail is available.[12] Python and Java use the older 'go find it, download it, install it' approach of most other software installation and since this almost always requires dependency tracking, the CPAN feature is a significant time and frustration-saver.

One of the deepest divides among the languages is that while Python and Java were designed from the ground up as pure OO languages, Perl has gradually added this functionality over time. The result is something of a dancing camel – it can be quite remarkable when it works, but it is difficult to describe to others. Combined with Perl's 'There's More Than One Way To Do It' (TMTOWTDI) philosophy and syntax, it can be fairly challenging to deconstruct another's OO Perl code. Both Java and Python have considerably more structured syntax and are therefore more easily understood, although both have quirks of their own; for example, Python uses whitespace (not braces) to segment logic blocks.

Java, like Python, is a pure OO system and has very wide library coverage. Unlike Python, Java often seems to be self-consciously OO and much more a formal programming language than either Perl or Python. As such, it is being used as a teaching language at many schools, much more so than Perl. It has also been embraced by a variety of companies seeking to break Microsoft's stranglehold on the computing public. In response, Microsoft has recently indicated that its forced marriage with Java will end in 2004 and has introduced the .NET and C# projects as a way of superseding Java, causing some consternation in the ranks of developers. However, since the Javaphilic companies range in size from Sun to IBM to Borland and others, there seems to be no immediate worry that Java will disappear. In the midst of this feuding, some astonishing programmer resources have been made freely available to tempt developers to use Java. These include integrated development environments, debuggers, profilers, extra libraries and just-in-time compilers to speed Java's often-sluggish performance.

Java has several implementations, from Sun, IBM, Microsoft and others. This results in competition to make the Java compiler and tools better, but it also has the side effect of programs sometimes working with only one of the several Java implementations.

In my opinion, Python is easier than either Perl or Java for a novice to learn, encourages a cleaner programming style, and certainly makes it easier to follow others' code. Python also has a large and growing base of additional software modules, including some that fall into the 'Killer app' category such as Zope,[13] a combination web-server, object database, content management system and application server with which BioPython has started integration efforts. Python also has the reputation that it is easier to write more compact code, which itself leads to fewer editing mistakes and thus more maintainable code. An additional advantage of Python is that it shares considerable API similarity with C++, allowing code to be prototyped in Python and then easily replaced with C++ as performance demands. This clarity of code, the ability to use multiple cross-platform GUI toolkits, and the availability of several Rapid Application Development tools for Python makes it a compelling language to wrap command-line tools with a GUI.

BioPerl comes as a 2.5 MB gzipped file and inherits the chameleon-like charm of its parent. BioPerl's tutorial is certainly the most complete, with different sections illustrating almost every feature in the toolkit. BioPerl is the oldest and most downloaded of the Bio* distributions and for some good reasons. It is certainly the most mature, has the most useful features and has the largest development community. The documentation of BioPerl's structure (60 levels, ~400 modules) is handled in an extremely well-designed layout that makes it easy to peruse the functionality of the package. The documentation for each module is unusually complete and contains short descriptions of the module, its dependencies and inheritance links, description and code examples for each method. It includes all the things that you would expect in a commercial package, and a few things that you might not – the source code and the e-mail address of the maintainer or author. In addition, there is already one text that introduces Perl and BioPerl to the novice bioinformaticist[14] and a more advanced one by two of BioPerl's principal contributors is in process.

The modules address a very wide array of functionality, including pure bioinformatics structures such handling and indexing most popular bio-specific database and flatfile formats (including all the Readseq-supported[15] formats, as well as BSML[16] and GAME XML[17]), auto-generation of bio-related graphics for web pages, classes and methods for describing and manipulating biological sequences, annotations, trees, alignments and maps. It has an extensive collection of modules for initiating several search methods and the parsing and manipulating the results of such searches. It also provides a number of analytical primitives such as pattern matching and related tools (motif finding, restriction enzyme mapping), and wrappers or handler routines for the results of other popular tools and techniques (BLAST, FastA, HMMR, Sim4, GeneID, Genemark and others). It also has the best module support for 3D and structural information.

It is hard not to recommend BioPerl. It certainly has the largest user base and functionality currently; it is easily upgraded and maintained and Perl is a user-driven and robust language. The documentation is exemplary and BioPerl is in active use and development at many of the large genome centres. Additionally, there is an online course,[18] so the beginner can be brought up to speed on the basics and get a quick BioPerl overview. Perl's greatest fault (and a great attraction to some) seems to be its TMTOWTDI philosophy which can lead to inscrutable syntax and its somewhat non-standard object system, although the

BioPerl project and code have reined in this style diversity.

The BioPython package (~1.7 MB gzipped) contains about 30 classes with approximately the same breadth as its siblings. What it covers is easy and straightforward to follow, and provides a good base for further development. Installation of the complete package along with all the dependencies involves separately downloading and installing components, some of which required manual intervention to convince them to install correctly. The included BioPython tutorial is an excellent overview not only into the BioPython structure but also to OO programming in general. I recommend it to anyone planning to use *any* of the Bio* languages or even learning to program.

BioPython includes methods for biological and regular expression pattern matching, interacting with local and remote BLAST resources and local FastA and ClustalW, interacting with the BioSQL database schema, searching PubMed, parsing a large number of database formats, and provides code fragments and entire scripts showing how to do this. Personally, even though I have more experience with Perl than with Python, these code examples are considerably clearer than those from BioPerl, although the Perl examples are quite complete. For example, extracting a FastA query sequence from a file, preparing it and submitting it to NCBI BLAST as well as parsing and printing the returned results takes about 30 lines of uncommented but easily understood code. I heartily recommend the BioPython online course[19] prepared by the same authors as BioPerl one mentioned above.

One of the areas where BioPython shines is in the area of being able to parse the many formats in bioinformatics. Rather than having to write a completely new parser for each new format, BioPython relies on the Scanner/Consumer methods that are the core of the Martel regular expression parsing engine,[20] which allows the user to make use of many current formats and easily define others using a SAX-like[21] approach. In this area, BioPython seems to outclass even the more regex-oriented BioPerl.

BioPython also allows easier creation and manipulation of objects than does BioPerl, not too surprising as Python was designed as an OO language and Perl had Objects grafted on later in life.

However, reflecting its youth and small subscription base, the depth of coverage and its documentation are not as well developed. This is unfortunate, as Python seems the ideal language to execute a distributed project such as this. It enforces a readable and succinct syntax, plays well with a number of other languages, and comes with good support for building GUIs. It almost seems that many programmers refuse to use it because it sounds too good to be true.

The BioJava code (5.6 MB gzipped source; 1.5 MB jar) is described in detail via the industry-standard JavaDoc API description format (~40 packages, 720 classes, from AbstractAlignmentStyler to ZiggyFeatureRenderer). Unlike both BioP*s, however, while the APIs are described reasonably well in the JavaDocs, details and examples about how the classes are actually used are sparse. This is made more problematic as I was not able to find an external online tutorial that described it in the same detail as for BioPython and BioPerl, although there are numerous tutorials for the base language of course. While the included BioJava tutorial contains brief descriptions of the way BioJava handles Symbols, Sequences, Features, I/O and interprocess coordination, it is not nearly as beginner-friendly as the BioP*s. In addition, while the introductory text provides some examples to illustrate its points, some of the sample code is based not on biological examples, but on a roulette wheel simulation. Until this lack of introductory material is addressed, BioJava will probably remain the toolkit of choice only of well-established Java

programmers, as it is difficult to determine what methods do what simply from browsing the JavaDocs.

BioJava, like its parent, uses Unicode for its basic string character. In Java, this can be quite useful, but in the strings of molecular biology, even one byte (which can code 256 characters) is generally overkill and BioJava by default extends the 2 byte Unicode character to 4 bytes for its object system of referring to resides, leading to considerable bloat for handling large sequences. In comparison, Perl and Python use single byte representations for residues. BioJava does provide many more biology-specific GUI elements than do its Biobrethren, including methods for rendering features and annotations onto a canvas, graphics specifically for multiple sequence comparisons, and pane decorations such as length tics and crosshairs so if designing custom graphical applications is a consideration, BioJava is certainly worth evaluating. BioJava also contains methods supporting some relatively esoteric numerical approaches such as support vector machines[22] (used in clustering) and suffix trees[23] (used in fast pattern searching), as well as the more common hidden Markov models[19] and direct pattern searching.

BioJava also shares its parent's affection for XML, including classes for handling various XML formats such as the aforementioned GAME XML and a large number of methods supporting AGAVE (now in limbo with Doubletwist's demise).

Summary

So the upshot is this: for small programs (<500 lines) that will be used only by yourself, it is hard to beat Perl and BioPerl. These constraints probably cover the needs of 90 per cent of personal bioinformatics programming requirements. For beginners, and for writing larger programs in the Bio domain, especially those to be shared and supported by others, Python's clarity and brevity make it very attractive. For those who might be leaning towards a career in bioinformatics and who want to learn only one language, Java has the widest general programming support, very good support in the Bio domain with BioJava, and is now the *de facto* language of business (the new COBOL, for better or worse). Note that a well-rounded bioinformaticist would be expected to know all of these languages and be able to choose the best for a particular effort.

Acknowledgments

Many thanks go to Jason Stewart for his Perl advice, Andrew Dalke for his Python advocacy and Don Gilbert for his comments on this manuscript.

Harry Mangalam
tacg Informatics,
1 Whistler Ct,
Irvine, CA 92612,
USA
Tel: +1 949 856 2847
E-mail: hjm@tacgi.com

References

1. URL: http://open-bio.org (links to the BioPerl, BioPython, BioJava, BioCORBA, and BioDAS pages).
2. URL: http://www.bioruby.org
3. URL: http://www.biolisp.org
4. URL: http://www.bioinformatics.org
5. URL: http://www.swig.org
6. Delano, W. (2000), PyMol URL: http://pymol.sourceforge.net. This open source application uses Python to provide the GUI and glue code with compiled C and OpenGL libraries to provide astonishing real-time graphics performance.
7. URL: http://pdl.perl.org
8. Friedl, J. E. F. (1997), 'Mastering Regular Expressions', O'Reilly and Associates, Sebastopol, CA. See also URL: http://etext.lib.virginia.edu/helpsheets/regex.html
9. URL: http://www.pythonware.com/library/an-introduction-to-tkinter.htm
10. URL: http://www.wxwindows.org
11. URL: http://www.trolltech.com
12. URL: http://bioperl.org/Core/external.shtml
13. URL: http://www.zope.org
14. Tisdall, J. D. (2001), 'Beginning Perl for Bioinformatics', O'Reilly & Associates, Sebastopol, CA.

15. Gilbert, D. G. (1990), 'ReadSeq program'. URL: http://iubio.bio.indiana.edu/soft/molbio/readseq/

16. Bioinformatic Sequence Markup Language, Labbook, Inc., URL: http://www.labbook.com/products/xmlbsml.asp

17. Genome Annotation Markup Elements, URL: http://www.bioxml.org/Projects/game/game0.1.html4

18. Letondal, C. and Schuerer, K. (2002), 'BioPerl Course', Pasteur Institute, Paris. URL: http://www.pasteur.fr/recherche/unites/sis/formation/bioperl/

19. Schuerer, K. and Letondal, C. (2002), 'Python Course in Bioinformatics', Pasteur Institute, Paris. URL: http://www.pasteur.fr/recherche/unites/sis/formation/python/

20. Dalke, A. P. (2001), Dalke Scientific. URL: http://www.dalkescientific.com/Martel/

21. SAX stands for *Simple API for XML* and follows the event-based model for parsing XML (as opposed to the tree-based model, which creates an in-memory hierarchy of XML objects). Like SAX, Martel follows an event-based approach to allow completed stanzas to be 'consumed' immediately. SAX is described at the URL: http://www.saxproject.org

22. Sturn, A., Quackenbush, J. and Trajanoski, Z. (2002), 'Genesis: cluster analysis of microarray data', *Bioinformatics*, Vol. 18(1), pp. 207–208. See also URL: http://www.support-vector.net

23. Gusfield, D. (1997), 'Algorithms on Strings, Trees, and Sequences', Cambridge University Press, Cambridge.