From C to Java, Scientific Data Analysis with Java, Jacl and Swank

Bruce A. Johnson

One Moon Scientific, Inc.

bruce@onemoonscientific.com

www.onemoonscientific.com

I. Introduction

Nuclear Magnetic Resonance (NMR) spectroscopy is an analytical technique that can be used to derive information about the structures and motions of molecules. This information can be used to determine the chemical nature of unknown molecules, determine the three dimensional structures of large biological molecules such as proteins and DNA, and visualize the internal structure of living organisms (MRI imaging). The author developed the program, NMRView, to facilitate the visualization and analysis of bio-molecular NMR data [1,2]. Over one hundred scientific papers are published each year that cite the use of NMRView

The NMR phenomenon derives from the fact that some nuclei have a quantum mechanical property of "spin angular momentum". In a magnetic field the different possible spin states have different energy levels and the populations of nuclei in the different states can be perturbed by the addition of energy at the frequency of radio waves. Hydrogen atoms, for example, resonate at about 600 MHz in a typical NMR magnet.

The exact frequency at which the nucleus of a given atom resonates, and the time it takes for the population of excited atoms to relax back to their equilibrium state, is dependent on the chemical environment in which the atom exists. It is the measurement of these differences in frequency and relaxation time that give rise to the myriad applications of NMR spectroscopy.

A common biological application of NMR involves the determination of the 3D structure of a protein [3] (Fig. 1). In this application the protein is expressed in a microorganism whose sole nutritional source of carbon is from molecules (typically glucose) in which the carbon atoms are all atoms of the NMR active isotope ^{13}C. Similarly the nitrogen source (typically ammonia) consists of molecules in which the nitrogen atoms are of the ^{15}N isotope.

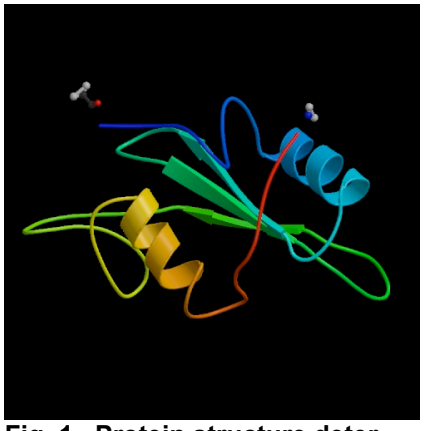
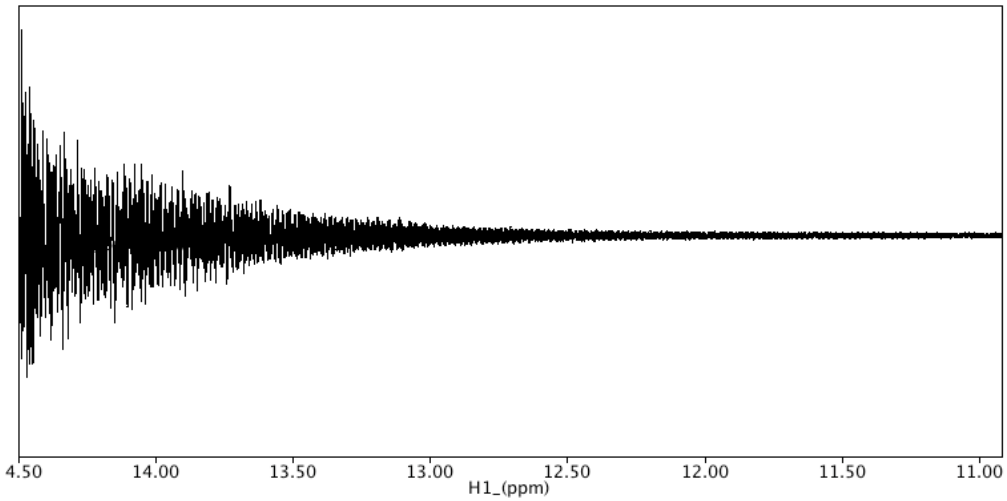
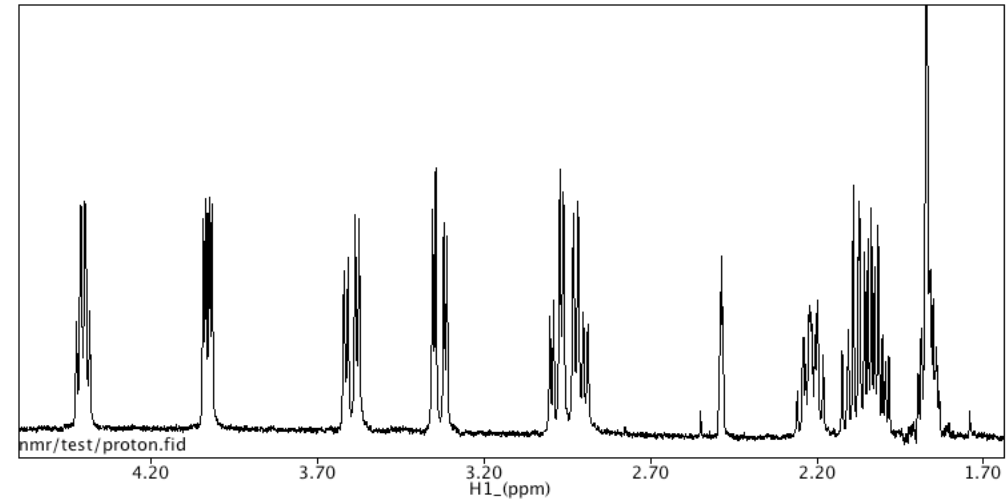


Fig. 1. Protein structure determined by NMR spectrosc

Signals from the ^{1}H, ^{13}C and ^{15}N atoms in the protein are analyzed to determine a set of constraints on the distances between the atoms. Various computational



A



B

Fig 2. NMRView spectrum widget showing time domain and frequency domain signals.

techniques are then used to generate 3D models consistent with those constraints. The data involved in this process may be derived from 10 or more different types of NMR experiments collected over a period of weeks. The individual data sets may range in size from megabytes to gigabytes in size.

Deriving useful information from the large amounts of NMR data involves processes of both computational analysis and human visualization. NMRView was written to facilitate this analysis. It allows the user to both visually correlate information in different datasets as well as apply computational techniques to extract and correlate information about the frequencies and intensities of the NMR data derived from specific atoms in the molecular structure.

NMRView incorporates Tcl (Tool Command Language) and the entire GUI is implemented with the Tk GUI toolkit. A custom Tk widget was implemented to produce interactive one-dimensional (Fig. 2) and two-dimensional (Fig. 3) data plots that represent any orthogonal slice through the multidimensional NMR datasets.

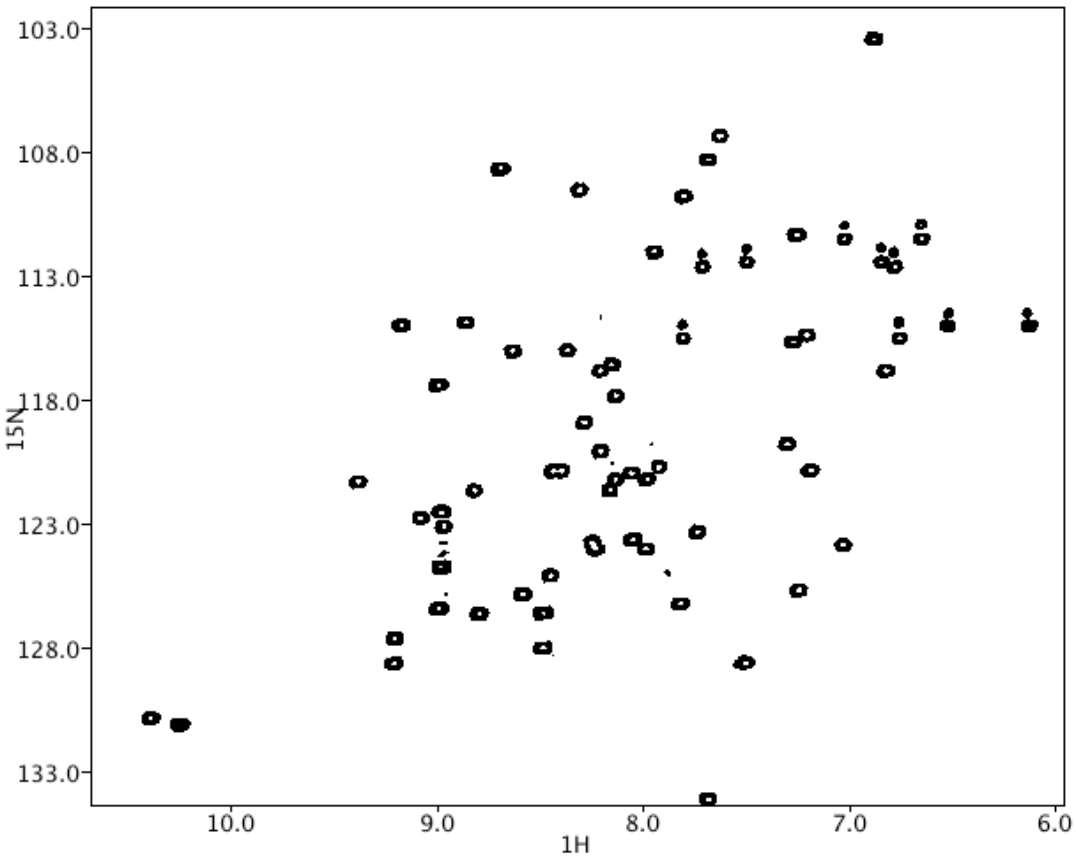


Fig. 3. NMR Spectrum widget showing two-dimensional contour plot of spectrum.

A variety of analysis routines are written in C, and exposed to the user as Tcl commands (Table 1.) These commands give low level access to the NMR data and implement a variety of algorithms such as clustering and pattern recognition.

Table 1. Some of the NMRView commands.

Command Name	Function
spectrum	Create a widget for rendering vector and contour plots of spectra.
nv_dataset	Open, read, write and analyze NMR data files.
nv_peak	Identify and analyze peaks (local maxima) in spectra.
vecmat	Create and carry out mathematical operations on vectors and matrices.

A large number of Tcl scripts, both written by the author and contributed by the user community, are used in the operation of NMRView.

Tcl scripting in NMRView is one of the key features that led to its great popularity in the bio-molecular NMR community. Scripting has been particularly important in the program because the field of NMR continues to be under rapid development. Using scripts to implement new analysis methods allows for the rapid progress necessary to keep up with scientific advances.

Despite it's popularity, it was clear that there was room for improvement of NMRView. One issue related to the effort to build NMRView for multiple platforms. NMRView was distributed as binary executables and so it was necessary to port and

compile the code on a variety of computers. Every release required a large commitment of time and access to the necessary computer platforms.

Porting the code to multiple platforms had one significant advantage: invariably, it led to the discovery of bugs that existed, but had not yet manifested themselves, on the other operating systems. This was somewhat unsettling, however, in that it was unclear how many bugs remained to be found. Aside from problems the occasional bugs caused, time that could be spent on algorithmic and scientific analysis was being devoted to preventing and fixing "trivial" programming errors.

Finally, the code as originally written in C was not readily amenable to the use of newer developments in software engineering. There is an almost natural representation of the scientific concepts involved in NMR in an object-oriented representation. Taking advantage of this parallelism was not readily possible within the C programming language.

Furthermore, useful new third-party libraries were being developed in programming languages other than C. These libraries could not be readily incorporated into NMRViewJ.

Adopting the Java programming language was seen as a possible means to advance the state of the NMRView application. Using Java would allow the development of the code on only a single platform, yet allow it to execute on more operating systems than the C version of NMRView. Most bugs in NMRViewC were related to misuse of pointers and overrunning array bounds. The fact that these errors are not possible in Java means that more time and effort can be spent on making NMRView more valuable to end-users. Finally, adoption of Java would allow the use of newer techniques of software engineering and the incorporation of newer third party libraries.

The use of Tcl/Tk was so important to the success of NMRView as a useful tool for scientists that a switch to Java could not be considered without a means to continue with these scripting tools. Using TclBlend, which allows one to incorporate a Java virtual machine into the Tcl environment, was rejected, as this option was inconsistent with the desire to create a platform independent solution. Furthermore, the interface between the C and Java portions of the code was both a potential source of instability and created an artificial barrier within the code.

The solution chosen was to implement the entire application in Java, using Jacl to execute Tcl scripts within the code. Jacl is an implementation of the Tcl language that is written entirely in Java [4]. While Jacl could substitute for Tcl, there was no analogous Java solution to replace Tk. After some experimentation, I decided that development of a Java implementation of Tk would be feasible.

Two key factors allowed for the feasibility of developing Swank ("Tk in Java") in a reasonable period of time [5]. Swing, the Java user interface toolkit, provided a rich variety of widgets with similar functionality to Tk widgets [6]. Using the Swing widgets meant that the behavior of Swank would not be as similar to Tk as it would if the Swank widgets were developed from lower level Java components. On the other hand, adopting Swing meant that a great deal of coding work could be skipped. Furthermore,

Jacl introspection command:

```
java::info methods javax.swing.JButton
```

Command result:

```
…
{setForeground java.awt.Color}
{setHorizontalAlignment int}
{setHorizontalTextPosition int}
{setIcon javax.swing.Icon}
{setIconTextGap int}
{setLabel java.lang.String}
{setLocale java.util.Locale}
{setName java.lang.String}
{setPressedIcon javax.swing.Icon}
{setRequestFocusEnabled boolean}
{setRolloverEnabled boolean}
{setRolloverIcon javax.swing.Icon}
…
```

Java code generated:

```
} else if (argv[i].toString().equals(
"-icontextgap")) {
                swkjbut-
ton.setIconTextGap(TclInteger.get(
interp, argv[i + 1]));
```

Figure 4: Jacl based introspection.

```
This Jacl code:
# -selectMode
set vWidgets "JList"

if {[lsearch $vWidgets $widget ] >= 0} {
   set specialGets [concat $specialGets {
      {
       setSelectionMode tkSelectMode SelectionMode –selectmode
      }
   }]
}

ultimately results in this Java code:
…
} else if (argv[i].toString().equals("-selectmode")) {
      swkjlist.setSelectionMode(SwankUtil.getTkSelectMode(interp, argv[i + 1]));
} else if …
```

Fig 5: Jacl code to generate Tk specific configuration options

using the Swing widgets provides a richer set of behaviors than the original Tk widgets.

The second key factor was the introspection capabilities of the Jacl language. Much of the code that forms the basis of Swank is generated by Jacl scripts that determine the fields and methods of each Swing component and then automatically produce Java code that provides a Tk-like interface to the components (Fig. 4). This generates a large number of configuration options for each widget. Some of these map coincidentally to the names and functions of Tk configuration options. In other cases, Jacl code is used to specifically generate Java code for Tk options. In some of these cases it is only necessary to generate code that parses the appropriate Tk option and maps it to an existing Java Swing method. In other cases specific Java code is written to enable the correct action in response to the specified option (Fig 5). This Java code is inserted in the generated Java file.

The result of using the "java::info methods" command, as illustrated in Fig. 4, is the method name and argument types that each method requires. Integer, double, and String arguments are processed with standard Jacl API commands. Other arguments are processed by hand written Java code specific to each argument type (Fig. 6.)

At present most Tk widgets are implemented in Swank and have similar behavior and implement most of the Tk configuration options. The above introspection protocol also generates many additional configuration options for each of the widgets.

Perhaps the two most significant differences with Tk are the lack of the option database and the idiosyncratic behavior of the scrollbars. The above protocol for generating the Java code that implements the Swank widgets relies on using existing code in the Swing widgets as much as possible. This design did not lend itself to a paradigm that facilitates the use of the option database. Still, this is a significant deficiency in Swank that should be rectified in future versions.

```
public static Locale getLocale(Interp interp, TclObject tclObject)
        throws TclException {
        TclObject[] argv = TclList.getElements(interp, tclObject);

        if (argv.length == 2) {
            return (new Locale(argv[0].toString(), argv[1].toString()));
        } else if (argv.length == 3) {
            return (new Locale(argv[0].toString(), argv[1].toString(),
                argv[2].toString()));
        } else {
            return (Locale.US);
        }
    }
```

Figure 6. Java code to get a Java Locale object from the command arguments.

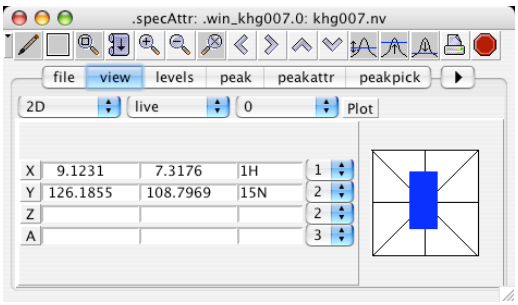


Figure 7. The NMRView spectrum control panel. Note the use of the Swank "jtabbedpane" widget.

Again, reliance on Swing code has made it difficult to get the correct Tk-like interaction between scrollbars and scrollable widgets. As an alternative, Swank implements a "jscrollpane" widget. This container can appropriately manage scrollable widgets. Horizontal and vertical scrollbars are automatically displayed as needed.

With Jacl and Swank it was possible to proceed with the development of NMRViewJ. Figure 7 illustrates one of the primary control panels of NMRView. This interface panel uses many of the widgets, including the canvas. The Swank tabbed pane allows the user to rapidly switch between different attribute groups. Figure 8 shows another NMRViewJ interface panel that provides a graphical interface to non-linear regression curve fitting routines.

The resulting application met most all the desired goals. It is cross platform: the same Java byte-code files run on Mac OS X, Windows, Linux and various varieties of Unix. It is robust: typical errors of NMRViewC do not occur, and when errors do occur they do not generally crash the program. Instead of a core dump, errors generally result in throwing a Java exception, manifested by an error message in the console.

NMRViewJ is a multi-threaded application: computationally intense processes run in separate threads, and take advantage of multiple processors if they are available. While this would have been possible to do in

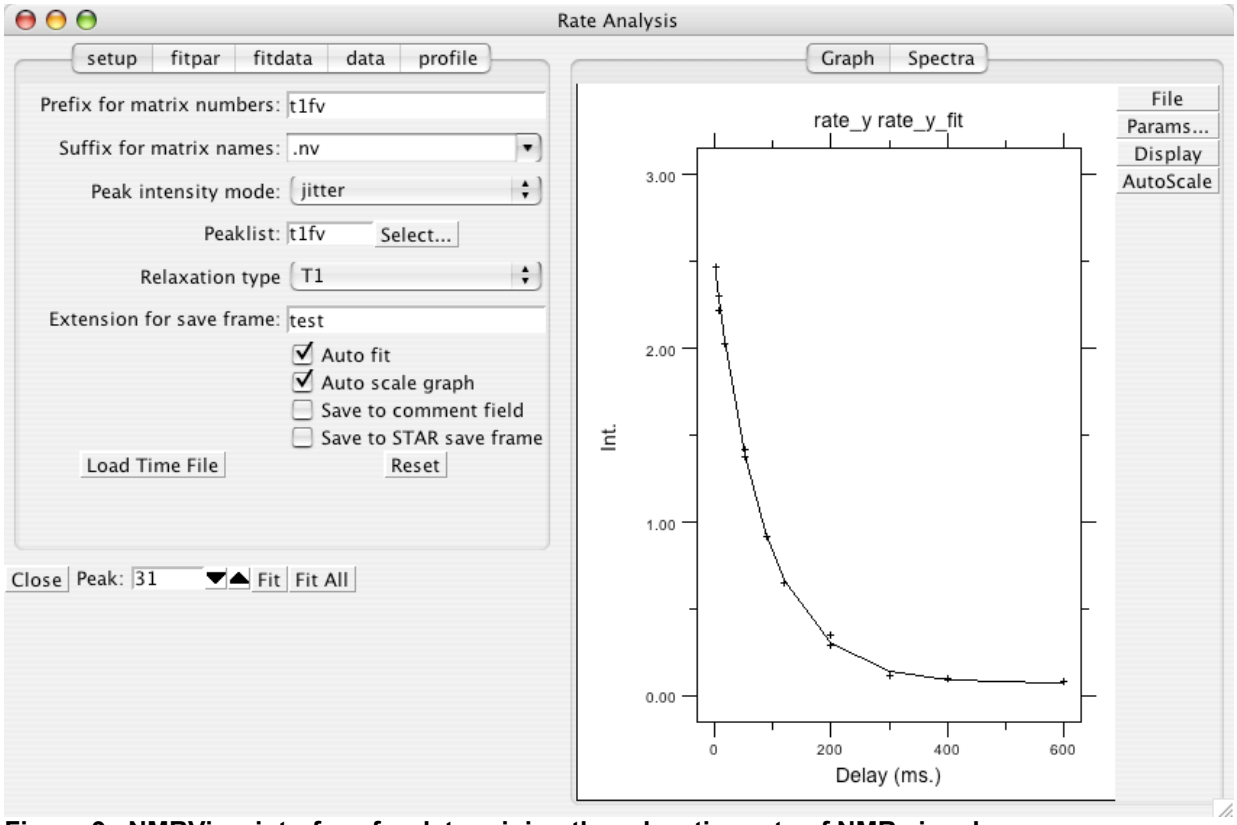


Figure 9. NMRView interface for determining the relaxation rate of NMR signals.

```
vecmat resize vec1 2048          # Create vector object named “vec1”
vread $fileHandle 1 vec1         # Read the first vector in file into the vector “vec1”
vecmat sb vec1    -offset 0.5    # Multiply vector by a cosine shaped window
                                 #         (sine shifted by 0.5*π)
vecmat zf vec1 1                 # Zero-fill vector (extend with zeros to 1x the length)
vecmat ft vec1                   # Fourier transform vector
vecmat real vec1                 # Discard the imaginary portion of the complex vector
```

Figure 10. Data processing script.

NMRViewC, the Java Threading API made it relatively straightforward to implement multi-threading in a cross-platform manner.

The use of Jacl and Swank in NMRViewJ has meant that it is largely backwards compatible with NMRViewC: most Tcl scripts are the same in the two versions. On the other hand, as development continues on new versions of NMRViewJ new features of Swank are being used to improve the user interface. Widgets such as the jtabbedpane, jtable, and jtree are being used to add more capabilities to the NMRViewJ GUI. A “print” command that calls methods of the Java Print API allows the user to print any of the widgets.

A significant design goal was to be able to take advantage of third party, open-source packages to be able to rapidly extend the feature set of NMRViewJ. Java and Jacl provide many advantages in this respect. The cross-platform nature of Java allows libraries to be incorporated without compilation on each platform. The Jacl introspection capability makes it possible to incorporate some libraries without writing any Java code.

A particular open source library that has been very useful is Colt, a high performance Java library for scientific and technical computing [7]. Colt provides storage and mathematical algorithms for one, two and three dimensional matrices. The algorithms provided with Colt were supplemented with those necessary for NMR data processing and these were exposed at the scripting level with the “vecmat” command.

The processing necessary to import and then transform the data illustrated in Fig. 2A to that in Fig. 2B was implemented with the following script shown in Figure 10.

A distinct advantage of working in the Jacl environment is that Java libraries such as Colt can be used without writing a single line of Java code. This is true even for libraries (like Colt) that were not explicitly written to have scripting interfaces. Jacl’s “java::new” command can be used to create a new Java object that corresponds to the Java class used as an argument to the command. The java::new command returns a handle to the underlying Java object. The handle can then be used to invoke the various public methods of the class.

Interacting with the Colt library using the Jacl introspection capability is illustrated in Fig. 11. In this example a 1D matrix (a vector) is created and the elements set to a value. This same protocol can be used to instantiate an object of any of the library’s classes that have a public constructor. Similarly, the handle to the object can be

```
% set dm1dA [java::new {cern.colt.matrix.impl.DenseDoubleMatrix1D int} 32]
java0x184
% set n [$dm1dA size]
32
% for {set I 0} {$I <  $n} {incr I} {$dm1dA set $I 0.1}
%$dm1dA zSum
3.2
%
```

Figure 11. Using the Colt library through Jacl’s introspection capability.

used to invoke any of the objects public methods.

A significant concern in developing NMRViewJ was whether there would be a significant drop in performance in switching from C to Java. So far this has not been a problem. Computationally intense processes written in Java have high performance: the Java code often runs nearly as fast or even faster than the C code. Little work has been done with NMRViewJ to optimize the code for performance. I expect that when careful code profiling and optimization is applied to code areas that are slower that performance of the Java code will be brought close to that of the C code.

The primary remaining issue is the poor performance of some computational routines written in Tcl scripts. A simple benchmark is illustrated in Fig. 12, and shows that Jacl code might run 25 times slower than Tcl code. This poor performance presumably is a consequence of the lack of the Tcl byte-code compiler in Jacl. The Jacl interpreter does not even cache the results of parsing procedures.

Despite the poor performance of Jacl code, which could be overcome with further development of Jacl, the experience of porting NMRView from C to Java has convinced the author that there are compelling reasons for using Java, Jacl and Swank to develop a wide variety of applications.

```
proc bench {} {
    set x 0.0
    for {set i 0} {$i < 5000} {incr i} {
        set x [expr {$i*3.0+$x}]
    }
    return $x
}

tclsh   ~20000 microseconds/iteration
jaclsh ~500000 microseconds/iteration

jaclsh ~25x times slower
```

Figure 12. Simple Tcl script benchmark.

[1] B. A. Johnson and R. A. Blevins, NMRView: a Computer Program for the Visualization and Analysis of NMR Data, J. Biomol. NMR 4,603-614. 1994

[2] B. A. Johnson, Using NMRView to Visualize and Analyze the NMR Spectra of Macromolecules, in *Protein NMR Techniques*, 2nd Edition pp. 313-352, Human Press, 2004.

[3] J. Cavanagh, W. J. Fairbrother, N. J. Skelton, and A. G. Palmer, Protein NMR Spectroscopy: Principles and Practice, Academic Press, 1995.

[4] http://tcljava.sourceforge.net/docs/website/index.html

[5] http://www.onemoonscientific.com/swank/index.html

[6] J. Elliott, R. Eckstein, M. Loy, D. Wood, B. Cole, *Java Swing* 2nd Edition, O'Reilly, 2002.

[7] W. Hoschek, Colt: Open Source Libraries for High Performance Scientific and Technical Computing in Java. *http://dsd.lbl.gov/~hoschek/colt*