The Java and C++ platforms for scientific computing

Dave Hale
Center for Wave Phenomena, Colorado School of Mines, Golden CO 80401, USA

ABSTRACT
Scientific computing is evolving beyond array processing to be more interactive, more graphical, more parallel, and less structured than it was when most scientific software was written. Languages such as C++ and Java, designed for object-oriented and generic programming, enhance the development of new data structures and algorithms that are part of this evolution.
But these languages complicate mixed-language programming. It is today more difficult than ever before to use software components developed for one language with another. The languages Java and C++ and their respective libraries and tools represent distinct software platforms that are incompatible. We must choose one or the other, and I favor the Java platform.

Key words: software development, scientific computing

1 INTRODUCTION

When developing scientific software, what programming languages should we use? What software libraries should we build upon? What operating systems should we support? Our answers to such questions define the platform of our software systems.

When extending an existing software system, these questions have already been answered, perhaps years ago when choices were fewer and simpler. Then, perhaps even our definition of scientific computing was simpler. Did that definition include multi-threaded computing, graph algorithms, or interactive graphics?

For future software systems, we must choose from an increasingly bewildering variety of platforms. We must consider multiple languages, multiple operating systems, and scientific computing beyond simple array processing.

Programming languages are only one part of our choice. For example, *FORTRAN* is the original programming language for scientific computing, and is well-suited to array processing. But FORTRAN as a platform falls short in other contexts, such as portable libraries for interactive graphics. While any platform for scientific computing today must be able to exploit existing FORTRAN software libraries, FORTRAN should not be the basis for those platforms.

1.1 Why *not* MATLAB?

A better example of a platform for scientific computing is *MATLAB*. As a programming language, MATLAB like FORTRAN is best-suited to array processing. But MATLAB is much more than a programming language. It is a complete software research and development environment, with built-in program editors and extensive libraries for numerical computing and data visualization. An ever increasing number of students of scientific computing today develop software almost exclusively in MATLAB.

This trend is unfortunate, for at least two reasons. First, MATLAB is a poor choice in contexts of scientific computing requiring complex data structures, such as graph structures or unstructured grids. It is also a poor choice for research requiring interactive graphics. Students that know only MATLAB will avoid important research problems for which array processing with MATLAB is a poor solution.

When I teach *Data Structures in C++* at the Colorado School of Mines, I tell students of science and engineering that *this* is the course that will separate them from the MATLAB programmers.

Second, MATLAB is a commercial product, with licenses that cost thousands of dollars when various add-ons for signal processing, filter design, etc. are included. The source code for MATLAB is not open for review

or modification by others. MATLAB is therefore an unlikely platform for open-source scientific computing.

Results of scientific research generated with MATLAB may be difficult to reproduce by those who do not have access to this product. Ideally, research should be conducted and reproducible using software tools that are freely available.

1.2 Java and C++

Today, two platforms for scientific computing are most promising. The first *Java platform* is defined by the Java and Jython programming languages and the many packages that are part of the standard Java runtime environment. The second *C++platform* is defined by the programming languages C++ and Python, their standard libraries, plus the non-standard Qt or wxWidgets and other libraries and tools.

Both platforms are freely available and widely used. Both support multiple operating systems, including Linux, Windows, and Mac OS X. Both support the use of software libraries written in other languages. And both facilitate a broad definition of scientific computing.

However, these two platforms are not equivalent. Furthermore, though both enable the use of software written in other languages, such as FORTRAN, *the Java and C++ platforms are largely incompatible. We must choose one or the other.*

```
// findMin in C
double findMin(
  double (*f)(double), double a, double b) {
  // returns x in [a,b] that minimizes f(x)
}
```

```
// findMin in C++
class Function {
public:
  virtual double f(double x) const = 0;
};
class MinFinder {
public:
  double findMin(
    const Function& f, double a, double b) {
    // returns x in [a,b] that minimizes f(x)
  }
};
```

```
// findMin in Java
interface Function {
  double f(double x);
}
public class MinFinder {
  public double findMin(
    Function f, double a, double b) {
    // returns x in [a,b] that minimizes f(x)
  }
}
```

Figure 1. A C function, a C++ class, and a Java class for finding the minimum of a function $f(x)$.

1.3 Why must we choose?

Why not simply use whatever programming language seems best for each task, and then link everything together as necessary? For example, why cannot geoscientists write FORTRAN and computer scientists write C++?

Mixed-language programming has never been robust, but it could always be made to work with languages like C and FORTRAN77. From C functions we could call FORTRAN77 subroutines and vice-versa. This worked because of de-facto standards for binary code generated by compilers for these two languages.

But for languages like C++ no such binary standard exists. Indeed, it is common that C++ software generated by one C++ compiler cannot be linked with a C++ library generated by another C++ compiler.

The lack of a binary standard for C++ is just one problem with mixed-language programming. Another problem is memory management. Scientific software written in Java or C++ typically consumes large amounts of memory for objects constructed on the heap. Java's garbage collection automatically frees memory allocated from the Java heap, but it knows nothing about C++ objects allocated elsewhere. While it is possible to wrap C++ systems in Java, the wrappers simply do not work like other Java classes, and they complicate memory management.

Another problem is exception handling. While Java and C++ exceptions are a significant improvement over error codes returned by functions (and then often ignored), they complicated mixed-language programming.

The same sorts of complications arise when attempting to mix Java or C++ with, say, user-defined types (classes) written in FORTRAN90. Object-oriented programming (OOP) was added to FORTRAN because it is useful for scientific computing. But OOP significantly complicates mixed-language programming, and this complexity will only increase as we develop more object-oriented software.

1.4 Why object-oriented programming?

If OOP complicates mixed-language programming, we might ask whether OOP is worth the trouble. After all, a lot of scientific software has been written with only functions and subroutines.

Our problem is that much of that software is difficult to use, maintain, and extend. Figure 1 provides one simple example. Here, the C function `findMin` is similar to the C function `brent` in Press et al.'s (1988) *Numerical Recipes in C*. It searches for the value x that

minimizes a specified function $f(x)$. In C, we represent the function $f(x)$ by a C function `f`, and we pass a pointer to that function to `findMin`.

The problem with the C function `findMin` is that the C function `f` accepts only one argument `x`, the argument that `findMin` will vary in its search for a minimum. However, in almost any practical use of `findMin`, the function `f` requires additional parameters or state. This extra state must be provided in some other way, because C functions are stateless.

For example, the C function `f` might compute traveltime for a seismic ray with takeoff angle `x`. Then the extra state might include a seismic source location, seismic wave velocities, and so on.

Our problem here is to somehow provide this extra information to the C function `f`. Solutions to this problem in C might be global variables, or another pointer to a structure containing the necessary extra state.

These solutions create other problems and are more complex than the C++ and Java solutions in Figure 1. Here, the classes `MinFinder` find the minimum of a function `f` provided by *any class* that implements the interface `Function`. And, unlike a C function, a C++ or Java class that implements `Function` can encapsulate any extra state required to evaluate the function `f`.

Here, we solved a simple function minimization problem using abstraction, encapsulation, and polymorphism, all fundamental concepts of OOP. While a programming language that supports OOP is not required to solve such problems, an OOP language greatly simplifies their solutions.

And problems like these are common in scientific computing. (We discuss more examples below.) The pervasiveness of such problems and the need to simplify their solution in scientific computing imply that future software systems will increasingly be built on platforms that support OOP.

Therefore, if modern scientific computing requires OOP and the leading Java and C++ platforms for OOP are largely incompatible, then we will be forced to choose one platform or the other.

Before choosing, we should compare different aspects of these platforms. Numerical performance is one obvious aspect, but differences here are less significant than we might expect. Other aspects include portability to different operating systems, and the quality and licensing of software libraries. But complexity is the aspect in which these two platforms differ most, and this difference leads me to favor the simpler Java platform.

2 THE JAVA PLATFORM

Table 1 lists common ingredients of the Java platform for scientific computing. These are, for example, the only components required to build and use the *Mines Java Toolkit*.

Version 5.0 of the Java Development Kit (JDK) from Sun includes over one hundred standard packages, with thousands of classes in those packages, and tens of thousands of methods in those classes. (The packages that I use most are java.lang, java.util, java.io, java.awt, and javax.swing.) Java source code for all of this software is provided with the JDK.

The Java platform

JDK	Sun's Java 2 Standard Edition (J2SE) Development Kit.
Jython	the Python programming language for Java (written in Java).
Ant	Apache's portable build tool for Java (written in Java).
GCC	the GNU compiler collection, for building shared libraries of code written in C++ C, and FORTRAN.

Table 1. A freely available set of tools for development of scientific software on the Java platform.

The JDK includes the tool `javac` for compiling Java programs. Also included in the JDK is the tool `javadoc` for building documentation extacted from comments inside Java source code. Sun uses this same tool to create the documentation for their Java packages, thereby creating a standard for documentation.

Our Java platform includes two non-standard packages — Jython and Ant. Jython is an implementation of the programming language Python. Because both Jython and Ant are written entirely in Java, they are portable to any computing system with a Java Runtime Environment (JRE).

2.1 Jython

We consider Jython to be an essential component of the Java platform for two reasons. The first is that Jython, like Python, facilitates research. Because Jython programs need not be compiled before running them, we can experiment quickly.

What is important here is not so much the small amount of time required to compile a Java program. Rather it is the time required to get back to some point in our research when we decided to try something different, something unanticipated.

If we program only in Java, then such a change in direction requires halting, editing, and recompiling our program, and then finding our way back to where we were before. The time in that last step is what we save by using Jython.

With Jython, we work interactively, using the command line to go in directions that we may not have anticipated when we launched Jython. This is of course the same way that many today work using MATLAB.

To work efficiently in this way, we require a large set

```
# JythonDemo.py
from edu.mines.jtk.dsp import FftReal
from edu.mines.jtk.util import Array
nfft = FftReal.nfftFast()
fft = FftReal(nfft)
rx = Array.randfloat(nfft)
cy = Array.czerofloat(nfft/2+1)
fft.realToComplex(-1,rx,cy)
```

Figure 2. A Jython program that computes the discrete Fourier transform of a pseudo-random sequence of floating point numbers. This program uses the Java classes `FftReal` and `Array` from the Mines Java Toolkit. But these classes have not been specially adapted for Jython; a Jython program can use any Java class.

of precompiled packages of software for scientific visualization, signal processing, linear algebra, etc., all accessible from Jython. This requirement leads to the second reason that Jython is an essential part of the Java platform for scientific computing.

A Jython program can use any Java class.

In particular, Jython has access to every Java class in the standard JRE, in addition to whatever non-standard Java classes we may write. For example, Figure 2 displays a small Jython program that uses two classes from the Mines Java Toolkit. We did not modify the Java classes `FftReal` and `Array` in any way to work with Jython. Nor did we write any Jython-Java *glue code* to enable this access. The Jython interpreter constructs this glue on demand, using information contained in all compiled Java packages.

2.2 Ant

Truly portable software requires a portable build system. For the Java platform, that system is Ant. According to the Apache Software Foundation,

> Apache Ant is a Java-based build tool. In theory, it is kind of like Make, but without Make's wrinkles.

When building software for different operating systems — Windows, Linux, and Mac OS X — the biggest "wrinkle" in using Make is that Makefiles contain commands that must vary among those different systems. Even the syntax for naming a file may be different.

Ant is written in Java so that it can exploit standard Java classes that provide portable methods for compiling and bundling Java software and manipulating filesystems. With Ant, the instructions for building the Mines Java Toolkit on any platform are contained in single file `build.xml`.

Furthermore, Ant can be extended by simply writing new Java classes. For the Mines Java Toolkit, we have extended Ant to enable compilation of non-Java software using the GNU Compiler Collection (GCC).

2.3 GCC

Sun's marketing of *100% Pure Java* never made much sense for scientific computing. Until recent years, Java performance has not been competitive with that of other languages like FORTRAN, C or C++. Even today, it may be impractical to rewrite in Java or translate to Java large libraries of existing scientific software.

A good example is LAPACK, a large (over 600,000 lines of code) set of FORTRAN subroutines for solving dense linear algebra problems. Such problems are pervasive in scientific computing, and LAPACK may well be the most robust and highly optimized numerical software ever written. Who is going to translate it?

Actually, a FORTRAN-to-Java (f2j) translator has been developed specifically to enable automatic translation of LAPACK source code (Doolin et al., 1999). This translator works much like the FORTRAN-to-C translator widely used today to enable the use of LAPACK without a FORTRAN compiler. However, the f2j project is incomplete and seems to have generated little interest.

A simpler solution is to wrap functions in LAPACK libraries with Java. Despite the *100% Pure Java* message, Sun's JDK has always enabled access to software written in other languages — *native code* — using the Java Native Interface (JNI).

Sun of course uses JNI to implement Java packages that require access to facilities that vary among different operating systems. In other words, a *100% Pure Java* strategy means that only Sun writes software in languages other than Java.

This strategy makes sense for Java applets running in a web browser, but in other contexts of scientific computing, it is nonsense. When we need to use a LAPACK subroutine written in FORTRAN, we wrap it in Java, as shown in Figure 3.

This example illustrates the sort of glue code we must write to bridge between Java and non-Java code. To simplify this glue code, we have written shim classes such as `JdoubleArray` in C++ and `#include`'d those in `Lapack.cpp`.

These C++ shims need not copy or convert the contents of Java arrays. Rather, they simply make those contents accessible to non-Java code. The FORTRAN subroutine `dgetrf` then works directly on Java arrays.

In `Lapack.cpp`, we use macros `JNI_TRY` and `JNI_CATCH` and the function `check` to convert errors into Java exceptions. We then compile and link this C++ glue with LAPACK libraries into a shared dynamic library, which we load inside `Lapack.java` using the standard Java method `System.loadLibrary`.

We use this same tactic to wrap other software for which translation may be impossible. For example, in the Mines Java Toolkit, we have wrapped in Java the popular OpenGL libraries for 3-D graphics. Vendors of graphics cards typically provide highly-tuned OpenGL libraries in binary form, without source code. With some

```
// Lapack.java
package edu.mines.jtk.lapack;
class Lapack {
  static native int dgetrf(
    int m, int n, double[] a, int lda, int[] ipiv);
  // ...
  static {
    System.loadLibrary("edu_mines_jtk_lapack");
  }
}
```

```
// Lapack.cpp
#include "../util/jniglue.h"
// ...
extern "C" int dgetrf_(
  int *m, int *n, double *a, int *lda,
  int *ipiv, int *info);
extern "C" JNIEXPORT jint JNICALL
Java_edu_mines_jtk_lapack_Lapack_dgetrf(
  JNIEnv* env, jclass cls,
  jint jm, jint jn, jdoubleArray ja,
  jint jlda, jintArray jipiv)
{
  JNI_TRY
  int m = jm;
  int n = jn;
  JdoubleArray a(env,ja);
  int lda = jlda;
  JintArray ipiv(env,jipiv);
  int info;
  dgetrf_(&m,&n,a,&lda,ipiv,&info);
  return check(env,info);
  JNI_CATCH
}
// ...
```

Figure 3. A Java wrapper with JNI glue written in C++ for the LAPACK subroutine `dgetrf` written in FORTRAN. Subroutine `dgetrf` computes the LU decomposition of a matrix.

JNI glue as in the example above, we can use these libraries in Java programs.

For compiling non-Java code, we chose the GCC because it is freely available for any system with a Java Runtime Environment. Other compilers may be used instead, but the GCC's availability and widespread use lead us to choose it for the both our Java and C++ platforms.

3 THE C++ PLATFORM

Table 2 lists one possible set of ingredients for a C++ platform that is comparable to the Java platform in Table 1.

As for the Java platform, we chose the GCC because it is freely available and widely used on many computing systems. We also favor GCC because it implements well ANSI/ISO standards for the C and C++ languages and their respective libraries.

The C++ platform

GCC	the GNU compiler collection for C++ C, and FORTRAN.
Python	the Python programming language.
Boost	C++ libraries from Boost.org, especially Boost.python and `boost::shared_ptr`. An alternative to Boost.python is SWIG.
Qt	C++ libraries from Trolltech, for graphical user interfaces, and possibly much more. Qt is freely available under either the open-source General Public License (GPL) or a commercial license. An alternative with less restrictive licensing is the wxWidgets library.
Doxygen	a software documentation system for C++ and Python (and Java!) that works much like `javadoc`.
SCons	a portable build tool (written in Python). An alternative is the Boost.Jam build tool.

Table 2. A freely available set of tools for development of scientific software on the C++ platform.

3.1 Python

We have already emphasized the importance of Python for scientific computing in our discussion of Jython above. Python is well-appreciated today by users of numerous Python modules, including NumPy, SciPy, and Matplotlib, developed specifically for scientific computing. With these modules, many today use Python is an alternative to MATLAB.

While Python modules for scientific computing are easy to use, they are more difficult to develop, because we must write glue code that connects Python to libraries of C functions and C++ classes. For Python (unlike Jython), this glue code cannot be generated on demand, because compiled C and C++ libraries do not contain the extra information required to do that. Before we launch Python, we must ensure that we have wrapped everything in those libraries that we might use.

The Boost.python system provides one way to generate these wrappers. As an example, Figure 4 shows how we might wrap one class in a C++ library for digital signal processing. This sort of wrapper enables the use of a C++ class `FftReal` in a Python program much like that in Figure 2.

The Python-C++ glue that we show here reminds us of the Java-C++ glue in Figure 3 that we wrote to enable the use of LAPACK in Java. But the two contexts are different. Python-C++ glue is needed to enable the use from Python of any C++ code developed *within* the C++ platform. The Java-C++ glue is required only when accessing software developed *outside* the Java platform. Python-C++ glue would be more like Jython-Java glue,

```
// PyDsp.cpp
#include <boost/python.hpp>
#include "edu/mines/ctk/dsp/FftReal.hpp"
// ...
using namespace boost::python;
using namespace edu_mines_ctk_dsp;
BOOST_PYTHON_MODULE(dsp) {
  class_<FftReal>("FftReal",init<int>())
    .staticmethod("nfftFast")
    .def("realToComplex",&FftReal::realToComplex)
    // ...
  ;
  // ...
}
```

Figure 4. Using Boost.python to generate Python wrappers for C++ classes in a library for digital signal processing.

except that Jython generates the latter automatically and transparently.

While Boost.python (and alternative systems such as SWIG) simplify the generation of C++ wrappers for Python, they remain more complex and less complete than Jython wrappers generated on demand. Two projects, Pyste and pyplusplus, are currently underway with the goal of generating Boost.python wrappers automatically.

Boost.python is just one library available from Boost.org.

3.2 Boost

We include in the C++ platform the C++ libraries from Boost.org, because many of these libraries will likely become part of the C++ standard library. At this time, ten Boost libraries are already included in the C++ Standards Committee's Library Technical Report (TR1), which is a step toward becoming part of the next C++ standard.

One example is `boost::shared_ptr`, a class template for smart pointers. Smart pointers provide a limited form of garbage collection called *reference counting*. The idea in reference counting is that C++ objects automatically delete themselves when they are no longer referenced, that is, when their reference count goes to zero. The effectiveness of reference counting is limited by the possibility of reference cycles that may prevent reference counts from ever reaching zero.

Despite its limitations, reference counting remains a widely used technique for memory management in C++ and `boost::shared_ptr` will be a welcome addition to the C++ standard.

3.3 Qt

The standard C++ libraries include no components for graphics programming or user interfaces, partly because these components vary so much among different operating systems.

Therefore, we include the Qt C++ libraries from Trolltech in the C++ platform for scientific computing. Qt facilitates the development of portable software for interactive graphical modeling and data visualization.

Qt provides much more as well. Among C++ libraries, Qt with over 400 classes comes closest to providing a complete C++ development kit, much as the JDK does for Java. (In fact, some of the C++ classes in Qt are designed specifically to emulate Java classes in the JDK.) With Qt, one can ignore the C++ standard library entirely.

Unfortunately, Qt is a commercial product with licensing terms that may not be acceptable for some projects. Therefore, many C++ programmers will choose an alternative and incompatible library, such as wxWidgets.

3.4 Doxygen

We include Doxygen in the C++ platform because of favorable experience with the `javadoc` system for generating documentation from comments embedded in Java programs. The proximity of those comments to the relevant source code helps to keep them up-to-date. As programmers change source code, they are more likely to change documentation that is nearby, than to update a separate document.

Although the C++ language does not specify a standard system for documentation, the Doxygen system appears to be a mature and widely used system for this purpose. Doxygen was originally designed for use with C++ projects that use Qt, although Trolltech does not use it for its own documentation. And Doxygen is used to generate the documentation for several of the Boost libraries.

3.5 SCons

According to its authors,

> SCons is an Open Source software construction tool — that is, a next-generation build tool. Think of SCons as an improved, cross-platform substitute for the classic Make utility ....

SCons has built-in support for C, C++, FORTRAN, Qt, and more.

SCons is written in Python, and standard Python modules provide the portable interface to commands and filesystems that enable a single build script to work for multiple operating systems. Furthermore, programmers can extend SCons using the language Python that they should already be using anyway.

In other words, SCons depends on Python just as Ant depends on Java. However, compared to Ant, SCons

```
// FftReal.java
package edu.mines.jtk.dsp;
public class FftReal {
  public FftReal(int nfft) {
    _nfft = nfft;
    // create cosine-sine tables?
    // create work arrays?
  }
  public static int nfftFast(int n) {
    // return valid FFT length not less than n
  }
  public void realToComplex(
    int sign, float[] rx, float[] cy) {
    // transform nfft real to nfft/2+1 complex
  }
  // ...
  private int _nfft;
}
```

Figure 5. Part of a Java class for real-to-complex (and complex-to-real) fast Fourier transforms.

is relatively new and less widely used. A more mature but less ambitious portable build tool is Boost.Jam.

4 JAVA VERSUS C++

Differences between the Java and C++ platforms imply that software libraries written for one will not likely be compatible with the other. This incompatibility is the reason that we must in practice choose one platform or the other.

But what about compatibility *within* these platforms? Within the Java platform, software libraries tend to be compatible because of the large number of classes provided with the standard JDK.

However, within the C++ platform, we must often make choices that are inconsistent with those made by other C++ programmers. As a simple example, to represent strings of text, will we use a `std::string`, a `QString`, or a `wxString`? This question arises only because C++ was used for years before the C++ standard library provided a standard class for character strings. (Yes, I too have written my own class String!)

As another example, consider how we might use a library written using `boost::shared_ptr` with another library that uses `QSharedDataPointer`. Incompatibilities like these within the C++ platform can be overcome, typically by writing shims and adaptors that significantly increase the complexity of our software.

Software complexity is the aspect of the Java and C++ platforms that distinguishes them most.

4.1 Complexity

Figure 5 shows an example of a Java class for computing fast Fourier transforms. This is the same class that we used in the Jython example of Figure 2.

```
// FftReal.hpp
#ifndef EDU_MINES_CTK_DSP_FFTREAL
#define EDU_MINES_CTK_DSP_FFTREAL
#include <complex>
namespace edu_mines_ctk_dsp {
  class FftReal {
  public:
    FftReal(int nfft);
    static int nfftFast(int n);
    void realToComplex(
      int sign,
      const float* rx,
      std::complex<float>* cy);
    // ...
  private:
    int _nfft;
  };
}
#endif
```

```
// FftReal.cpp
#include "edu/mines/ctk/dsp/FftReal.h"
namespace edu_mines_ctk_dsp {
  FftReal::FftReal(int nfft) {
    _nfft = nfft;
    // create cosine-sine tables?
    // create work arrays?
  }
  int FftReal::nfftFast(int n) {
    // return valid FFT length not less than n
  }
  void FftReal::realToComplex(
    int sign,
    const float* rx,
    std::complex<float>* cy) {
    // transform nfft real to nfft/2+1 complex
  }
  // ...
}
```

Figure 6. Part of a C++ class for real-to-complex (and complex-to-real) fast Fourier transforms.

We construct an `FftReal` object by providing a valid length `nfft` for the FFT. To obtain a valid length, we might first call the method `FftReal.nfftFast`. We might then apply the transform by calling the method `FftReal.realToComplex`.

Our current implementation of `FftReal` requires no cosine-sine tables or work arrays. But this fact is hidden from software that uses `FftReal`. If we later change our implementation to one that does require extra tables or arrays, we can make that change without changing or even recompiling any software that uses `FftReal`. This *information hiding* is a feature of object-oriented programming.

Figure 6 shows an example of a comparable C++ class `FftReal`. This is the class for which we used Boost.python to generate a Python wrapper in Figure 4.

The most obvious difference between the Java and

C++ versions of `FftReal` is that the C++ version consists of two files. One file (`FftReal.hpp`) provides the class declaration and the other (`FftReal.cpp`) contains its implementation. Software that uses `FftReal` must `#include` the header file `FftReal.hpp`, and then link with a binary compiled version of the implementation in `FftReal.cpp`.

This separation of interface from implementation is desirable. It enables changes to the implementation in `FftReal.cpp` without recompiling software that `#include`s `FftReal.hpp`. But as the Java version of `FftReal` demonstrates, this separation does not require two separate files.

We need not have used two files in our C++ version of `FftReal`, either. Instead, we could have provided the entire implementation inline in the header file `FftReal.hpp`.

But inlining all of our C++ software would significantly increase compile times and memory consumption. Inlining also complicates techniques for avoiding name collisions. And any change to the implementation in `FftReal.hpp` would then require recompiling all software that uses `FftReal`. Therefore, many C++ libraries, such as Qt, separate class declarations from their implementations, as we have in this example.

However, our choice here is inconsistent with that made for the C++ standard library and most of the Boost libraries. These libraries consist almost entirely of header files that are `#include`d. This is partly because these libraries exploit techniques for generic programming (templates) that make it difficult to hide implementation.

Another complication in `FftReal.hpp` are the lines near the top that begin with `#ifndef` and `#define`. These lines remain necessary today for arcane reasons. Students of C++ routinely forget them, or they copy them from another file and forget to change them, or they forget the matching `#endif` at the bottom. Errors like these cause significant confusion when developing C++ software.

4.1.1 complex numbers

Note that our C++ version of `FftReal` in Figure 6 uses a type `complex<float>` from the C++ standard library. In C++, we can define types like complex numbers and make those types work like C++ built-in types.

Java has no complex numbers. Nor does Java provide the facilities (such as operator overloading) used in C++ to define new types that work like primitive types such as `int` and `float`.

Therefore, in our Java version of `FftReal` in Figure 5, we specify the output array of the method `realToComplex` as an array of `float`s. By convention, the `float`s in this array alternate between the real and imaginary parts of each complex number in the array.

We adopt this convention because complex numbers in Java are both ugly and costly. Figure 7 illustrates the use in Java of a class `Cfloat` from the Mines Java Toolkit. We designed `Cfloat` to work as much as possible like the standard C++ class `complex<float>`.

```
// UseComplex.java
import edu.mines.jtk.util.Cfloat;
public class UseComplex {
  public static void main(String[] args) {
    Cfloat a = new Cfloat(1.0f,0.0f);
    Cfloat b = new Cfloat(0.0f,1.0f);
    Cfloat c = a.plus(b); // ugly
    System.out.println("a+b = "+c);
  }
}
```

```
// UseComplex.cpp
#include <iostream>
#include <complex>
using namespace std;
int main() {
  complex<float> a(1.0f,0.0f);
  complex<float> b(0.0f,1.0f);
  complex<float> c = a+b; // pretty
  cout << "a+b = " << c << endl;
}
```

Figure 7. Java and C++ programs with complex arithmetic.

Because Java does not permit overloading of operators like `+`, a simple `a+b` in C++ becomes `a.plus(b)` in Java. The syntax for complex arithmetic in Java is ugly.

But the bigger problem with complex numbers in Java is that each `Cfloat`, like any Java object, must be constructed on the Java heap. There a `Cfloat` consumes twice as much memory as a C++ `complex<float>` — 16 bytes instead of 8 bytes. For large arrays of complex numbers, we avoid the extra time and space required to construct each `Cfloat` by packing alternating real and imaginary parts into arrays of `float`s.

This packing of real-imaginary parts means that we must remember to construct arrays with lengths equal to twice the number of complex elements to be stored inside them. However, if we forget to do this, the Java Virtual Machine (JVM) will simply throw an ArrayIndexOutOfBounds exception. By default, the JVM will handle this exception by printing a call stack, with line numbers for each method called, that enables us to determine the context of our error.

4.1.2 garbage collection

All Java objects consume an extra 8 bytes of memory. The Java Virtual Machine (JVM) uses these bytes to keep track of objects and to automatically reclaim memory through *garbage collection* when they are no longer needed.

Garbage collection is one of Java's best features. It greatly simplifies memory management, by almost elim-

inating a whole class of errors familiar to C++ programmers, such as memory leaks and dangling pointers.

When we construct a C++ object on the heap with operator `new`, we obtain a pointer to (address of) that object. If we lose that pointer before calling operator `delete`, we then *leak* memory that cannot be reclaimed. If we call `delete` but maintain the pointer, then that pointer is *dangling* and should not be used. In short, a memory leak is a pointer that we *have* forgotten, and a dangling pointer is one that we *should have* forgotten.

These pitfalls in managing memory in C++ have led to various attempts to avoid them. Some establish conventions like "any object A that `new`s another object B is responsible for `delete`ing object B." Rules like this are simple, but overly restrictive and not enforceable.

Reference counting, as in `boost::shared_ptr`, provides a more automatic method for memory management. But reference counting can be costly in both extra time and memory consumed, and it suffers from the problem of reference cycles.

Many have advocated the addition of *garbage collection* to C++; and Microsoft has in fact done this in their language *C++/CLI*, which they describe as an evolution of C++. C++/CLI may be a fine language, but it is not C++.

Garbage collection has always been a part of the Java language. In Java, we `new` objects but cannot `delete` them. The JVM will reclaim their memory automatically, when they are no reachable.

Being reachable is not the same as being referenced. Two Java objects may reference each other cyclically but still be garbage if they cannot be reached (accessed) by any thread running in the JVM.

Figure 8 shows Java and C++ classes for which automatic garbage collection is especially useful. A tetrahedral mesh is a collection of space-filling, non-overlapping tetrahedra, here represented by a compact data structure. With each tetrahedron (`Tet`), we store references to four neighboring `Tet`s and the four `Node`s at its corners.

In C++ these references are pointers to `Tet`s and `Node`s that we must frequently `new` and `delete`, perhaps as we interactively modify a `TetMesh`. Knowing when to `new` a `Tet` is easy. Knowing when to `delete` a `Tet` is more difficult.

Is the `Tet` in question referenced by another `Tet`? If so, then perhaps we should not `delete` it, because that might create a dangling pointer. But what if we plan to also `delete` that other `Tet`? Then we might want to `delete` both of them; forgetting to do so might create a memory leak.

Here, reference counting is not an option, because that would add significant overhead in both time and memory, and because circular references are abundant in a `TetMesh`.

In Java, we simply `new` `Tet`s as we need them, and the JVM automatically frees memory allocated for `Tet`s when they become unreachable garbage.

```
// TetMesh.java
public class TetMesh {
  public static class Node {
    private float x,y,z;
    private Tet t;
  }
  public static class Tet {
    private Node na,nb,nc,nd;
    private Tet ta,tb,tc,td;
  }
}
```

```
// TetMesh.hpp
class TetMesh {
public:
  class Tet;
  class Node {
    float x,y,z;
    Tet *t;
  };
  class Tet {
    Node *na,*nb,*nc,*nd;
    Tet *ta,*tb,*tc,*td;
  };
};
```

Figure 8. Parts of Java and C++ classes that represent an unstructured tetrahedral mesh. A `TetMesh` is a quadrupally-linked list of `Tet`s. Each `Tet` has references to four `Node`s and four neighboring `Tet`s. Each `Node` has a reference to one of the `Tet`s that references it. In C++, the references are pointers.

I have successfully implemented `TetMesh` in both Java and C++. The C++ version is almost exactly the same as the Java version, except for the added complexity of managing memory. Memory management in C++ is almost always feasible, at the cost of significantly increasing software complexity.

4.2 Performance

Years ago, I would implement inner loops of numerical software in C and then wrap those loops in Java using the Java Native Interface (JNI).

This *90% Pure Java* strategy worked well. I wrote computational kernels for which Java was slow in C, and everything else, especially anything object-oriented, in Java. This strategy worked best for applications in which most of the time was spent in that small amount of code written in C.

Today, the difference between Java and C++ (or C) performance is less significant. In some cases, Java is faster. Today, I tend to use JNI only for wrapping huge software packages such as OpenGL and LAPACK, for which only C or FORTRAN interfaces are available.

Figure 9 shows typical benchmark results for four kernels of digital signal processing (DSP). Shown here

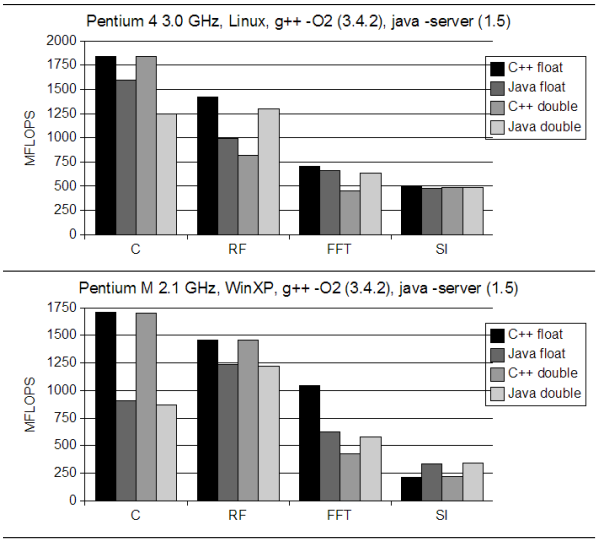

Figure 9. Typical benchmark results for DSP kernels: convolution (C), recursive filtering (RF), fast Fourier transform (FFT), and sinc interpolation (SI). Higher MFLOPS (millions of floating point operations per second) is better.

are results for both Java and C++ versions of single- and double-precision kernels. The Java versions are standalone adaptations from the DSP package in the Mines Java Toolkit. The C++ versions are as similar as possible to the Java versions. Source code for both the Java and C++ benchmark programs is available from the website for the Mines Java Toolkit.

Benchmarking Java against C or C++ is a popular endeavor. Another set of benchmark tests that is especially relevant to scientific computing is the SciMark 2.0 suite, which is also available on the web.

4.3 Summary

We conclude our comparison of Java and C++ with two short lists of features and bugs in the Java platform for scientific computing:

Java features

- huge standard library
- garbage collection
- array index-out-of-bounds checking
- arrays that know their lengths
- simple and useful exception handling
- language support for multi-threading
- fast compiles
- one `.java` file per class
- standard strings of two-byte characters
- Jython's access to any Java class

Java bugs

- no complex arithmetic
- no tiny objects
- no operator overloading

In these lists, adjectives like "huge", "simple", and "fast" are subjective and relative to the C++ platform for scientific computing. We highlighted many of these features and bugs with examples above.

Of course, the best way to appreciate the differences between the Java and C++ platforms is to work extensively with both of them. My experience in industry and in the classroom, developing scientific software in both C++ and Java, leads me to value most the simplicity of development with the Java platform.

5 CONCLUSION

When asked which platform we should use to develop scientific software, we might hope to avoid controversy and answer "both". This answer might even work for software components that are designed to work together only through data exchanged among applications running as separate processes.

But this answer does not work for the foundation libraries upon which future scientific software applications will be built. Libraries of components developed for one platform today are already difficult to use in another. And this difficulty will only increase as scientific software continues to exploit capabilities such as object-oriented and generic programming.

We will not all choose the same platform for scientific computing. But we will all choose, because none of us can afford to work in parallel on mutually incompatible platforms. Choose wisely.

ACKNOWLEDGMENT

Thanks to Steve Smith for running some of the DSP benchmark tests.

REFERENCES

Doolin, D., Dongarra, J., and Seymour, K., 1999, JLAPACK - compiling LAPACK Fortran to Java: Scientific Programming, v. 7, no. 2, p. 111–138.

Press, W.H., Flannery, B.P., Teukolsky, S.A., and Vetterling, W.T., 1988, Numerical recipes in C — the art of scientific computing: Cambridge University Press.