Practical Considerations of Open Source Delivery
Eric Lyness
Microtel, LLC
NASA GSFC Planetary Environments Laboratory, code 699
Software Lead for Mars Organic Molecule Analyzer

The days are long gone when one developer wrote an application that provided the data processing for a project. Code is now developed on many platforms, in many languages and often by scientists and engineers, not just professional programmers. This paper discusses the practical difficulties with delivering the software to support a typical research project.

The most difficult task for projects delivering source code to an open source repository might be simply defining what is meant by “source code.” Projects frequently rely on fragile, ad hoc collections of code in many forms from many sources that have been used and modified over many years by an array of developers with different skills and preferences. Deliveries of the typical jumble of software products supporting research projects to a public archive would be of limited value and very limited lifetime.

Conglomerations
A single project could rely on code existing as:

1. Small scripts in Python or Perl along with some Bash scripts.
2. Applications in C++.
3. Web-based java script applications
4. Code in commercial languages such as LabVIEW, Matlab, IDL, Igor or numerous others.
5. Code embedded in spreadsheets using Visual Basic for Applications or formulas.

This devolution of code into different languages is an inevitable result of laboratory work. It’s caused by two things: First the typical data flow of a research project and second the heritage code of the organization. Whatever the cause, the myriad code used by a project will result in havoc to deliver to an open source archive.

Data Flow from Lab to Laptop
Consider the typical data flow of a research project. Computers in labs record data from instrumentation. Software is written to process the raw instrument files to provide corrections and custom processing for the project. If there is more than one instrument, separate processing is done on each. This partially processed data is combined and trended using another computer and more software, often running on a different operating system. Commercial software such as Matlab or IDL might be used. Finally, the data is sifted, filtered and corrected to get the ultimate science results, often using software on the scientists’ own computers.

In this data flow, the code modules often interact with each other through files of custom, undocumented formats so that they can be copied from one system to another in a makeshift
sneaker-net interface. The best version of a given piece of software or data file is usually just “the one that worked yesterday”.

Heritage Code and Licenses
Teams within NASA rely on a conglomeration of heritage code. This might have been developed by any number of contractors and civil servants over several years and even decades. In an ideal case, a professional software developer has maintained the code in an organized configuration management system. However, this is rarely the case. When it is the case, the organized code might only represent a part of the complete software. Code that is not maintained in configuration management exists on the computer it runs on, and the current version is whatever exists there at the moment.

As the heritage code might be developed over a period of decades, it is likely that at least some of the heritage code had been developed by a contractor under a license that does not permit delivery to a public archive.

External Libraries
Software developers and scientists often rely on third-party libraries. A google search often results in a library that can save the developer some time. The libraries may be commercial or may be open source. A single program might use more than one external library. Often the very specific libraries required by the sort of research NASA does have very limited distribution and support and are compatible only with specific operating systems and specific versions of compilers. In some cases, a computer or operating system might be purchased specifically to be compatible with one of these libraries. Source code dependent on these libraries would be useless without it.

Burden of Delivery
Any delivery to an open source archive would have to:

- Identify all operating systems, development environments and languages with versions.
- Identify any external libraries and where they come from and specific versions and operating systems required.
- Identify code developed by contractors under licenses not permitting delivery to an open source archive and remove it, potentially making the complete delivery non-functional.
- Identify all ad hoc “interfaces” such as moving files from one system to another.
- Identify all the files and versions of the files constituting the complete delivery.
- Identify all configuration files and versions defining the operation of the software.
- Identify code in commercial applications such as Excel

Only a serious, conscientious effort by the research team will provide a delivery that is at all useful. Though this effort might provide a value to the team in the long term, the immediate value to the team will not be apparent. Research money is more and more competitive and the inevitable schedule and budget issues will always pressure the team to provide less than
thorough deliveries or put it off to such a time as the exact software that was used is no longer certain or possibly no longer exists.

An overseeing organization could be employed to review the deliveries, as is done with data delivered to the Planetary Data System. However, such a review would be exceptionally expensive when it comes to source code. Reproducing the operating systems, development environments and commercial software alone would be unmanageable. But even if this were done, it would be insufficient as the reviewer would also need some understanding of the data and the science otherwise the reviewer could easily miss the case where only a small, though functional, part of the data pipeline is delivered.

**Users’ Problems and Software Life Expectancy**

Like a reviewer, users of the archive would need to reproduce the operating systems and development environments, purchase any commercial software, and find any external public libraries. Operating systems quickly become obsolete and organizations such as NASA may not even permit their use. For instance, an Apple MAC operating system released in 2016 is now forbidden on the NASA network less than two years later.

Commercial development environments, such as National Instruments LabVIEW, rarely allow users to download old versions of their application. External public libraries go through continuous upgrades by the community and backwards compatibility is unlikely to exceed a few years.

**Evolving Languages**

Consider the C++ programming language. This is one of the most stable languages. The standard rarely changes and C++ code written 20 years ago might still compile today. However, the compiler is the least of the problem. The code relies on an operating system interface and run time library that supports the user interface. There are no standards for these interfaces and they depend on the environment used to develop the code. C++ code written using Microsoft Visual Studio will not be inherently compatible with a Linux machine because of the operating system dependent libraries. Even within Windows, the code that compiled and ran under Windows 7 may not compile and run in future Windows versions. Scripting languages such as Python suffer the same limited life issues but also suffer from a standard that changes more often than C++. For instance, code in Python 2.x is often not compatible with Python 3.x. Code in commercial applications such as Matlab or Excel are dependent on the commercial provider maintaining backward compatibility, or in fact staying in business.

**Summary**

Even a complete and conscientious delivery of the source code would be obsolete within as little as two years as a combination of operating systems, languages and support libraries change. Those responsible for the delivery will understand this imminent obsolescence and be less motivated to spend the time to generate a complete, functional delivery.
A review of the delivery would be very expensive especially considering the limited life expectancy of the code.

The end users will have a large burden in trying to make the software work given the myriad sources, operating systems and external libraries.

If the intent of open source code delivery is transparency to the public, the difficulties of using the software and the rapid obsolescence of the software could be perceived as deliberate obfuscation and lead to results opposite of the intent.

**About the author:** Eric Lyness has been developing software for laboratory automation systems for over 25 years and working on NASA systems for over 15 years. He has supported software for small research projects as well as large flight projects including the Sample Analysis at Mars instrument aboard the Curiosity rover. He wrote the software and generated the Planetary Data System delivery for the LADEE Neutral Mass Spectrometer mission. He is now the software lead for the Mars Organic Molecule Analyzer, scheduled to land on Mars in 2021.