Response to Reviewers' Comments on "PyTrilinos: Recent Advances in the Python Interface to Trilinos"

William F. Spotz

January 31, 2012

I would like to thank the reviewers for their thoughtful comments and thought-provoking questions. My responses to these comments and questions are given below:

Reviewer 1

1. **The author provides some short examples of the PyTrilinos module. Practitioners will need to write longer codes to implement many practical programs, particularly when using Trilinos to perform numerical simulation. Are there any software design approaches or code structuring techniques which the author has found particularly useful for building good software? A capstone example of a slightly larger scientific computing problem being solved with this package would be appreciated.**

   It has been my observation of scientific software development that agile development is among the most popular and effective approaches. This is likely because software that is intended to answer research questions is notoriously resistant to a requirements phase. Agile development rejects planning in favor of feedback from an iterative approach to code development. Python is well-suited to this because of its demonstrated ability to quickly facilitate high-level code development. NumPy and SciPy enable a wide range of scientific computing categories. PyTrilinos adds to this functionality that which is needed for robust implicit solvers, thus expanding the capability space of agile-developed Python code for scientific research.

   In specific response to the reviewer's question, I have moved most of section 2 to section 2.2 and added a section 2.1 that reviews the origin of PyTrilinos. This is a multiphysics coupling application enabled by Jacobian Free Newton Krylov that hopefully provides the "capstone" example the reviewer is looking for.

2. **In the same vein as the previous comment, are there any particularly good methods for using PyTrilinos in an objected oriented fashion? Are there some particularly useful inheritance and/or polymorphic examples derived from PyTrilinos classes the author has found useful?**

   In answering questions from all three reviewers, I have provided several new examples of useful and preferred ways of using PyTrilinos in the manuscript. The answer to question 1

is a good example of this, but other examples are now scattered throughout the text as well (and in my responses here).

3. **I am curious if the SWIG approach has led to any compromises in code design beyond what was described in Section 7. Would there be any significant advantages to a custom API for parts of Trilinos?**

   The two primary compromises due to using SWIG are the handling of templates (covered in section 7) and nested classes, which SWIG does not support. I have added a discussion of nested classes to section 2.2, paragraph 2.

4. **One important advantage of PyTrilinos is its ability to set up parallel computations. While it looks like the example in Section 4.2 is on multiple processors, more direction as to how to set up and partition a parallel computation using PyTrilinos would be helpful.**

   Section 2.2, paragraph 5 now goes into more detail regarding running parallel PyTrilinos scripts.

5. **The author notes on page 5 that MATLAB outperforms PyTrilinos for dense matrix/vector multiplies. Is there some design choice in particular that necessitates this or is it a matter of expending effort where best used?**

   This is because Trilinos is designed for unstructured sparse problems. The requisite indirect addressing introduces some overhead for dense problems relative to MATLAB. This issue is independent of the Python interfaces. I have expanded the discussion in section 2.2, paragraph 6 to address this.

6. **The author brings up the infamous loop performance issue which is common to many interpreted languages. One of the most important things to teach users is how to avoid loops when using these tools. Are there any capabilities in PyTrilinos which can be used to avoid direct loops?**

   Yes, there are some capabilities in PyTrilinos for avoiding loops, but not as many as there should be. `Epetra.Vector`s, for example, are also NumPy arrays, and so can take advantage of slice indexing to avoid Python loops. Other objects, such as graphs or matrices, do not provide access to slice indexing. This fact is now mentioned in section 4.3 after the code example. Other techniques for avoiding loops are now expanded upon in section 2.2, paragraph 7, using a bulleted list to increase clarity.

7. **The descriptions in Section 4 are a bit difficult for those who aren't very familiar to Trilinos to understand. A bit more detail about these methods, as well as their uses, would be helpful.**

   I have rewritten the 4.x subsection introductions so that they are hopefully more clear to readers not familiar with Trilinos.

8. **It looks as though the example in Section 4.3 generates a figure. Could this be included?**

   The script as written does not produce a figure, but that ability was an important component of the work performed by the students who developed the Python interface to Isorropia. The `buildgraph()` function in that script builds a simple tridiagonal matrix, which would not produce an interesting figure. I now discuss that function in terms of the loop issue, and

so am disinclined to change it. On the other hand, I understand the interest in seeing this particular type of output. So I have expanded the discussion somewhat to cover graphical representations of matrices and provided a visual example for a more interesting matrix.

9. **It was unclear to me based on the description in Section 5.2 what, if anything, the programmer needs to do to declare an object that will play nicely with the `Teuchos::RCP` approach. Please clarify the programmer's responsibilities. With respect to `RCP`, one of the main performance issues is allocation/deallocation. Is there a way to reduce deallocation unless more space is needed? Also, some functions may be called repeatedly and allocate/deallocate memory which could be retained unless it caused problems in the context of scientific computing.**

   The design principle is that the Python programmer will never have to write code that refers to a `Teuchos::RCP`. Inevitably, however, a PyTrilinos user will encounter documentation that refers to objects encapsulated by `Teuchos::RCP` (for example, the automatically-generated docstrings include both the Python and C++ signatures of the exposed and underlying methods). For this reason, it is very useful for a PyTrilinos user to understand that method arguments or return values that are of type `Teuchos::RCP<object>` in C++, are simply handled as type `object` in Python. Since this was not clear, I have attempted to re-write section 5.2 to improve clarity.

   In general, reference-counted pointers should reduce deallocations. But nothing special was done to further reduce deallocations. Interestingly, developers of Jpetra (the Java interface to Epetra) ran into the opposite problem: they could not control deallocation and thus ran out of memory in certain situations because objects that were no longer needed were still occupying memory.

10. **Section 7: templated classes ... nasty! How can Cython help specifically? This seems like a deep and fundamental incompatible implementation problem, but one which Python avoids by use of PyObjects. Can we have templates make equivalent `PyTrilinosObjects` for better interaction with Python?**

    My goal is to develop a distributed vector type in Python that handles a variety of scalar types the same way that non-distributed NumPy arrays do. While the implementation details are currently a topic of debate, what is clear is that the Python `PyTrilinos.Tpetra.Vector` interface will necessarily be significantly different from the C++ `Tpetra::Vector` interface. To use SWIG to accomplish this, we would have to write a new C++ class and tell SWIG to wrap that. Cython will require a similar approach, but ultimately the new class will be easier to write because Cython interface design is easier than C++ interface design, and Cython interfaces are more appropriate for Python.

    The reviewer is correct that supporting templated classes is a deep and fundamental problem. If a compiled extension module for Python is written using C++ templates, then every supported type must be included a priori in the compiled object. The compiled extension can never be as flexible as pure Python code. On the other hand, our goal, which is similar to the goals of NumPy, is to support a well-defined subset of numeric types (acknowledging that NumPy does support some more exotic types as well). We believe this goal is achievable.

    I don't think we need to define a `PyTrilinosObject`. A very general approach would be to have every PyTrilinos class derive from `PyObject` (which they already do) and then write specializations for every supported templated class where every template argument is a `PyObject`.

I think ultimately this would impose a steep performance penalty and I am much more inclined to support specific cases such as the `Tpetra.Vector` case I describe in the paper.

Reviewer 2

1. **It would be helpful to include a reference or two where the capabilities of the package have been showcased. This will provide supporting evidence and strengthen PyTrilinos position as an open source, massively parallel mathematical and scientific computing tool.**

   A short discussion of other projects that use PyTrilinos, including appropriate citations, has been added to the end of the Introduction.

2. **The author mentions several advantages and disadvantages of PyTrilinos. The description of some of the advantages is scattered throughout the paper and section 7 of the article gives a great account of the last issues and possible solutions. It would be good to add a couple of sentences on the other issues as well (page 5, par. 4).**

   I have split the paragraph on advantages and disadvantages into two paragraphs, and expanded my discussion in each.

3. **A short code example in section 3, showing the docstrings generated by SWIG with and without the help of the XML files, might be helpful in further illustrating the improvement in documentation.**

   This has been added.

4. **Section 5.2 should include a few sentences with the differences between `Teuchos::RCP` and `boost::shared_ptr`.**

   This has been added, including a citation to the Sandia Technical Report *Teuchos C++ Memory Management Classes, Idioms, and Related Topics: The Complete Reference.*

Reviewer 3

1. **It may be helpful to highlight python packages that rely on PyTrilinos in a major way. This will help to emphasize the importance of PyTrilinos.**

   Reviewer 2 made a similar comment. This discussion was added to the end of the Introduction.

2. **The example on page 8 uses `galeriList = { "n":  10*10, "nx":10, "ny":10 }`. It would be better to write:**

```
nx = 10
ny = 10
galeriList = { "n": nx * ny, "nx" : nx, "ny" : ny },
```

   This change has been made.

3. **In the example on page 9, "10" is explicitly used in multiple places without being parameterized.**

   This value is now parameterized.

4. **In the examples on page 11 and 12, python loops are used. Would this be a good opportunity to demonstrate the use of numpy in conjunction with pytrilinos by vectorizing the loops if possible? If not, it might be nice to include an example of numpy-pytrilinos interaction as this is a vital aspect of using PyTrilinos effectively.**

   It turns out this is not an opportunity to demonstrate the use of NumPy to vectorize loops because the loop iterates over rows of an `Epetra.CrsGraph`, and this object does not support slice indexing. Back where I first mention the use of slice indexing to improve performance in the Review, I have expanded the text to include an example. Beyond this paper, the idea of adding slice indexing (or even fancy indexing) to graph and matrix objects is a good one, and will hopefully find its way into future versions of PyTrilinos.

5. **On page 5, in the statement regarding ref 12, is it only trilinos itself that is slower than matlab for dense matrix calculations or is it the pytrilinos interface between python and c++ that degrades efficiency? I think that should be clearer.**

   Reviewer 1 posed a similar question. It is Trilinos that is slower than MATLAB for dense calculations, and not the interface or wrapper logic that causes the difference. This is because Trilinos in general is optimized for unstructured, sparse calculations and that introduces a layer of indirect addressing. I have re-written this section to hopefully make this more clear.

6. **On page 16 the concept of a model evaluator is rather abstract and not clear to the reviewer. Can a concrete example be given or could more explanation be given. What objects would the model evaluator have in state? Does it describe the model in some way?**

   I have rewritten the introductory paragraph for an audience less familiar with the concept of model evaluators. I now explicitly list all of the different types of models that Trilinos `ModelEvaluator` classes support, and I think this makes the idea much more clear. I decided against a coding example, because they tend to be long and I was afraid it might create more confusion than clarity.

7. **Extra comma in reference 2 listing.**

   Done.