PyTrilinos: Recent Advances in the Python Interface to Trilinos

William F. Spotz

Sandia National Laboratories, Albuquerque, NM

May 31, 2011

Abstract

PyTrilinos is a set of Python interfaces to compiled Trilinos packages. This collection supports serial and parallel dense linear algebra, serial and parallel sparse linear algebra, direct and iterative linear solution techniques, domain decomposition and multilevel preconditioners, nonlinear solvers and continuation algorithms, eigensolvers and partitioning algorithms. Also included are a variety of related utility functions and classes, including distributed I/O, coloring algorithms and matrix generation. PyTrilinos vector objects are compatible with the popular NumPy Python module. As a Python front end to compiled libraries, PyTrilinos takes advantage of the flexibility and ease of use of Python, and the efficiency of the underlying C++, C and FORTRAN numerical kernels. This paper covers recent, previously unpublished advances in the PyTrilinos package.

Keywords: Trilinos, Python, sparse linear algebra, parallel programming, nonlinear solvers, eigensolvers, partitioning, preconditioners.

1   Introduction

Python is an increasingly popular language for scientific computing. NumPy [10] (Numerical Python) provides an extremely convenient, high-level interface to homogeneous, contiguous arrays of data. SciPy [5] (Scientific Python) is built on top of NumPy and provides a Python interface to the most popular scientific libraries available. Matplotlib [8] provides high-quality plotting capabilities. Parallelism is available via a long list of open source modules. Robust solvers are provided by the Python interfaces to PETSc [2] and Trilinos. The SciPy Conference [12] continues to grow in popularity, and the presence of several Python sessions at scientific computing conferences such as SuperComputing [15] and the SIAM Conference on Computational Science and Engineering [14] attest to the strong interest in using Python for scientific programming. In fact, Python is now seen as a credible competitor to MATLAB [9], perhaps the single most popular platform for small to medium scale scientific computing. This is in part because Python and MATLAB share certain traits in common; they are both high level, interpreted, interactive and dynamic.

At the other end of the scientific computing spectrum are compiled codes written in FORTRAN, C and C++ that run on massively parallel architectures to solve large, multiscale, multiphysics problems. These codes utilize low level languages in an attempt to achieve near-optimal processing efficiencies. While their success at this goal is debatable, there is no question that compiled languages are more efficient than interpreted languages.

However, as the complexity of simulations increases, more and more code is devoted to setup and initialization of a series of subproblems that must all interface together to describe the larger problem. This "command and control" software can dominate the numerical kernels in terms of total lines of code, while the kernels themselves still dominate runtime execution. And low level languages are notoriously ill-suited to these types of bookkeeping tasks. It is not hard to make the case that low level languages should handle the numerically intense kernels and high level languages should be utilized for command and control.

This perhaps explains the rise in popularity of Python for scientific computing. As a high level,

object oriented language, it is very well suited to the high level description of scientific problems, to provide the modularity so often sought by scientists asking "what if" questions, and to glue together the various components of a successful scientific simulation. A well designed scientific Python code will hand off the numerically intense computations to optimized compiled code, while maintaining usability, readability and developmental scalability at the highest level.

PyTrilinos provides just such a capability. By supporting massively parallel vector and multivector objects as well as massively parallel operators such as sparse matrices, along side solver algorithms that utilize these objects, PyTrilinos enables the development of complex codes with convenient interfaces provided by Python combined with the efficient solution algorithms of Trilinos. This paper details recent advances in the PyTrilinos package, including its new documentation system, recently added Trilinos modules, a focus on improvements to the Teuchos module, and model evaluators. We finish with a vision for interfaces to second generation Trilinos packages.

2 Review

The first PyTrilinos article [11], published in 2008, described the status of the project circa 2007. In order to put recent advances in context, we begin with a review of the topics covered and conclusions drawn in that paper.

PyTrilinos is a Python interface to selected Trilinos packages. The "wrapper code" necessary to provide such an interface is generated almost entirely by SWIG [6], the Simple Wrapper and Interface Generator. Other tools could have been chosen to facilitate this wrapping, including SIDL/Babel [4], Boost.Python [3], and more recently, Cython [13]. All three of these tools require a manually written interface definition, which can fall out of sync with the wrapper source if the source interface changes. This is perhaps the primary advantage of using SWIG to wrap Trilinos, which is a very large code base that is under active development. The default SWIG behavior is to produce Python interfaces with a near one-to-one mapping to it source C or C++ code. It was decided early in the PyTrilinos project that this was an appropriate approach for a code base as

large as Trilinos.

Because of its strong compatibility with NumPy (Numerical Python), PyTrilinos plays a role in a suite of mathematical and scientific Python tools that can be interpreted as an open source competitor to MATLAB. This suite has the advantage of being free, of leveraging a more powerful scripting language, and of expanded interoperability with other languages. PyTrilinos adds to this suite implicit solver capabilities designed from the ground up for massively parallel computing.

At the time of publication of [11], PyTrilinos provided interfaces to linear algebra services packages Epetra and EpetraExt; tools and testing packages Teuchos, Triutils, Galeri and New_Package; linear solver packages Amesos and AztecOO; preconditioning packages IFPACK and ML; and nonlinear solver packages NOX and LOCA. Since this publication, PyTrilinos has released wrappers to eigen-solver package Anasazi and complex linear algebra services package Komplex. The next release of Trilinos should include a new PyTrilinos package Isorropia, which provides parallel partitioning algorithms. Support for New_Package has been discontinued, as the purpose of this package is to provide a template for adding new packages to Trilinos, and its utility for new PyTrilinos packages was limited.

The Trilinos approach to supporting parallelism is to provide abstract communicator classes (for historical reasons, one in Epetra and one in Teuchos) and then provide concrete implementations of these classes in the form of serial communicators and MPI communicators (and potentially others in the future). If Trilinos is configured with MPI support, then the MPI communicators are made available to both the C++ and Python programmer. Furthermore in PyTrilinos, `MPI_Init()` will be called automatically when the Epetra or Teuchos modules are imported, and `MPI_Finalize()` will be registered with the `atexit` module. This module behavior is made compatible with other MPI Python interfaces such as pyMPI, PyPar or mpi4py by executing these initialization and finalization calls only if MPI has not already been initialized. If `MPI_Init()` has already been called, then PyTrilinos assumes that the user is responsible for finalization.

Reference [11] contains comparisons between PyTrilinos and MATLAB for basic operations that are common to both. The high-level conclusions are that MATLAB is more efficient at performing

dense matrix vector multiplies and that PyTrilinos is more efficient at both sparse matrix vector multiplies and sparse matrix assignment. These results reflect the different primary objectives of the two tools.

PyTrilinos was also compared to Trilinos itself. The primary usage differences are the dynamic nature of Python; memory management concerns present in C++ but not Python; no header files in Python; and high-level containers in Python versus low-level C arrays. The performance differences depend ultimately upon the granularity of the algorithm being tested. Fine grained algorithms that do a lot of work (especially loops) in Python suffer significant performance penalties. Coarse grained algorithms that hand off the bulk of the work to compiled routines can exhibit imperceptible performance hits in both serial and parallel. In situations where no coarse grain algorithm exists in Python, there are potential alternatives to avoid Python loops, for example slice syntax in NumPy, or weave, which compiles embedded C/C++ code in Python. Other alternatives include the previously mentioned wrapper tools SWIG, SIDL/Babel, Boost.Python or Cython.

The advantages of PyTrilinos include rapid prototyping, brevity, modularity, reusability, explorative computing, integration, software quality via testing in Python, stability, and high-level scripting for data input. The disadvantages include portability issues, shared library support on massively parallel computers, lack of compile-time checks, performance considerations, and an awkward mapping of C++ template code (see Section 7 for more details on this last issue).

3 Documentation

With Trilinos Release 9.0 (PyTrilinos version 4.1) a significant improvement to PyTrilinos documentation was made. The most important form of documentation for Python modules is the docstring. Wherever the first line of code within a Python function, method, class or module is a string, that string is interpreted as documentation for that code element. Many Python documentation tools, including the Python `help()` function, access these docstrings for their content. SWIG provides some very basic docstring generation capabilities, but this is limited to docstrings that contain the

code element name (function, method, class or module), and argument types and names (where appropriate). Such docstrings are certainly better than nothing, but often fall far short of describing the code sufficiently to help programmers use it. In the code-generation environment of SWIG, it is possible to provide static documentation directives, but difficult to provide these for every function, method, class and module for a large project such as Trilinos.

Furthermore, static documentation would be undesirable because Trilinos developers provide significant documentation of code elements via Doxygen [7]. Doxygen is a tool that parses C/C++ code and interprets specially designated comments as documentation. Doxygen uses this information to generate high-quality documentation in a variety of forms including web pages and PDF files.

Doxygen can dump its code parse tree and associated documentation strings to an XML file. This capability allowed our colleague Prabhu Ramachandran of the Indian Institute of Technology to write a Python script that parses this data and then writes a SWIG interface file that provides directives that generate automatic docstrings. The end result is that SWIG will generate Python code that includes docstrings that mirror their corresponding C++ Doxygen documentation. These docstrings can be overridden on a case-by-case basis where the Python interface differs from the C++ interface. This results in an extremely useful, interactive documentation system for PyTrilinos.

4   Recently Added Packages

4.1   Anasazi

Anasazi [1] is the Trilinos eigensolver package and was one of the first Trilinos packages to utilize templates. Templates in Anasazi allow a single code base for different scalar types (`float`, `double`, `complex`, `complex double`, etc.), different multivector types (`Epetra_MultiVector`, `Tpetra::MultiVector` and a variety of Thyra multivectors, etc.), and operator types (`Epetra_Operator`, `Tpetra::Operator`, etc.). Since the only supported set of scalar data, multivectors and operators in PyTrilinos is the Epetra package, the Python interface to Anasazi in PyTrilinos accepts only Epetra objects.

Perhaps the most important customization to the Python interface to Anasazi is the conversion of returned eigenvalues from C++ type

```
std::vector< Anasazi::Value< ScalarType > >
```

to a NumPy array of complex double precision values. Along with the `Anasazi::Eigensolution` and `Anasazi::MultiVec` classes, this covers the Anasazi data types.

`PyTrilinos.Anasazi` also provides output and sort managers, as well as operator, eigenproblem and status test classes to give full coverage of the package. The list of supported eigensolvers includes block Davidson, block Krylov-Schur and LOBPCG.

The following is a simple script for solving the first four eigenvalues of a matrix obtained from finite differencing the 2D Laplace operator on a 10×10 grid using Block Davidson:

```
from PyTrilinos import Epetra, Galeri, Anasazi
comm       = Epetra.PyComm()
nx         = 10
ny         = nx
galeriList = { "n": nx*ny, "nx":nx, "ny":ny }
map        = Galeri.CreateMap("Linear", comm, galeriList)
matrix     = Galeri.CreateCrsMatrix("Laplace2D", map, galeriList)
printer    = Anasazi.BasicOutputManager()
nev        = 4
ivec       = Epetra.MultiVector(map, blockSize)
ivec.Random()
myProblem  = Anasazi.BasicEigenproblem(matrix, ivec)
myProblem.setHermitian(True)
myProblem.setNEV(nev)
myProblem.setProblem():
myPL = {"Which"                 : "LM",
```

```
            "Block Size"           : 5,

            "Num Blocks"           : 8,

            "Maximum Restarts"     : 100,

            "Convergence Tolerance" : 1.0e-8 }
    mySolverMgr = Anasazi.BlockDavidsonSolMgr(myProblem, myPL)

    returnCode  = mySolverMgr.solve()

    sol         = myProblem.getSolution()


    if comm.MyPID() == 0: print sol.Evals()
```

4.2   Komplex

Komplex is a workaround package to provide support for complex linear algebra problems using the double-precision real-valued-only Epetra package. This involves storing two vectors of the same type and distribution, one representing the real part and one representing the imaginary part. For dense or sparse matrices, two matrices are stored, again for the real and imaginary parts. This is accomplished with the definition of a single class, Komplex_LinearProblem, which translates to a Python class Komplex.LinearProblem. The following script is a simple example of using the Komplex module:

```
    from PyTrilinos import Epetra, AztecOO, Komplex
    comm = Epetra.PyComm()
    n    = 10 * comm.NumProc()
    map  = Epetra.Map(n, 0, comm)


    # Build the problem matrix
    Ar   = Epetra.CrsMatrix(Epetra.Copy, map, 1)

    Ai   = Epetra.CrsMatrix(Epetra.Copy, map, 1)

    for gid in map.MyGlobalElements():
```

```python
    Ar.InsertGlobalValues(gid, [10 * (1 + float(gid)/n),], [gid,])
    Ai.InsertGlobalValues(gid, [10 * (1 - float(gid)/n),], [gid,])
Ar.FillComplete()
Ai.FillComplete()


# Build the solution vector and the RHS
xr  = Epetra.Vector(map)
xi  = Epetra.Vector(map)
br  = Epetra.Vector(map)
bi  = Epetra.Vector(map)
for gid in map.MyGlobalElements():
    lid = map.LID(gid)
    br[lid] = Ar[gid,gid] * -1.0
    bi[lid] = Ai[gid,gid] * -1.0


# Build the complex problem
problem = Komplex.LinearProblem(1, 0, Ar, 1, 0, Ai,
                                xr, xi, br, bi)


# Set up the solver and iterate to a solution
solver = AztecOO.AztecOO(problem.KomplexProblem())
plist  = {"Solver"  : "GMRES",
          "Precond" : "None",
          "Output"  : 16      }
solver.SetParameters(plist,True)
solver.Iterate(n, 1e-5)
if comm.MyPID() == 0: print "Real part"
print xr
if comm.MyPID() == 0: print "Imaginary part"
print xi
```

9

4.3 Isorropia

Isorropia is a Trilinos package built on top of the Zoltan package for partitioning and dynamic load balancing. It provides object oriented interfaces to Zoltan specialized for Epetra, Tpetra and potentially other future linear algebra packages. Isorropia is used to construct or modify partitions for vector and sparse matrix classes.

In 2010-11, Sandia National Laboratories participated in a Harvey Mudd College Clinic that teamed Sandia researchers with Harvey Mudd students Joe DeBlasio, Katie Ewing, Audrey Lawrence and Michael Leece to conduct research on new matrix partitioning methods. One of the benefits of this project was the development of a Python interface to Isorropia so that the students could rapidly develop a Python-based visualizer for the matrix partitions they were producing.

`PyTrilinos.Isorropia` mirrors the C++ Isorropia package in that is has a top-level module with abstract base classes for operators, colorers, partitioners, redistributors, cost describers, orderers and level schedulers. There is then an Epetra submodule that contains concrete implementations for these classes implemented for Epetra data objects.

The students developed their project within their own repository, and the SWIG wrappers they produced depend somewhat on the modifications they made to the Isorropia code. While this new Isorropia code will probably be ported to the main branch slowly, it is nevertheless our intention to include Isorropia as a wrapped PyTrilinos package with Trilinos release 10.8.

The following is a short script that demonstrates usage of some of the Isorropia classes:

```
from PyTrilinos import Epetra, Isorropia, Teuchos
def buildGraph(comm, nRows):
    map = Epetra.Map(nRows, 0, comm)
    graph = Epetra.CrsGraph(Epetra.Copy, map, 3)
    for gid in map.MyGlobalElements():
        if gid == 0:
```

```python
            indices = [0, 1]
        elif gid == nRows-1:
            indices = [nRows-2, nRows-1]
        else:
            indices = [gid-1, gid, gid+1]
        graph.InsertGlobalIndices(gid, indices)
    graph.FillComplete()
    return graph


comm       = Epetra.PyComm()
nRows      = 10 * comm.NumProc()
crsg       = buildGraph(comm, nRows)
colorer    = Isorropia.Epetra.Colorer(crsg)
print colorer.elemsWithColor(0)
pList      = {"Partitioning Method" : "Random"}
partitioner = Isorropia.Epetra.Partitioner(crsg, pList)
partitioner.partition(True)
print partitioner
pList["Partitioning Method"] = "HyperGraph2D"
part2D     = Isorropia.Epetra.Partitioner2D(crsg, pList)
print part2D
Redis      = Isorropia.Epetra.Redistributor(partitioner)
newCrsg    = Redis.redistribute(crsg)
print Redis
print newCrsg
```

5 Improvements to the Teuchos Module

As the primary tools package for Trilinos, Teuchos has many useful capabilities. Some of these capabilities are already provided by standard Python libraries, e.g. command-line option and argument processing, and so are not wrapped in PyTrilinos. Thus the list of Python interfaces to Teuchos tools is small relative to the package's overall capabilities: communicators, reference counted pointers and parameter lists, including XML support.

5.1 Teuchos Communicators

Teuchos communicators are analogous to Epetra communicators: a virtual base class for the purpose of polymorphism, and serial and MPI concrete implementations to enable these two modes of operation. Unlike an Epetra communicator, Teuchos communicators are templated on an ordinal type and include some advanced programming techniques not allowed under the self-imposed restrictions for Epetra. There is also a default Teuchos communicator based on an ordinal of type `int`.

5.2 Teuchos Reference Counted Pointers

The new reference counted pointer support is perhaps the most important improvement to PyTrilinos involving Teuchos. The `Teuchos::RCP` class is very similar to the `boost::shared_ptr` class with a few minor differences. By developing `Teuchos::RCP`, Trilinos developers have avoided a dependency on boost and have gained tighter control of memory management issues.

Reference counted pointers are a memory management technique that allows objects to be allocated dynamically, be referenced by other objects, and to delay deallocation until the last object that needs a reference to that object has been destroyed. In pure Python, reference counting is performed automatically without any effort from the programmer whatsoever. In the Python/C API, the programmer must pay careful attention to reference counting issues, and call incrementing

and decrementing macros at the proper times to ensure proper memory management. In C++, classes such as `boost::shared_ptr` and `Teuchos::RCP` have been written that keep reference counts current automatically.

More and more Trilinos packages are adopting the `RCP` as a means of secure memory management. This has presented a problem for PyTrilinos. The philosophy of maintaining interfaces with nearly one-to-one mapping between C++ and Python is hard to justify in the face of `RCP`s. Python programmers would be asked, in what would appear to be a mostly random interface, to sometimes pass an object to a function and other times create a new object, encapsulated in the `RCP` class, in order to pass to a function. This could be nothing but a source of frustration for Python programmers, who have always been protected from the issue of memory management.

The proper way to implement shared pointers is to hide them completely from the Python interface. To their credit, SWIG developers have recognized this and have provided an experimental capability to do this. But it was not until the release of SWIG version 2.0 that all the bugs were addressed in the Python code generator.

The basic idea is this: if a class defines a type that is at some time referenced as a shared pointer, then all instances of that object within the Python extension module should be stored as a shared pointer. The default storage method is with a raw pointer that relies upon Python reference counts to determine when the object is deallocated. By changing the storage technique, this implies that the conversion code for accessing an instance must be specialized in all cases. In fact, the bulk of the shared pointer support in SWIG is a set of new typemaps that define these new conversions.

In order to provide the support for `Teuchos::RCP` that SWIG now provides for tt shared_ptr classes (in both the `boost` and `std` namespaces), we had to define some macros prior to applying the `%include` directive to the SWIG `boost_shared_ptr.i` interface file in order to refer to `Teuchos::RCP`. This was far more straightforward than might be expected. We had to override the output typemaps in order to use the ownership flag that the `RCP` constructor provides, and we had to provide director input and output typemaps, which the SWIG library currently does not implement (see Section 6 for a discussion of SWIG directors).

The end result is that whenever a Trilinos C++ routine argument requests an `RCP`-wrapped object, the equivalent Python routine argument will accept an unadorned object. This results in simplified programming with increased readability an no loss of capability.

5.3 Teuchos Parameter Lists

At the time of [11], `Teuchos::ParameterList`s were supported. Wherever a `ParameterList` was expected as input, the Python programmer could provide a Python dictionary in its place. Wherever a `ParameterList` was returned as output, a new type of object, a `PyDictParameterList`, was returned in Python. This has been simplified and improved somewhat. Python dictionaries are still accepted as input, but now the `ParameterList` class has been wrapped properly, with the addition of several methods and operators so that it behaves also as a Python dictionary. This allows access to some `ParameterList` capabilities that Python dictionaries do not possess, such as the usage flags, while still allowing Python programmers to access `ParameterList` capabilities using highly convenient dictionary syntax.

Another addition to the Teuchos Python module are wrappers for XML classes XMLObject, XML-ParameterListReader, XMLParameterListWriter, XMLInputSource, FileInputSource and String-InputSource, which together provide full I/O capabilities for `ParameterLists` in XML format.

6 Model Evaluator

A model evaluator is a concept implemented in both EpetraExt and Thyra that allows the generic definition of a model that can then be passed to a variety of solvers, including nonlinear solvers, continuation algorithms and optimization packages. The `ModelEvaluator` class in EpetraExt is now supported in PyTrilinos.

The `ModelEvaluator` class is a virtual base class with a `modelEval()` method that must be implemented by a derived class. A solver or optimization object that has a ModelEvaluator interface

will call this `modelEval()` method whenever a model evaluation is required.

SWIG supports the use case of writing such a derived class in Python, including the `modelEval()` method. This is known as cross-language polymorphism and works as follows: SWIG generates a C++ class that inherits from the `ModelEvaluator` class that provides compilable code for all virtual methods. Such a class is known as a director class and such methods are known as director methods. This is because the class is wrapped with a Python interface and that interface is checked dynamically for implementations of the director methods. If such Python methods are found, the underlying C++ methods "direct" execution to the Python code. If not, the default C++ implementation will be called, or for a pure virtual method, an exception raised.

The Python interface for the EpetraExt `ModelEvaluator` class was a challenge to develop because the `ModelEvaluator` makes heavy use of nested classes. This is the one example of C++ code that is not supported by SWIG. There are workarounds for this situation suggested by SWIG developers, but the `ModelEvaluator` class is complex enough that it resisted these workarounds. The ultimate solution was to redefine the nested classes as non-nested pure Python classes and then write typemaps that convert these Python arguments to the appropriate underlying C++ types.

`ModelEvaluator` classes are useful only if there are corresponding solver objects which support the ModelEvaluator interface. To date only the NOX nonlinear solver package supports the ModelEvaluator interface within PyTrilinos.

7   Vision for Second Generation Packages

First generation Trilinos packages are designed around the Epetra package, a package that provides distributed linear algebra classes such as vectors, multivectors, operators, and sparse matrices. Development of Epetra began over a decade ago, when variations in C++ compilers from platform to platform were considerable. For this reason, Epetra was designed without templates or namespaces, to maximize portability. As a result, Epetra scalar data is always double and ordinal types are always ints. In the intervening decade, needs have inevitably arisen for complex and single precision

scalar data (especially for GPUs) and long ordinal data (as global problem sizes have grown). To address these needs, a second generation linear algebra services package, Tpetra (Templated Petra) has been developed, with template arguments for scalar data, local ordinals and global ordinals.

PyTrilinos provides wrappers to a significant subset of first generation packages. This includes Teuchos, which has evolved to support both first and second generation packages, and Anasazi, which was designed from its origins to both utilize templates and interface to Epetra. But Tpetra does not yet have a PyTrilinos interface, and so most of the second generation packages do not either.

C++ templates present a challenge to designing Python interfaces that wrap such code. This stems from the fact that C++ implements templates with a heavy dependence on syntax while the same concept is implemented in Python without syntax. In other words, because Python function and method arguments do not specify types, any argument type will work as long there is support for all of the operators and functions applied to that argument. Thus Python "templates" are implicit and implemented at run time, while C++ implements templates at compile time. As a Python interface to compiled code, Python extensions must link to a finite set of concrete instantiations of C++ template code, which makes this use case far less generic than pure Python is.

The SWIG method for generating wrapper code for templates is to parse template code internally but only produce wrappers for concrete instantiations, each one requiring a unique name. As a simple example of this, consider an attempt to wrap the C++ `Tpetra::Vector` class. We will focus on a subset of a significant simplification of the class that will expose wrapping issues of concern:

```
namespace Tpetra {

    template< class T >

    class Vector {

        Vector(size_type n = 0);

    };

}
```

16

We can instantiate Tpetra::Vector with any type T, and this will produce an array of data of type T. We would like a Python interface that gives us access to this class with a variety of types supported. To wrap this class with SWIG, one might include the following in the SWIG interface file:

```
%template(Vector_int    ) Tpetra::Vector<int>;
%template(Vector_long   ) Tpetra::Vector<long>;
%template(Vector_float  ) Tpetra::Vector<float>;
%template(Vector_double ) Tpetra::Vector<double>;
```

This would produce a Python extension module with definitions for four independent classes:

```
from PyTrilinos import Tpetra
n = 10
vi = Tpetra.Vector_int(n)
vl = Tpetra.Vector_long(n)
vf = Tpetra.Vector_float(n)
vd = Tpetra.Vector_double(n)
```

This interface is decidedly not "pythonic". It is also not scalable; C++ code that utilizes multiple template parameters would quickly result in names that are too long or too cryptic to be usable or readable.

A preferable interface would consist of a single Python class named Vector. This would require the addition of a technique to specify the scalar data type. The NumPy module has already addressed this issue with the utilization of `dtype` arguments, and so a similar solution would be both familiar to scientific Python programmers and compatible with NumPy:

```
from PyTrilinos import Tpetra
n = 10
```

```
vi = Tpetra.Vector(n, dtype="i")

vl = Tpetra.Vector(n, dtype="l")

vf = Tpetra.Vector(n, dtype="f")

vd = Tpetra.Vector(n, dtype="d")
```

To accomplish this in SWIG would require the development of a complete new class in C++ that is not itself templated, but is capable of internally storing several templated Vector classes of different (predefined) types, only one of which would be active with allocated data. With this paradigm, SWIG loses some of its advantages. We are no longer targeting a nearly one-to-one interface between C++ and Python and must spend considerable effort designing a new interface. This interface definition makes other wrapping tools, such as SIDL/Babel, Boost.Python or Cython much more attractive. Cython in particular might be particularly adept at helping to develop "pythonic" interfaces.

This raises the question of whether interfaces developed with SWIG can be compatible with interfaces developed with another tool, most likely Cython. If so, Cython may be an attractive approach for wrapping second generation Trilinos packages. If not, the sheer inertia of SWIG-wrapped PyTrilinos packages may dictate continuing to use SWIG to generate the Python interfaces. Clearly, some research must be conducted to determine the best path forward for developing second generation Trilinos packages for PyTrilinos.

8   Concluding Remarks

The Trilinos Project is now over a decade old and has seen massive growth in that time. Beginning with three packages for linear algebra services, iterative solvers and preconditioners, Trilinos has grown to now encompass 46 packages in its current release, with more planned for the future. It has grown from a suite of solver technologies to a suite of simulation tools, now include meshing, discretizations, partitioning, load balancing, automatic differentiation, optimization and much more. As C++ compilers have matured, more advanced programming techniques have been employed.

18

As lessons have been learned regarding object oriented programming for scientific computing, those lessons have been deployed in the code base.

These advances, coupled with the need for stable interfaces, have inevitably led to the development of second generation packages such as Tpetra, Belos and Ifpack2. PyTrilinos provides Python interfaces to first generation packages, with an emphasis on Epetra and those packages that process Epetra objects. As the second generation packages mature, we expect pressure to mount to provide Python interfaces to them. Tpetra vectors, with their support of multiple data types, are more powerful than Epetra vectors, with their restriction to double precision. This makes Tpetra vectors more like NumPy arrays, with the added capability of distribution over parallel computing architectures. Thus Tpetra and the packages that use it are obvious candidates for Python wrappers. So while PyTrilinos currently provides an impressive set of capabilities, it is also at a crossroads, facing a large new set of potential capabilities to provide in the form of second generation Trilinos packages.

References

[1] C. G. Baker, U. L. Hetmaniuk, R. B. Lehoucq, and H. K. Thornquist. Anasazi software for the numerical solution of large-scale eigenvalue problems. *ACM Transactions on Mathematical Software (TOMS)*, 36(3), July 2009.

[2] Satish Balay, Jed Brown, , Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Barry F. Smith, and Hong Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

[3] Boost.Python. http://www.boost.org/doc/libs/1_46_1/libs/python/doc/.

[4] Andrew Cleary, Scott Kohn, Steven G. Smith, and Brent Smolinski. Language interoperability mechanisms for high-performance scientific applications, presented at SIAM Workshop on Object-Oriented Methods for Inter-operable Scientific and Engineering Computing. Technical

Report UCRL-JC-131823, Lawrence Livermore National Laboratory, Yorktown Heights, NY, 1998.

[5] SciPy Community. *SciPy Reference Guide*, 2011.

[6] T. L. Cottom. Using SWIG to bind C++ to Python. *Computing in Science & Engineering*, 5(2), 2003.

[7] Doxygen. http://www.stack.nl/ dimitri/doxygen/.

[8] John D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3), 2007.

[9] Matlab - the language of technical computing. http://www.mathworks.com/products/matlab/.

[10] Travis Oliphant. *Guide to NumPy*. Trelgol Publishing, 2008.

[11] M. Sala, W. Spotz, and M. Heroux. PyTrilinos: High-performance distributed-memory solvers for Python. *ACM Transactions on Mathematical Software (TOMS)*, 34, March 2008.

[12] SciPy Conferences. http://conference.scipy.org/.

[13] Dag Sverre Seljebotn. Fast numerical computations with cython. In Gaël Varoquaux, Stéfan van der Walt, and Jarrod Millman, editors, *Proceedings of the 8th Python in Science Conference*, pages 15–22, Pasadena, CA, 2009.

[14] SIAM CS&E Conferences. http://www.siam.org/meetings/archives.php#CS.

[15] The SC Conference Series. http://www.supercomp.org/.