

PyTrilinos Developers Guide

Bill Spatz
wfspotz@sandia.gov
Sandia National Laboratories

28 October 2010

Copyright: Sandia Corporation, 2010

Abstract

PyTrilinos is a python interface to selected Trilinos packages. The Trilinos Project is a collection of over 30 software packages written primarily in C++ that provide linear-, nonlinear-, and eigen-solvers, along with preconditioners and supporting utilities, that are object-oriented, parallel and serial, for sparse and dense problems. PyTrilinos is one of those packages, and provides python interfaces to the most popular and important Trilinos packages. This Guide provides information, both necessary and recommended, for developing PyTrilinos packages.

Contents

Introduction to PyTrilinos Development	2
Adding New Modules to PyTrilinos	3
The PyTrilinos Build System	4
Single Namespace Packages	4
Nested Namespace Packages	5
The pytrilinos Library	6
Package-Specific Configuration Options	6
PyTrilinos Documentation System	6
Python Docstrings	6
PyTrilinos Docstring Policy	6
Module docstrings	7
Function and Method Docstrings	7
Appending Doxygen Documentation to Docstrings	8
Overriding Doxygen Documentation	8
Exception Handling in PyTrilinos	9
Practical Considerations	11
Handling C-Array Arguments	11

Reference Counted Pointers	12
PyTrilinos and RCPs	12
When to Store PyTrilinos Classes as RCPs	13
Usage Details	14
Testing	14
Naming Conventions	14
Build System	14
Running All Tests	15
Test Script Conventions	15
Unit Tests	16
Example Scripts	16

Introduction to PyTrilinos Development

The purpose of PyTrilinos is to provide a high-level, dynamic and interactive interface to selected Trilinos packages. Being a PyTrilinos developer requires that you are, or will become, an expert in the following areas:

- **C++.** The vast majority of Trilinos development is in C++, and much of that development takes advantage of advanced C++ or object-oriented techniques such as polymorphism, templating and reference-counted memory management. A high level of comfort with C++ is required.
- **Python.** Obviously, PyTrilinos is a package of python modules, so familiarity with the language is a must. Beyond that, however, it is important to understand the philosophical underpinnings of the language (which are very different from C++) in order to develop interfaces that are “pythonic”, i.e. that are consistent with the many conventions of the language.
- **Trilinos.** At a bare minimum, a PyTrilinos developer should have a decent level of expertise regarding the Trilinos package that is to be wrapped, as well as those packages that the package interacts with. This expertise would include related development tools such as the CMake build system and the GIT version control system.
- **SWIG.** SWIG is the Simple Wrapper and Interface Generator, and is the workhorse for generating the code that gets compiled to become python extension modules. The ‘S’ in SWIG comes from the fact that SWIG can read and parse a header file and generate code that will compile, can be imported into a python shell, and mimics the C/C++ interface (within the constraints of the language). However, SWIG is very powerful, and almost all of its provisions can be overwritten by the python interface developer. Trilinos is sufficiently complex that such overrides are commonplace, and the PyTrilinos developer must have a high degree of comfort doing this.
- **The Python C/API.** When such overrides are necessary, it usually involves writing code that is compliant with the Python C/API. In addition to the structs and function prototypes provided by the Python C/API, this means a thorough understanding of python exception handling (essential for a dynamic language) and reference counting (which requires more programmer intervention in C than in C++).

- **NumPy.** NumPy is the accepted standard python package for handling arrays of contiguous (or strided) homogeneous data. Any PyTrilinos classes or arguments that involve arrays of data should provide an interface that is highly compatible with NumPy.

Each of these languages, libraries, tools and interfaces have extensive documentation available on-line, and should be accessed frequently by the PyTrilinos developer. The purpose of this Developers Guide is detail how these tools are used together to generate PyTrilinos, and to document certain conventions that have been adopted for all PyTrilinos packages.

Adding New Modules to PyTrilinos

If you want to add a Trilinos package as a new module in PyTrilinos, here are the necessary steps.

1. Add the package name to the `cmake/Dependencies.cmake` file, where variable `LIB_OPTIONAL_DEP_PACKAGES` is set. It is important that packages be listed in build order.
2. If your package uses nested namespaces, then the python interface should use corresponding nested modules. To partially facilitate this, within the `src/PyTrilinos` directory, make a directory with your package name. Repeat for nested namespaces.

In `src/PyTrilinos/CMakeLists.txt`, you will find logic such as:

```
# The NOX module employs namespaces, so include the NOX directory
IF (${PACKAGE_NAME}_ENABLE_NOX)
  ADD_SUBDIRECTORY (NOX)
ENDIF (${PACKAGE_NAME}_ENABLE_NOX)
```

Add similar logic for your code. In each nested directory, create a new `CMakeLists.txt` file and provide similar logic.

3. In the top-level `CMakeLists.txt` file, you will find a series of if-blocks that look like this:

```
IF (PyTrilinos_ENABLE_Teuchos)
  APPEND_SET (${PACKAGE_NAME}_PACKAGES Teuchos)
  APPEND_SET (${PACKAGE_NAME}_MODULES Teuchos)
ENDIF (PyTrilinos_ENABLE_Teuchos)
```

Add a similar if-block for your new package, making sure it is placed in build-order relative to the other packages. See the logic for the NOX package if your package supports nested namespaces (sub-modules).

4. If your package will require compiled code that will be archived in the pytrilinos shared library, add the headers and sources to `src/CMakeLists.txt`, using the existing if-blocks as a guide.
5. If your package supports nested namespaces (sub-modules), then in `src/CMakeLists.txt`, find the loop prefaced with the comment:

```
# Loop over the PyTrilinos-enabled Trilinos modules and define the
# swig-generated PyTrilinos extension modules
```

Use the existing if-blocks to add logic to support your sub-modules.

6. Add your package by writing the required SWIG interface files. For a standard PACKAGE, it will be in the file

```
PACKAGE.i
```

For a package that supports nested namespaces, the primary SWIG interface file will be named:

```
`PACKAGE__init__.i`
```

In both cases, of course, PACKAGE will be replaced with your package name.

The PyTrilinos Build System

The move by Trilinos to the CMake build system has had many advantages for PyTrilinos. The two primary advantages being that shared libraries and python extension libraries can now both be built automatically by the host build system. Previously, shared libraries were built by the PyTrilinos package in an ad-hoc manner, and the python extension libraries were built using the python distutils module, which has some reports of portability issues. The move to CMake has improved the reliability and robustness of the PyTrilinos build system.

From a build-system point of view, there are two types of Trilinos packages that are supported by PyTrilinos: those that are contained within a single namespace, like Teuchos, Epetra and AztecOO; and those that are contained within nested namespaces, such as NOX. These two types are detailed below, in addition to the pytrilinos library and package-specific configuration options.

Single Namespace Packages

All supported packages must be listed in the LIB_OPTIONAL_DEP_PACKAGES variable in the cmake/Dependencies.cmake file, and they should be listed in build order.

The PyTrilinos build system maintains two variables, PyTrilinos_PACKAGES and PyTrilinos_MODULES, both set in the top-level CMakeLists.txt file, to keep track of what to build. (Note that these variables are referenced as \${PACKAGE_NAME}_PACKAGES and \${PACKAGE_NAME}_MODULES within CMakeLists.txt.) For single namespace packages, the entries in these two variables is the same. For example,

```
IF (PyTrilinos_ENABLE_Teuchos)
  APPEND_SET (${PACKAGE_NAME}_PACKAGES Teuchos)
  APPEND_SET (${PACKAGE_NAME}_MODULES Teuchos)
ENDIF (PyTrilinos_ENABLE_Teuchos)
```

In this instance, the build system now expects to find a SWIG interface file in the source directory src/Teuchos.i that defines module PyTrilinos.Teuchos. The end products will be placed in the build directory src/PyTrilinos:

```
src/PyTrilinos/Teuchos.py
src/PyTrilinos/Teuchos.pyc
src/PyTrilinos/_Teuchos.so
```

Nested Namespace Packages

Using NOX as an example, the `LIB_OPTIONAL_DEP_PACKAGES` variable in the `cmake/Dependencies.cmake` file should contain the entry NOX and it should be listed in build order.

In the top-level `CMakeLists.txt` file, the variable `PyTrilinos_PACKAGES` should be appended as before, with a single entry. But `PyTrilinos_MODULES` should contain an entry for each nested namespace:

```
IF (PyTrilinos_ENABLE_NOX)
  APPEND_SET (${PACKAGE_NAME}_PACKAGES NOX)
  APPEND_SET (${PACKAGE_NAME}_MODULES NOX.__init__ )
  APPEND_SET (${PACKAGE_NAME}_MODULES NOX.Abstract )
  APPEND_SET (${PACKAGE_NAME}_MODULES NOX.StatusTest)
  APPEND_SET (${PACKAGE_NAME}_MODULES NOX.Solver )
  IF (NOX_ENABLE_Epetra)
    APPEND_SET (${PACKAGE_NAME}_MODULES NOX.Epetra.__init__ )
    APPEND_SET (${PACKAGE_NAME}_MODULES NOX.Epetra.Interface)
  ENDIF (NOX_ENABLE_Epetra)
ENDIF (PyTrilinos_ENABLE_NOX)
```

For every entry in `PyTrilinos_MODULES`, there should be a corresponding SWIG file in the source directory:

```
src/NOX.__init__.i
src/NOX.Abstract.i
src/NOX.StatusTest.i
src/NOX.Solver.i
src/NOX.Epetra.__init__.i
src/NOX.Epetra.Interface.i
```

which will produce the following build products:

```
src/PyTrilinos/NOX/__init__.py
src/PyTrilinos/NOX/__init__.pyc
src/PyTrilinos/NOX/__init__.so
src/PyTrilinos/NOX/Abstract.py
src/PyTrilinos/NOX/Abstract.pyc
src/PyTrilinos/NOX/_Abstract.so
src/PyTrilinos/NOX/StatusTest.py
src/PyTrilinos/NOX/StatusTest.pyc
src/PyTrilinos/NOX/_StatusTest.so
src/PyTrilinos/NOX/Solver.py
src/PyTrilinos/NOX/Solver.pyc
src/PyTrilinos/NOX/_Solver.so
src/PyTrilinos/NOX/Epetra/__init__.py
src/PyTrilinos/NOX/Epetra/__init__.pyc
src/PyTrilinos/NOX/Epetra/___init__.so
```

```
src/PyTrilinos/NOX/Epetra/Interface.py
src/PyTrilinos/NOX/Epetra/Interface.pyc
src/PyTrilinos/NOX/Epetra/_Interface.so
```

The pytrilinos Library

If you develop a module that requires compiled code not generated by SWIG, it should be put in the `pytrilinos` library. Simply append entries the the `HEADERS` and `SOURCES` variables in `src/CMakeLists.txt` file.

Package-Specific Configuration Options

If you need package-specific configuration options set, they should be done so in `src/CMakeLists.txt` prior to the call to `PACKAGE_CONFIGURE_FILE()` and in `cmake/PyTrilinos_config.h.in`. Currently, the following variables are set depending upon the top-level Trilinos configuration:

```
HAVE_EPETRA
HAVE_TEUCHOS
HAVE_NOX_EPETRA
HAVE_NOX_EPETRAEXT
HAVE_MPI
```

PyTrilinos Documentation System

Python Docstrings

Python has an effective interactive documentation system, utilizing what are known as “docstrings”. If the first statement of a python module, a python function or a python class is a string constant, then that string becomes the documentation string, or “doc-string”, for the given module, function or class. For example,

```
def func(arg):
    "Docstring for function func"
    return "Hello, world!"
```

Docstrings are fundamentally different from python comments, because of their position relative to the code they describe and because they are used by the python help facility and other documentation tools, such as `pydoc`:

```
>>> help(func)
Help on function func in module __main__:

func(arg)
    Docstring for function func
>>>
```

PyTrilinos Docstring Policy

It is the policy of PyTrilinos that python docstrings should be maintained to provide useful, current and accurate information to PyTrilinos users that

- correctly reflect the calling arguments of PyTrilinos methods and functions,
- match the corresponding doxygen documentation provided by Trilinos developers when appropriate, and
- supercede the doxygen documentation when the python interface and/or implementation is different from the C++ interface and/or implementation.

The difficulty in adhering to these policies is that the python code (and therefore the python docstrings) that makes up PyTrilinos is automatically generated by swig. Fortunately, there are several tools available that allow these policies to be met.

Module docstrings

Every PyTrilinos module should contain a module docstring. This is typically implemented in the package interface file as follows:

```
%define %my_package_docstring
"
Multi-line documentation string for
My_Package
"
%enddef
%module (docstring = %my_package_docstring) My_Package
```

SWIG does not require that SWIG macro names begin with “%”, but it is allowed and it is the convention for PyTrilinos SWIG macros. Further, PyTrilinos module docstrings follow a set format:

```
PyTrilinos.My_Package is the python interface to the Trilinos
such-and-such package My_Package:
```

```
http://trilinos.sandia.gov/packages/my_package
```

```
The purpose of My_Package is to . . . . The python version of the
My_Package package supports the following classes:
```

```
* Class1          - Short description 1
* Class2          - Short description 2
* ...
```

```
Any other notes about the package as a whole. . . .
```

Function and Method Docstrings

Docstrings for functions and methods are partially implemented by the SWIG %feature called autodoc. Before any code is directed to be wrapped within an interface file (by the first %include directive), the following feature should be activated:

```
%feature("autodoc", "1");
```

This automatically adds docstrings to each function and class method showing the function name, argument list, and return type. The argument list contains both type and name information (as per the "1" option). If a function or method is overloaded, each valid calling signature is listed.

Appending Doxygen Documentation to Docstrings

A system is in place for automatically extracting Doxygen documentation from a Trilinos package source code and inserting it into the package's python module. First, Doxygen is used to extract the documentation and save it in XML format. Second, a python script is run to read this XML data and convert it to a series of valid SWIG statements of the form::

```
%feature("docstring") <symbol-name> <docstring>
```

Third, each package interface file implements an %include statement to read these documentation directives. This adds the Doxygen documentation string to the end of the string created by %feature("autodoc", "1").

All of this infrastructure is contained within the PyTrilinos subdirectory:

```
doc/Doxygen
```

Within this subdirectory are a series of Doxygen control files, one for each package: Doxyfile_Teuchos, Doxyfile_Epetra, etc. These Doxyfiles have certain features in common: they all generate XML output:

```
GENERATE_XML = YES
```

while suppressing all other forms of output. They all direct output to their own subdirectory:

```
OUTPUT_DIRECTORY = Teuchos
XML_OUTPUT = .
```

If you add documentation for a new package, then add the package name to the PyTrilinos/doc/Doxygen/Makefile variable PACKAGES. When you run make from the shell, this will cause doxygen to be run on the Doxyfile and the script doxy2swig.py (generously provided by Prabhu Ramachandran) to be run on the resulting output. This produces a file with a _dox.i suffix, e.g. Teuchos_dox.i, intended to be included into the package interface file src/Teuchos.i. To facilitate this inclusion, SWIG is invoked with -I\$(top_srcdir)/doc/Doxygen.

Overriding Doxygen Documentation

Sometimes the Doxygen documentation is not appropriate for a given python function or method. This is often the case when %typemaps are employed or when the python implementation replaces the C++ interface by using %ignore, %extend and/or %inline. This requires that the automatically created docstrings be overridden or appended, and can be accomplished by providing a SWIG documentation directive *after* the %include "Teuchos_dox.i" (for example) and that redefines the docstring for a specified symbol. These directives should be placed directly in the appropriate interface file. By convention, all directives pertaining to the symbols within

a single `%include` file are grouped together. The documentation directives should be at the top of these groupings.

There are two SWIG documentation directives that can be employed to override or append docstrings. If you wish to append documentation to what is already present, use:

```
%feature("docstring")
<symbol-name>
"
<docstring>
";
```

If you wish to replace the existant docstring completely, use the directive:

```
%feature("autodoc",
"<docstring>")
<symbol-name>;
```

The formatting given here tends to be the most readable for the widest range of situations, both for the interface files and for the resulting docstrings.

Exception Handling in PyTrilinos

All C++ exceptions raised by Trilinos should be caught by PyTrilinos and converted to python exceptions. Fortunately, there is a relatively straightforward SWIG facility for doing this. Each SWIG interface file shall include an `%exception` directive prior to any Trilinos header file `%include` directives:

```
%include "exception.i"
%exception
{
    try
    {
        $action
    }
    catch(...)
    {
        SWIG_exception(SWIG_UnknownError, "Unknown C++ exception");
    }
}
```

It is possible to implement an `%exception` directive that includes a symbol name, prior to the first "{", that is specific to a function or method. By omitting this symbol name, we are applying this `%exception` to *all* functions and methods that get wrapped. Here, `$action` is a SWIG macro that is replaced by the generated code for calling the wrapped function or method. The `catch(...)` syntax ensures that *every* exception that might be thrown gets caught.

`SWIG_exception` is a C macro #define-ed at the top of the generated source file. `SWIG_UnknownError` is also a macro that evaluates to an integer. See the SWIG documentation for valid SWIG error macro names.

The directive given above is useful, but all exceptions will get converted to type `UnknownError` with a nearly meaningless error message. Realistically, we need

to convert a wider range of exceptions to more meaningful python exception types, and produce more useful error messages. SWIG also provides a macro for treating most standard C++ exceptions, converting them to appropriate python exceptions and extracting their error message from their `what()` method. Simply change the `%exception` directive to:

```
%exception
{
    try
    {
        $action
    }
    SWIG_CATCH_STDEXCEPT
    catch(...)
    {
        SWIG_exception(SWIG_UnknownError, "Unknown C++ exception");
    }
}
```

and this will convert the vast majority of standard exceptions to appropriate python exceptions with useful error messages.

There are Trilinos packages that throw exceptions other than those found in the standard library. These can be caught anywhere prior to the `catch(...)` syntax, although in general, it is best to put them before the `SWIG_CATCH_STDEXCEPT` macro, especially if the package exceptions inherit from the standard exceptions. Here is the current Teuchos exception handler:

```
%exception
{
    try
    {
        $action
        if (PyErr_Occurred()) SWIG_fail;
    }
    catch(Teuchos::Exceptions::InvalidParameterType & e)
    {
        SWIG_exception(SWIG_TypeError, e.what());
    }
    catch(Teuchos::Exceptions::InvalidParameter & e)
    {
        PyErr_SetString(PyExc_KeyError, e.what());
        SWIG_fail;
    }
    SWIG_CATCH_STDEXCEPT
    catch(...)
    {
        SWIG_exception(SWIG_UnknownError, "Unknown C++ exception");
    }
}
```

A few notes: (1) After the `$action` macro, there is a call to `PyErr_Occurred()`. This is because the Teuchos wrappers `%extend` certain classes and those new meth-

ods can set python errors. Alternatively, you could raise C++ exceptions in all of these extensions, and then skip the `PyErr_Occurred()` check. (2) `SWIG_fail` is a C macro provided by SWIG that evaluates to `goto fail`, where `fail` is a label that exists within all wrapper functions. (3) The `Teuchos::Exceptions::InvalidParameter` exception is most closely related to the python `KeyError` exception, but SWIG does not have a corresponding SWIG error for this. Therefore, I use the `PyErr_SetString()` function and `SWIG_fail` macro.

Practical Considerations

Most PyTrilinos packages will need to `%import "Teuchos.i"` and/or `%import "Epetra.i"`. Both of these interface files implement their own `%exception` directive, but both of them “turn off” exception handling by including a:

```
%exception;
```

at the end of the file. This is considered good practice and should be followed in *all* PyTrilinos SWIG interface files.

Nevertheless, experience shows that the following represents the best order for `%include-s` and `%import-s` when dealing with exceptions in PyTrilinos:

```
%include "exception.i"
%import "Teuchos.i"
%import "Epetra.i"
%exception
{
    ...
}
```

Putting the `%include "exception.i"` after the `%import` directives can result in undefined symbols when you compile the generated wrapper code.

Finally, every effort should be made to prevent users from getting an `Unknown C++ exception` error message. Study the package to determine as many of the possible exceptions that might be thrown as you can, and explicitly include them in the `%exception` directive. Whenever testing or use of PyTrilinos results in an `Unknown C++ exception` error message, track it down and then explicitly allow for it within the appropriate SWIG interface file. There is no excuse for a meaningless error message.

Handling C-Array Arguments

Trilinos defines a number of high-level array-like objects that store contiguous, homogenous data. Nevertheless, there are instances when Trilinos objects pass low-level C-arrays as input or return arguments.

SWIG does not handle this case automatically in the manner we would like. However, PyTrilinos has adopted a set of interface conventions for dealing with them and simple methods for achieving those interfaces.

Built-in python containers, such as `list`, are discontinuous and heterogeneous, which makes them unsuitable for efficiently handling C-array type data. Fortunately, there is a third party module named `NumPy` that has been adopted by the python community for just this purpose. (This adoption has been hard won -- NumPy brought together two

divergent efforts named Numeric and NumArray.) Included in the NumPy distribution is a file named `numpy.i`, which is a SWIG interface file that provides typemaps and other tools. This file has been copied to the PyTrilinos/src directory and is used by PyTrilinos SWIG interface files for handling C-array arguments.

To learn how to use `numpy.i`, its [documentation is online](#).

Code that needs to interface with NumPy should call the function `import_array()`, but only once. To avoid the possibility of two unrelated python modules both calling `import_array()` in a conflicting way, NumPy requires that you define a macro `PY_ARRAY_UNIQUE_SYMBOL`. In PyTrilinos, we do:

```
#define PY_ARRAY_UNIQUE_SYMBOL PyTrilinos
```

However, we must also guard against two or more PyTrilinos modules calling `import_array()`. To do this, we define a singleton class `NumPyImporter` that calls `import_array()` and lives in the `pytrilinos` shared library. All PyTrilinos extension modules link against this library and so the first one to be imported will instantiate the `NumPyImporter` object, which calls `import_array()` in its constructor.

For this reason, the initialization instructions in the `numpy.i` instructions should be ignored, as they are for a single python module environment. Instead, all a PyTrilinos developer has to do is add:

```
%{  
#include "numpy_include.h"  
%}  
%include "numpy.i"
```

to his SWIG interface file, and then start using the `%apply` directive as described in the `numpy.i` documentation.

Reference Counted Pointers

Reference counting is a memory management technique whereby dynamically allocated objects can be tracked, and can be guaranteed to exist as long as any other entity (such as a class or the local scope of a function) needs it to. Such entities can “take a reference” to the dynamically-allocated reference-counted object, and the technique works by ensuring that the object is never destroyed until its reference count falls to zero.

A reference counted pointer is a data structure for implementing such a memory management technique. It consists of a raw pointer to an object and an integer count of the number of entities that have taken a reference to the object. It also consists of methods for increasing and decreasing the reference count as needed, and a destructor that waits until the reference count is zero before destroying the object. These are also called shared pointers.

PyTrilinos and RCPs

Python implements just such a reference counting scheme for every python object that is constructed. It occurs “behind the scenes” and “just works”, like python is famous for. The only people who ever have to worry about python reference counting are

programmers who write code using the Python/C API -- which does include PyTrilinos developers.

C++ does not have a standard reference counting system, although several reference counting classes have been developed, which are much more powerful than can be developed in C (such as the Python/C API uses) and much less prone to error. There is the `boost::shared_ptr<>` class, which is planned to be moved to the `std` namespace at some point in the future. Trilinos could not wait for this standardization process, and thus was born the `Teuchos::RCP<>`, which of course stands for “reference counted pointer”.

Several Trilinos packages use RCPs, and so PyTrilinos has a policy for dealing with them. First, if any package anywhere stores instances of a class using an RCP, then PyTrilinos will always store instances of that class internally using an RCP. This ensures that all reference counts remain accurate under all use cases. Second, python programmers should never have to deal with `Teuchos::RCP<>`. That is to say, there is no `Teuchos.RCP` class in PyTrilinos. If a C++ method requires a `Teuchos::RCP<>` of some class, the python interface will take an unadorned instance of that class. The conversion to an RCP will happen behind the scenes. And the python programmer, who has always benefitted from reference counts without ever exerting any effort, will continue in this happy state.

By default, SWIG generates code that stores dynamically allocated objects using a raw pointer. With SWIG 2.0.0, the python code generator in SWIG has a (relatively?) bug free implementation of using reference counted pointers instead of raw pointers. This is coupled with a large set of typemaps that alter the conversion code between C++ and python, taking the new storage method into account. This is implemented for `boost::shared_ptr<>` and `std::shared_ptr<>`. It provides a `%shared_ptr()` SWIG macro that the user invokes on a class that should be stored as a shared pointer.

The `Teuchos_RCP.i` file leverages the SWIG-provided `boost_shared_ptr.i` file by using `#define` statements to make all of the provided logic work with `Teuchos::RCP<>`. It also provides replacement typemaps for when the `Teuchos::RCP<>` interface is different from the `boost::shared_ptr<>` interface and additional typemaps for directors, which `boost_shared_ptr.i` does not provide. This new logic is accessed by using a `%teuchos_rcp()` SWIG macro on a class.

When to Store PyTrilinos Classes as RCPs

When do you use the `%teuchos_rcp()` macro? First, whenever you encounter a class that is wrapped in `Teuchos::RCP<>` as an input or output argument in a class method or function, then that class needs to be stored internally as as an RCP by using `%teuchos_rcp()`.

Second, if `B` is a base class and `D` derives from it, then `%teuchos_rcp(B)` requires that `%teuchos_rcp(D)` also be invoked. Failure to do so will result in an extension module that will fail to compile. The reason is due to how type checking is performed for these RCP classes and the need for derived classes to be recognized as proper instances of base classes.

Third, using `B` and `D` as before, if `%teuchos_rcp(D)` is invoked, then `%teuchos_rcp(B)` almost certainly should be as well. The extension module will compile even if this rule is not followed, but type checking will fail under certain circumstances. This rule can be ignored if `B` is an implementation-only base class, but this is rare.

Usage Details

If a class is defined in Package 1, but not used as an RCP in Package 1, and is then used as an RCP in Package 2, the `%teuchos_rcp()` macro should be invoked in the SWIG interface file for Package 1. This way, when `%import` is used on Package 1 from other package SWIG interface files, the storage method remains constant among all the packages.

If you are (in effect) wrapping Package 2, and Package 1 is Epetra, then there are some additional SWIG macros you should know. If the Epetra class is an array storage class implemented as a hybrid numpy array, then you should use the `%teuchos_rcp_epetra_numpy()` macro. This should never be necessary, however, because all of these classes have already been treated.

Testing

There are two directories in the PyTrilinos package that provide python scripts for testing PyTrilinos, `example` and `test`. Generally speaking, unit tests go in the `test` directory and descriptive or demonstrative example scripts go in `example`.

Naming Conventions

Unit tests scripts shall begin with `test`, followed by the package name and an underscore, followed by a short description of the test, typically the class name being tested. For example:

```
testTeuchos_ParameterList
testEpetra_MultiVector
```

are the base names of the unit tests for `Teuchos.ParameterList` and `Epetra.MultiVector`, respectively. In certain situations, the underscore and test description can be omitted.

Example scripts shall begin with `ex`, followed by the name of the primary package being demonstrated, followed by an underscore and a short description of the test. For example:

```
exEpetra_Comm
exAztecOO_Operator
```

are the base names for example scripts demonstrating the `Epetra.Comm` class and an `AztecOO` solver that uses an `Epetra.Operator`, respectively. In certain situations, the underscore and test description can be omitted.

Build System

Running `make` in either the `top`, `example` or `test` build directories copies the test scripts from the source directory to the build directory while performing certain text substitutions. Similar to the configuration file naming convention, the source files have the suffix `.py.in` and the build files have the suffix `.py`. The reason for this is that CMake variable values can be substituted during the copy procedure. For example, the first line of each test script source file is now:

```
#! ${PYTHON_EXECUTABLE}
```

which means that the python executable that is recognized (and compiled against) by the CMake build system will also be the python executable invoked by the PyTrilinos test scripts. Note that this substitution is available to the test developers for any variable that is defined in the `CMakeCache.txt` file found in the top build directory.

The `CMakeLists.txt` files in the `test` and `example` directories control which scripts get copied to the build directory. Each test script, or group of related test scripts, should be protected with `IF()` statements, depending on which PyTrilinos modules need to be present for the script to run.

Running All Tests

To run all of the enabled tests, first make sure PyTrilinos and all tests and examples are up-to-date:

```
$ cd BUILD/packages/PyTrilinos
$ make
```

Then you can use the CMake `ctest` program to run all of the tests:

```
$ ctest -W 10
```

The results of all of the tests can be found in the `Testing` directory, present in the directory from which `ctest` was run.

Test Script Conventions

All test scripts shall use the `optparse` module to parse command line options and support the following options:

```
-t, --testharness      test local build modules; prevent loading
                        system-installed modules
-v VERBOSITY, --verbosity=VERBOSITY
                        set the verbosity level [default 2]
```

The `-t` option is to force use of the locally-built PyTrilinos, preventing the importing of any installed PyTrilinos modules. The verbosity option is used in all test scripts and optionally in any example scripts where it makes sense.

Tests scripts use the `from PyTrilinosImport` function in the `testutil` module, local to both `test` and `example` directories, to control where the PyTrilinos modules are imported from. The user controls this import location from the command line: `-t` or `--testharness` indicates that the build directory should be used; otherwise an import from standard locations will be attempted.

This policy is enabled by code in each test/example script like the following:

```
parser = OptionParser()
parser.add_option("-t", "--testharness", action="store_true",
                  dest="testharness", default=False,
                  help="test local build modules; prevent loading system-in
options,args = parser.parse_args()
from testutil import fromPyTrilinosImport
Teuchos = fromPyTrilinosImport('Teuchos', options.testharness)
Epetra   = fromPyTrilinosImport('Epetra' , options.testharness)
```

If the user specifies `-t` or `--testharness` then `options.testharness` will be `True`, else it will be `False`. When `fromPyTrilinosImport()` is called, the `options.testharness` argument will determine where the import is read from.

Test scripts shall run in both serial or parallel. You may use either:

```
comm = Teuchos.DefaultComm.getComm()
```

or:

```
comm = Epetra.PyComm()
```

to obtain an appropriate communicator object for the test scripts. By convention, set the variable `iAmRoot` to either `True` or `False` depending on whether the communicator's rank is 0.

The test script shall output `End Result: TEST PASSED` if the test passes correctly. This helps the Trilinos test harness determine which tests pass and which do not, especially in parallel.

Unit Tests

Unit tests are based on the `unittest` python library module. Test case classes inherit from `unittest.TestCase`. Individual tests are implemented as methods of these classes that begin with `test`. See the python documentation (<http://www.python.org>) for details.

Each unit test script can have one or more `TestCase` classes. In `main()`, each test case class should be added to a `unittest.TestSuite` object named `suite`.

Unit tests shall print, from processor 0, a header with a message such as "Testing Epetra.Object" with a line of asterisks above and below the message:

```
*****
Testing Epetra.Object
*****
```

The suite of tests should be run with:

```
verbosity = options.verbosity * int(iAmRoot)
result = unittest.TextTestRunner(verbosity=verbosity).run(suite)
```

and the success should be determined and output via (for the case of an Epetra communicator):

```
errsPlusFails = comm.SumAll(len(result.errors) + len(result.failures))
if errsPlusFails == 0 and iAmRoot: print "End Result: TEST PASSED"
sys.exit(errsPlusFails)
```

If there are no errors and no failures on any processors, then the test will pass.

Example Scripts

Example scripts are more free-form and should be written for readability, to make for clear demonstrations of PyTrilinos usage. However, it is encouraged that example script output be consistent with unit test output whenever possible.