Lab 1

Interfacing With C++ Using Cython

Lab Objective: *Learn to interface with object files using Cython. This lab should be worked through on a machine that has already been configured to build Cython extensions using gcc or MinGW.*

Interfacing with C++

Cython can also be used to wrap C++ functions and classes. It can compile entire modules to C++ instead of C if we specify C++ as the language in the setup file. Here we will focus primarily on wrapping C++ classes since wrapping a C++ function is roughly the same as wrapping a function from C or Fortran. Cython also supports wrapping templated classes and functions. Many other C++ features like operator overloading and the `new` and `del` operators are also supported when using C++.

The process for exposing a C++ class to Cython is essentially the same. The C++ code must first be compiled to an object file. In our Cython file we then declare that we will be importing functions and classes from a C++ file so that the Cython compiler knows how they will behave. We then compile our Cython file by first compiling to C++

Included with this lab are three C++ classes. The `Permutation` class is the primary class we will be wrapping. It implements arithmetic for the Cauchy cycle representation of permutations. For those not familiar with permutation arithmetic, there are some examples in the file `unittest.py`. The other two classes are used in the construction of the `Permutation` class. Because the Permutation class depends on the `Node` and `Cycle` classes, they must all be compiled together. Here is the `setup.py` to set up the wrapper for the `Permutation` class.

```
# Import needed setup functions
from distutils.core import setup
from distutils.extension import Extension
# This is part of Cython's interface for distutils.
from Cython.Distutils import build_ext
# We still need to run one command via command line.
from os import system
```

```
8
# Compile the .o files we will be accessing.
10 # This is independent of the process to build
# the Python extension module.
12 system("g++ -fPIC -c Permutation.cpp Cycle.cpp Node.cpp")

14 # Tell Python how to compile the extension.
# Notice that we specify the language as C++ this time.
16 ext_modules = [Extension(
                        # Module name:
18                      "pypermutation",
                        # Cython source files:
20                      sources=["pypermutation.pyx", "Permutation.pxd"],
                        # Language:
22                      language="c++",
                        extra_compile_args=["-fPIC", "-O2"],
24                      # Other files needed for linking:
                        extra_link_args=["Permutation.o","Cycle.o","Node.o"])]
26
# Build the extension.
28 setup(
    name="pypermutation",
30  cmdclass = {'build_ext': build_ext},
    ext_modules = ext_modules)
```

pypermutation_setup.py

Warning

When wrapping C++ classes, templates, etc. you should always make sure that you specify the language as C++ as we have shown in this setup file.

We need to tell the Cython compiler how to interface with the C++ class. This can be done in much the same way that we defined the interfaces for C and Fortran functions, but we now need to declare that we are importing a class with several functions that act on it. We will include the declarations corresponding to this class in a `.pxd` file. `.pxd` are called *declaration* files. Though declarations are, strictly speaking, not "Cython headers", they can be used in similar ways. Here we will declare the Permutation class and some of its public methods so that we can use them in Cython.

```
1 # Import the needed classes from the standard template library.
2 from libcpp.vector cimport vector
  from libcpp.string cimport string
4 from libcpp cimport bool

6 # Shorten the type identifier for unsigned integers.
  # This is just for convenience in the code.
8 ctypedef unsigned int uint

10 # Here we declare that we will be importing external
  # functions and/or classes from the header "Permutation.hpp".
12 cdef extern from "Permutation.hpp":
      # This line declares that we are importing a C++ class.
14    cdef cppclass Permutation:
```

```
        # Here we define the constructors.
        # We use the nogil keyword to tell Cython that
        # the global interpreter lock should not apply to these
        # functions.
        # The constructor method also needs the extra keyword 'except +'.
        # This tells Cython to raise a proper Python exception if
        # a C++ error is raised.
        Permutation() nogil except +
        Permutation(vector[vector[uint]]& cycles) nogil except +
        # From here on we declare the other methods we may be
        # interested in using in Cython.
        bool verify() nogil
        string get_string() nogil
        uint trace(uint index) nogil
        Permutation* inverse() nogil
        uint get_max() nogil
        uint get_size() nogil
        void reduce() nogil
        Permutation* operator*(Permutation& other) nogil
```

Permutation.pxd

This `.pxd` file can be included in our Cython file with the line `from Permutation cimport Permutation`. Cython comes with several pre-built declaration files. These declaration files are good examples of the proper syntax for wrapping C++ class templates. Some of these declaration files allow Cython code to interface with containers and other classes from the C++ standard library like `deque`, `list`, `map`, `pair`, `queue`, `set`, `stack`, `string`, `utility`, and `vector`. Each of these declaration files can be loaded by using something like

```
from libcpp.vector cimport vector
```

Cython also supports some kinds of automatic conversion from Python data types to standard library containers. See the Cython documentation for more information on interfacing with C++ containers.

As we mentioned before, the general approach for wrapping a C++ class is to make a Cython class that wraps a pointer to the C++ class. Here is an example of how this is done.

```
# Import the declaration for the C++ Permutation class from
# the Permuation declaration file.
from Permutation cimport Permutation
# Also import the vector template.
# This is a necessary part of the interface for the Permuation class.
from libcpp.vector cimport vector

# Shorten the type identifier for unsigned integers.
# This is just for convenience in the code.
ctypedef unsigned int uint

# Here we define the Cython wrapper class.
cdef class PyPermutation:
    # Make the wrapper class store a pointer to an instance
    # of the C++ class we are wrapping.
    cdef Permutation* thisptr

    # Here is the initialization step.
```

```
    # This is a special Cython function that is
    # called when one of the objects for the wrapper
    # class is initialized.
    # This is where we deal with all necessary initialization.
    # It is somewhat analogous to a constructor for a C++ class.
    def __cinit__(self, li=None):
        cdef vector[vector[uint]] pre_permutation
        cdef vector[uint] pre_cycle
        # Here we perform some basic checking for consistency
        # in the input and also build the vectors that will
        # be used to construct the Permutation object.
        if isinstance(li, (list, tuple)):
            # works assuming li is a list of lists of ints.
            for cycle in li:
                for index in cycle:
                    if index < 0:
                        raise ValueError("index must be nonnegative")
                    pre_cycle.push_back(index)
                pre_permutation.push_back(pre_cycle)
                pre_cycle.clear()
            self.thisptr = new Permutation(pre_permutation)
            # add verify method for Permutation.
            # verify here.
            # then reduce.

    # This is another special Cython function that is
    # called when one of these wrapper classes is deallocated.
    # Here we take care of all necessary deallocation steps.
    # In this case, all necessary deallocation is taken care of
    # by the destructor method for the Permutation object,
    # so all we have to do is use the del operator to deallocate it.
    def __dealloc__(self):
        del self.thisptr

    # Here we define how the object is printed.
    # Notice the automatic conversion from a
    # C++ string to a Python string.
    def __str__(self):
        return self.thisptr.get_string()

    # This returns the index of a given index under the permutation.
    def trace(self, index):
        if index < 0:
            raise ValueError("index must be nonnegative")
        return self.thisptr.trace(index)

    # This returns the preimage of a given index under the permutation.
    def trace_inverse(self, index):
        raise NotImplementedError

    # This computes and returns a PyPermutation object that wraps
    # a C++ Permutation object representing the inverse of this permutation.
    def inverse(self):
        inv = PyPermutation()
        inv.thisptr = self.thisptr.inverse()
        return inv

    # Here we define some basic methods for viewing attributes of
    # the Permutation object.
    # This returns the maximum index modified by the permutation.
```

```
    def get_max(self):
        return self.thisptr.get_max()

    # This returns the minimum index modified by the permutation.
    def get_min(self):
        raise NotImplementedError

    # This returns the number of cycles currently stored in the permutation.
    def get_size(self):
        return self.thisptr.get_size()

    # This method reduces the Permutation to its disjoint cycle form.
    def reduce(self):
        self.thisptr.reduce()

    # This method performs multiplication between two Permutation objects.
    def __mul__(PyPermutation self, PyPermutation other):
        # Raise an error if the user tries to multiply this object by
        # anything other than another PyPermutation object.
        if not isinstance(other, PyPermutation):
            return NotImplemented

        # Start by initializing a new PyPermutation object
        # containing a null pointer.
        cdef PyPermutation result = PyPermutation()
        # Set its 'thisptr' attribute equal to a pointer
        # to the result of multiplication between the two
        # Permutation objectst we are multiplying.
        # Notice that we dereference the pointers to perform
        # the multiplication, but that, as defined in the Permutation header,
        # the multiplication operator returns a pointer to the result.
        result.thisptr = (self.thisptr[0]) * (other.thisptr[0])
        # Return the PyPermutation object we have just constructed.
        return result

    def __pow__(PyPermutation self, int p, z):
        raise NotImplementedError
```

pypermutation.pyx

Notice that we included the declaration file for the class.

Problem 1. The Permutation class includes methods to trace an index through the inverse permutation, get the minimum index that is changed by the permutation, and raise the permutation to a power. The declarations for these functions are commented out in the `pypermutation.pyx` file. Uncomment the declarations and add in the code necessary so that these methods are also available in Python. These functions are declared in the header `Permutation.hpp`, but they are not declared in the declaration file `Permutation.pxd`. Add the proper declarations to the declaration file.

Note: the argument z that is included in the power function for the Permutation class is a mandatory argument for the exponentiation operator in Cython. It tells the computer to compute the power of a number mod another number. In this case, raise an error if z is not `None`.

Look at the `__mul__` method for an example of how to construct and return a new PyPermutation object. Keep in mind that the `power` method of the `Permutation` class in C++ returns a pointer to a new `Permutation` object.