hbspy - A Python Interface to the Hierarchical B-spline C++ Library

Spencer Lyon

August 2, 2013

A senior capstone project submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Bachelor of Science

Dr. Derek Thomas, Advisor

Department of Physics and Astronomy

Brigham Young University

August 2013

ABSTRACT

hbspy - A Python Interface to the Hierarchical B-spline C++ Library

Spencer Lyon
Department of Physics and Astronomy
Bachelor of Science

I describe the creation of a Python interface to the HBS C++ library. HBS stands for hierarchal B-splines and the C++ library is used to represent surfaces or volumes of arbitrary complexity in terms of hierarchal splines. This library is under active development by BYU faculty in the Physics and Engineering departments. I will defend the choice of using Python as the high-level interface. I will also describe projects that facilitate wrapping compiled languages (like C, C++ or Fortran) in Python. Among them are SWIG, Boost.Python, Cython, and a relatively new project – XDress. XDress blends an expressive typesystem, C/C++ source code parsers, and code generating utilities into an easy to use system for constructing Python wrappers for C or C++ code via Cython.

Keywords: Python, C++, algebraic geometry, B-splines

Contents

1 Introduction **1**
- 1.1 Background . . . 1
- 1.2 Motivation . . . 3
- 1.3 Context . . . 3

2 Methods **3**
- 2.1 SWIG . . . 4
  - 2.1.1 SWIG Usage Example . . . 5
- 2.2 Boost.Python . . . 8
- 2.3 Cython . . . 9
  - 2.3.1 Cython Type Example . . . 10
  - 2.3.2 Cython Wrapping Example . . . 13
- 2.4 XDress . . . 18
  - 2.4.1 `xdressrc.py` . . . 18
  - 2.4.2 XDress Plugins . . . 20

3 Results and Discussion **22**
- 3.1 XDress and HBS . . . 22

4 Conclusion **22**

A HBS (C++) Code Listings **24**

hbspy - A Python Interface to the Hierarchical B-spline C++ Library

1 Introduction

1.1 Background

A physicist is interested in discovering and explaining why things are the way they are. This is usually done by making observations, isolating important variables or factors, and building models. In order to use and solve these models physicists need a way to represent them visually and/or in terms of mathematical functions. Especially in physics, these mathematical functions are differential or difference equations with an associated set of boundary conditions.

Finite element methods (FEM) are numerical techniques for finding solutions to boundary value problems using the calculus of variations. From a high level FEM can be thought up of dividing a system in to small components (finite elements), defining the relevant equations for each component, then gathering the pieces together again for computation. Perhaps the best known application of FEM is an engineering tool known as finite element analysis (FEA). FEA works by dividing a surface into a mesh, which is often defined internally by a spline of some sort. A spline is piecewise defined polynomial function that is also smooth where the polynomials pieces come together [1]. Among the most common class of splines are B-splines.

Often the mathematics underlying the visual design (in a CAD program, for instance) is fundamentally different than the mathematics used in the analysis of the design (called finite element analysis). This disparity creates extra work translating the design representation into a format that would be suitable for rigorous analysis. Recent work has been done at BYU to construct a set of tools that can be used both by CAD programs to represent designs and in the analysis of those designs. These tools are known as hierarchical B-splines (HBS) and they have been implemented in C++.

In addition to the CAD integration possibilities for the HBS library, hierarchical B-splines are appealing to physicists due to various inherent mathematical properties:

- HBS basis functions are a partition of unity and have a compact support.
- HBS curves can be made C^∞ between knots and C^{p-k} at knots (p is the degree of spline, k is multiplicity of knot). In this way the user can control the degree of continuity at knot locations.
- Local refinement of basis functions is possible (not generally true of splines).
- Solutions obtained using HBS curves are both accurate and smooth.
- Geometric structure of governing PDEs can be incorporated directly into the basis (for example $\nabla \cdot \mathbf{B} = 0$ in EM, or $\nabla \cdot \mathbf{v} = 0$ in incompressible flow).

The vision for the HBS library is that is will become the most powerful and flexible discretization package for engineering and physics. Because the library is currently written in C++, it will be available only to those who know that language; greatly limiting the potential user base. This proposal will outline a plan to lower the barrier to entry for using the library by creating a python interface to the existing C++ library.

1.2 Motivation

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

1.3 Context

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

2 Methods

In this section I describe the different approaches that were employed during the creation of `hbspy`. I will give an overview of the tools that were used or considered for this project as well as a short usage example for each tool. To maintain consistency and make differences across the methods more apparent, I will use a selection of the code from the HBS C++ library. The main components of this example code are a C++ class `HKnotVector`, a function `numClamp`, and a few typedefs, `DoubleVec`, `IntVec`, and `IntVecVec`. This actual source code can be found in Appendix A.

2.1 SWIG

SWIG[1] is an acronym meaning simplified wrapper and interface generator. The following excerpt from the SWIG homepage provides a good explanation of what SWIG is commonly used for:

> SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages. SWIG is used with different types of target languages including common scripting languages such as Perl, PHP, Python, Tcl and Ruby. . . SWIG is most commonly used to create high-level interpreted or compiled programming environments, user interfaces, and as a tool for testing and prototyping C/C++ software. SWIG is typically used to parse C/C++ interfaces and generate the 'glue code' required for the above target languages to call into the C/C++ code.

SWIG is a very well-established project; the first version appeared in July 1995 and the most recent version was released in May 2013. Over the years, SWIG has developed in to a very powerful and flexible tool. The best expression of this flexibility is that SWIG officially has at least partial support for nineteen different target languages, whereas other tools that will be discussed in this section are Python specific. A great aspect of this flexibility is that users can run SWIG on the exact same set of files and generate wrappers for different target languages by simply changing a single command line argument.

[1] SWIG is free and open source. The source code is hosted at `https://github.com/swig/swig` and the homepage for the project is `http://www.swig.org/`.

However, SWIG is not a perfect tool. Due in part to the freedom it gives users to choose amongst multiple output languages, SWIG generates wrapper code that is relatively difficult to customize for a specific target language. Furthermore, in order to use SWIG, a user must supply an additional interface file (commonly with a `.i` suffix) in which the user uses a C-like syntax to describe the desired interface. Finally, the last main drawback I noticed when testing SWIG for `hbspy` is that the building/compiling phase for SWIG is non-trivial.

2.1.1 SWIG Usage Example

To give an idea of how to use SWIG, I outline how to construct a Python interface to the code contained in Appendix A. The first step is to create a SWIG interface file where the desired wrapper is designed. I will present the wrapper used to expose the class `HKnotVector`, and then explain the key components.

Listing 1: `HKnotVector.i`: SWIG interface for HKnotVector

```
1  %module hbspy
2
3  %{
4  #include "../HKnotVector.h"
5  %}
6
7  %include "std_vector.i"
8  namespace std {
9      %template(IntVec)  vector<int>;
10     %template(DoubleVec) vector<double>;
11     %template(IntVecVec) vector<vector<int> >;
12 }
13
14 %import "../common.h"
15 %include "../HKnotVector.h"
```

- **Line 1** Declare the name of the module. In large projects the module name allows SWIG to create wrappers that don't have issues with namespace resolution.
- **Lines 3-5** This is a special block that is copied and pasted, with out SWIG parsing, directly into the generated C/C++ portion of the wrapper. If there are things that need to happen

for the underlying source to function, but SWIG doesn't need to know about, they go here.

- **Lines 7-12** Notice the use of the `%include` where C++ programmers are used to seeing `#include`. This is a special SWIG statement that instructs SWIG to access the file `"std_vector.i"` (included as part of SWIG) and give the interface access to the `vector` class from within the namespace `std`. I then then expose the typdefs found in `"common.h"` as swig `templates`.

- **Line 14** The SWIG `%import` directive is used to tell the wrapper that important items live in `common.h`, but that no wrapper code needs to be generated for that file.

- **Line 15** Finally the SWIG `%include` directive is used to include the main file `HKnotVector.h` in the generated wrapper.

Although the interface file is only 15 lines, there are a lot of things going on. One thing to note about this interface is that when it is run, the entire `HKnotVector` class (really everything defined in `HKnotVector.h`) is wrapped and exposed to the target language. This could pose problems if various types, functions, or class attributes shouldn't be accessed outside of C or C++.

Using this file is a two-step process: 1) Run SWIG on the `"HKnotVector.i"` and generate the interface, 2) incorporate the generated files into a build system so that they can be `imported` into Python. This first step is very straightforward and can be accomplished by running the following from the command line:

```
swig -c++ -python HKnotVector.i
```

This command runs SWIG, tells it that the source language is C++, the target language is Python and that the interface file is `HKnotVector.i`. After running the command two files will be generated `HKnotVector_wrap.cxx` and `hbspy.py`. Together these files files make up the wrapper of HKnotVector.

The next step is to incorporate these files into a build system so that they can be compiled in a way that the system Python can interact with them. The SWIG documentation gives a few possible methods for doing this, but the recommended solution is to let Python handle the compiling. This will ensure that the correct libraries are linked at compile time and that the version of Python directing the compilation will be able to use the objects. To do this, a `setup.py` file must be created. The interface file for `HKnotVector` appears below (Note that an explanation of key parts of the file are explained after the code is displayed).

Listing 2: `setup.py` file for SWIG

```
1  #!/usr/bin/env python
2  """
3  setup.py file for building SWIG hbs extensions
4  """
5
6  from distutils.core import setup, Extension
7
8  h_knot_vector = Extension('_hbspy',
9                             sources=['./HKnotVector_wrap.cxx']
10                            )
11
12 setup(name='hbspy',
13       version='0.1',
14       author="Spencer Lyon",
15       description="""Wrapping HBS for python using SWIG""",
16       ext_modules=[h_knot_vector],
17       py_modules=["hbspy"],
18       )
```

- **Line 6** From the Python `distutils` package, import the `setup` function and the `Extension` class. The `setup` function is the main driving point in this file and will direct the compilation. The `Extension` class holds all the information the `setup` function needs to compile the objects.

- **Lines 8-10** Describe the HKnotVector extension. Notice the first argument given to the `Extension` constructor is `"_hbspy"`. This argument tells the `setup` function what to name the shared object (or dynamic linking library on Windows) where the compiled wrapper will be placed. Without custom configuration, SWIG requires that this name be a leading underscore followed by the `%module` name defined in the interface file.

- **Lines 12-18** Call the `setup` function to build all the `Extensions` in the `ext_modules` list. This is also where other metadata about the project goes.

The final step in building the interface is to have python compile the wrappers. This is done on the command line with a single command:

```
python setup.py build_ext --inplace
```

This command tells the system Python (whatever `python` resolves to on the user's `$PATH`) to build the extensions outlined in `setup.py` inplace, meaning in the current working directory.

In the end, I decided not to use SWIG to create the interface to the entire HBS library. The verbose C-like interface files and the need to create a separate interface file for each source file made SWIG more difficult than necessary. In addition, the fact that all code from an exposed C++ source file is wrapped was overkill for this project. However, as can be seen from this small exercise, it is a fairly straightforward, if tedious, process to use SWIG to create a Python interface to C++ code.

2.2 Boost.Python

Boost.Python[2] (henceforth Boost for short) is an alternative to SWIG and is a highly specialized tool for wrapping C++ for Python use. This apparent lack of flexibility has allowed the Boost developers to provide a very natural and complete coverage of the C++ language. Some key C++ features that are supported in boost are

- References and Pointers

[2]Boost.Python is part of the free peer-reviewed Boost project. Boost can be downloaded from the main projects webstie at `http://www.boost.org/`. The documentation for Boost.Python can be found at `http://www.boost.org/doc/libs/1_54_0/libs/python/doc/index.html`.

- Efficient function overloading
- C++ to Python exception translation (cuts down on `SEGFAULT`s)
- Functions or methods with default and keyword arguments
- Exporting C++ iterators as Python iterators
- Control over Python documentation strings

On the other hand, Boost has some limitations. First, Boost has a difficult `Bjam` utility for compiling the wrappers. `Bjam` is similar to `make`, but has a difficult and strange syntax. Second, the generated wrapper code is generally very verbose. While is is probably do to supporting some C++ features that other wrapping tools do not, it has at least two major drawbacks: 1) It takes a long time to compile the wrappers and 2) the python-side execution is typically noticeably slower than the code generated with other tools. Finally, the major drawback and ultimate reason why I did not use Boost for `hbspy` is that it is very difficult to install. After reading the (sparse) documentation and searching the internet, I still could not get Boost.Python correctly installed and configured on my system. This would be a major roadblock to future users of the Python bindings and would actually detract from the main justification for creating the bindings: lowering the bar to entry for using HBS in research. For these reasons, I will not include a usage example for Boost.Python, but because I spent quite a bit of time on it and many people seem to like it, I felt it needed to be addressed in this report.

2.3 Cython

I now shift focus a bit and talk about a different type of tool, `Cython` [3] Instead of purely being a tool to wrap compiled languages for use in Python, Cython is actually a super-set of the Python

[3] `Cython` is free and open source. The source code is hosted at https://github.com/cython/cython and the homepage for the project is http://cython.org/.

language; anything that is valid Python code is also valid Cython code. However, Cython adds a few major improvements:

- Variables, functions, and classes can be given static types. This avoids much of the overhead inherent in a "duck-typed" interpreted language.
- Cython programs can make direct calls to C, C++, and Fortran code. This allows the user to directly mix python low-level, high performance compiled code.

Cython accomplishes this translating by the Cython code directly to C or C++, which can then be compiled and loaded into any Python script or session. This means that blocks of code where all objects have been given static types can be written directly in C and therefore achieve almost[4] C-like performance. In addition, the ability to directly call C, C++, or Fortran makes Cython a viable option for wrapping low-level code for use in Python. I will now show examples these two features.

2.3.1 Cython Type Example

REVIEW: Should this section on Cython typing be in an appendix? It is useful for the wrapping example, but makes this section long.

The main point of entry for adding static types in Cython is the `cdef` keyword. This can be used before any object to assign a type to it. All C types can be used as valid `cdef` declarations: numeric types, structs, unions, pointers, ect. When using `cdef`, Cython will generate C code that does automatic type conversion between related Python and C types. The end result of code that has been properly typed using `cdef` is much faster code - sometimes faster by orders of magnitude.

[4] The almost is necessary because there is small overhead in calling the compiled routines from Python and getting the results back.

To demonstrate the use of the cdef keyword I will show Python and Cython versions of a pairwise-distance function. This function takes in an nxm matrix that represents n points in m dimensions and it will return an nxn matrix containing the Euclidean distance between each point in the input array and every other point in that array. I show Python and Cython versions below and then explain the differences:

Listing 3: pairs.py: Pure Python pairwise distance function

```python
from math import sqrt
import numpy as np


def dist(x):
    n = x.shape[0]
    m = x.shape[1]
    ret = np.empty((n, n))
    for i in range(n):
        for j in range(n):
            d = 0.0
            for k in range(m):
                tmp = x[i, k] - x[j, k]
                d += tmp * tmp
            ret[i, j] = sqrt(d)
    return ret
```

Listing 4: cy_pairs.pyx: Cython pairwise distance function

```cython
1  from libc.math cimport sqrt
2  import numpy as np
3
4
5  cpdef dist(double[:, ::1] x):
6      cdef int n = x.shape[0]
7      cdef int m = x.shape[1]
8      cdef double[:, ::1] ret = np.empty((n, n))
9      cdef double d, tmp
10     cdef int i, j, k
11     for i in range(n):
12         for j in range(n):
13             d = 0.0
14             for k in range(m):
15                 tmp = x[i, k] - x[j, k]
16                 d += tmp * tmp
17             ret[i, j] = sqrt(d)
18     return ret
```

- **Line 1** Cython exposes the C standard library via libc.<headerName>. The sqrt function from the standard library is a bit faster than the one from Python's math package. Note that I must use the additional Cython keyword `cimport` to access this function.

- **Line 5** Notice the use of the keyword `cpdef`. This keyword is used to define functions or classes that need to be callable from both Python and C. Were I to have used `cdef` here, the function would be translated to a C function and I would not be able to call it from python. Behind the scenes `cpdef` instructs the Cython to C translator to make two versions of the function.

- **Line 5** Also note that on line 5 I declare a Cython typed memoryview using `double[:, ::1]`. This statement tells Cython that `x` will be a two dimensional array of doubles. In addition, the `::1` in the second position tells Cython that `x` will be C-contiguous[5]. This allows the generated C code to use natural C array operations on `x`.

- **Lines 6-10** Here I give static types to all variables local to the function. Note the use of the typed memory view again on line 8. Also note that Cython requires types to be declared at the top level of a function. For that reason, I declared `d` and `tmp` as double and `i, j, k` as int before entering first `for` loop.

- The rest of the function is identical to the pure Python version.

In order to use the Cython version of the function, we must instruct Cython to translate it to C and then compile it for Python use. There are many ways to do this, but as with SWIG it is easiest to let Python handle it for us using a `setup.py` file. A setup.py file for this function appears below:

Listing 5: `setup.py` file for Cython pairwise distance

```python
import os
from distutils.core import setup
from distutils.extension import Extension
import numpy as np

inc_dirs = [np.get_include(),
            '..']

hkv = Extension("hbspy.HKnotVector", ['HKnotVector.pyx'],
                include_dirs=['..', '.', np.get_include()], language="c++")

setup(name="Pairwise distance", ext_modules=[hkv])
```

[5]Note that by default all numpy arrays are C-contiguous.

This file is very simple: lines 1 and 2 import the `setup` and `cythonize` functions and line 4 calls the `setup` function where the extension modules are given using the `cythonize` function. The only remaining step is to build the extension using the command used to build the SWIG extension above. I repeat the command here:

```
python setup.py build_ext --inplace
```

I timed both of these functions using `x = np.random.randn(1500, 5)` as the input array. Both functions returned the exact same answer, but the execution time was very different. The Python function took 21.2 seconds to execute, whereas the Cython version only took 75.6 milliseconds: a speedup of over 280x [6]!

2.3.2 Cython Wrapping Example

Building on the static typing example, I will now show how to use Cython to wrap the HKnotVector example. In order to use external libraries, you need to tell Cython two things: 1) what file the external components are defined in (usually a header `.h` file) and 2) which parts of that file you would like to access from Cython. For example, instead of calling `from libc.math cimport sqrt`, I could have done the following:

```
cdef extern from "math.h":
    double sqrt(double x)
```

In the first line I started a `cdef extern` block. The syntax is simply cdef extern from <headerName>:, where `headerName` is the name of the external file where the desired objects are defined (`"math.h"` for this example). Everything in the indented block following the `:` is part of this `cdef`

[6] I also have a more optimized version of the Cython code that only takes 14.9 milliseconds to run. While that shows a speed improvement of over 1400x, it makes use of some advanced Cython features that are beyond the scope of this report.

`extern` block and contains the external declarations that need to be exposed to Cython.

In a larger project, it is often necessary to create a Cython interface file (with a `.pxd` extension), which does for Cython what a `.h` interface file does for C/C++ . This is necessary when you have multiple Cython `.pyx` files that need to access the same external source. The declarations go into a Cython `cdef extern` block in a `.pxd` file. This interface is very similar to the C/C++ interface; often users can copy and paste directly from C to Cython. The actual implementation will go into a file with the same name, but with a `.pyx` extension. This is very similar to `.h` and `.c` files for C. Furthermore, when wrapping a set of C++ classes, people often put the extern definitions in a file named something like `cpp_<headerName>.pxd` and Cython declarations in a file named `<headerName>.pxd`. This is important because it is generally necessary to have one interface file for external declarations (the file named `cpp_`), and another interface file exposing the Cython implementation.

The structure of a Cython wrapper is best understood by example, which I now show as I wrap HKnotVecter. I begin with Listing 6, which is the file `cpp_HKnotVector.cpp`. In this file I use the `vector` class defined in `libcpp.vector` and include the all the declarations that appear in `HKnotVector.h` (Listing 11 in Appendix A).

REVIEW: Should these code listings be put in an appendix? Part of me thinks yes because they are long, but the other part says no because they are central to what I have done with hbspy

Listing 6: `cpp_HKnotVector.pxd`

```
from libcpp.vector cimport vector as cpp_vector

cdef extern from "HKnotVector.h" namespace "hbs":

    cdef cppclass HKnotVector:
        # constructors
        HKnotVector()
        HKnotVector(unsigned int, const cpp_vector[double] &)

        # methods
```

```
        unsigned int degree()
        bint isEven()
        bint isOdd()
```

The next part of the wrapper is the Cython interface `HKnotVector.pxd` in Listing 7. This is a very minimal file that declares the `HKnotVector` class and sets up some initial attributes of the class.

Listing 7: `HKnotVector.pxd`

```
from hbspy cimport cpp_HKnotVector

cdef class HKnotVector:
    cdef void * _inst
    cdef public bint _free_inst
```

The final part and main of the wrapper is `HKnotVector.pyx`, shown here in Listing 8. This is where all attributes and methods declared in either of the interface file are implemented.

Listing 8: `HKnotVector.pyx`

```
cimport numpy as np
from libc.stdlib cimport free
from libcpp.vector cimport vector as cpp_vector
import numpy as np

np.import_array()


cdef class HKnotVector:
    def __cinit__(self, *args, **kwargs):
        self._inst = NULL
        self._free_inst = True

    def __dealloc__(self):
        if self._free_inst:
            free(self._inst)

    # constuctors
    def _constructor1(self):
        self._inst = new cpp_HKnotVector.HKnotVector()

    def _constructor2(self, degree, knots):
        cdef cpp_vector[double] cpp_knots
        cdef int i
        cdef int knots_size = len(knots)
        cpp_knots = cpp_vector[double](<size_t> knots_size)
        for i in range(knots_size):
            cpp_knots[i] = <double> knots[i]
        self._inst = new cpp_HKnotVector.HKnotVector(<unsigned int> long(degree),
            cpp_knots)

    def __init__(self, *args, **kwargs):
```

```
32          if len(args) == 2:
33              self._constructor2(*args, **kwargs)
34          else:
35              self._constructor1(*args, **kwargs)
36
37      # methods
38      def degree(self):
39          cdef unsigned int rtnval
40          rtnval = (<cpp_HKnotVector.HKnotVector *> self._inst).degree()
41          return int(rtnval)
42
43
44      def isEven(self):
45          cdef bint rtnval
46          rtnval = (<cpp_HKnotVector.HKnotVector *> self._inst).isEven()
47          return bool(rtnval)
48
49
50      def isOdd(self):
51          cdef bint rtnval
52          rtnval = (<cpp_HKnotVector.HKnotVector *> self._inst).isOdd()
53          return bool(rtnval)
```

- **Lines 1-6** All necessary items are imported and set up. Notice that neither of the interface files are not actually imported here. If a `.pxd` and a `.pyx` file are in the same directory and have the same name, then all things imported or defined in the `.pxd` are automatically available in the `.pyx` file.

- **Lines 9-16** Use `cdef` to declare the class and set up a few special Cython methods. `__cinit__` is called immediately after the user tries to create an instance of the class and usually holds the minimial setup required to avoid a `SEGFAULT` from null pointers. The `__dealloc__` method is called when the object is passed through the Python garbage collector and is implemented here to avoid memory leaks.

- **Lines 18-35** Here the overloaded constructor for HKnotVector is set up. Two private methods (private by convention of starting with a single underscore) are implemented to handle each the overloads. The `__init__` method is called after `__cinit__` when an `HKnotVector` instance is created and is implemented to dispatch object creation to one of the overloaded constructors.

- **Lines 37-53** The methods declared in `cpp_HKnotVector.pxd` are implemented. Pretty much the only thing that needs to happen here is type checking. To do this I use `cdef` to statically declare variable types and cast objects using $< \cdot >$.

Now that the wrapper is completed, it needs to be incorporated into a build system and complied into a shared object so that Python can access it. As before, we let python handle this step using a `setup.py` file, which I have included in Listing 9. There are only a few differences between this file and the other `setup.py` files presented earlier. First, in lines 6 and 10 I explicitly specify the include directories for the `HKnotVector` extension. Also, I setup the `hbspy` package with a module named `HKnotVector`. This happens on lines 8 and 17.

Listing 9: `setup.py` for Cython wrapper of HKnotVector

```
1  from distutils.core import setup
2  from distutils.extension import Extension
3  from Cython.Distutils import build_ext
4  import numpy as np
5
6  incdirs = ['..', '.', np.get_include()]
7
8  HKnotVector = Extension("hbspy.HKnotVector",
9                          ["hbspy/HKnotVector.pyx"],
10                         include_dirs=incdirs, language="c++")
11
12 ext_modules = [HKnotVector]
13
14 setup(name='hbspy',
15       cmdclass={'build_ext': build_ext},
16       ext_modules=ext_modules,
17       packages=['hbspy']
18       )
```

As can be see from this example, wrapping code using Cython provides absolute control over the structure and fell of the wrapper, but it takes a lot more work than, for example, SWIG. I have only wrapped a very small portion of the HBS library in this example, but it illustrates the point. The sheer size of the HBS library makes it unreasonable to construct a wrapper by hand using Cython. Additionally, the core HBS C++ library is still being developed and is therefore liable to change at any time. Trying to keep the Cython wrapper up to date would be a difficult and error-prone task. For these reasons, I decided not to use a by-hand Cython approach in creating `hbspy`.

2.4 XDress

The final tool I evaluated when creating the Python wrapper for HBS is XDress [7]. XDress is a very young project that first appeared on github in April 2013. XDress is written in pure python and is an automatic Python wrapper generator for C and C++ source. It constructs the wrapper in a three stage process.

1. External (to XDress) parsing tools are run on the source and a static xml representation of the data structures is generated. Currently, XDress uses GCC-XML[8] for C++ parsing and pycparser[9] for C.
2. The generated xml files are parsed and the C-based API is described in terms of an internal XDress typesytem. This typesystem is very dynamic and was designed from the ground up with API generation in mind. It is the main enabling feature of XDress.
3. XDress uses various built-in and/or user-supplied plugins to take the API stored in the typesystem and form Cython bindings.

2.4.1 XDRESSRC.PY

Compared to the other methods discussed here, XDress is very easy to use . The main point of entry for using xdress is to call `xdress` from the command line. When this command is executed (with no extra arguments options) it will scan the current directory for a file named `xdressrc.py`. All the instructions for XDress are put into this single python file. It is easiest to understand the types of instructions that need to be in this file by example, so I present one here.

[7] XDress is free and open source. The source code is hosted at https://github.com/xdress/xdress and the homepage for the project is http://xdress.org/.

[8] GCC-XML is free and open source. The source code is hosted at https://github.com/gccxml/gccxml and the homepage for the project is http://gccxml.github.io/HTML/Index.html.

[9] pycparser is free and open source. The source code is hosed at https://github.com/eliben/pycparser and the (limited) documentation is found in README.rst in the source.

Listing 10: Sample `xdresrc.py`: SWIG interface for HKnotVector

```
1  package = 'package'
2  packagedir = 'output'
3  sourcedir = 'src'
4
5  plugins = ('xdress.stlwrap', 'xdress.autoall', 'xdress.autodescribe',
6             'xdress.cythongen', 'foopack.barplug')
7
8  ## Which stl containers we need for this code
9  stlcontainers = [('vector', 'float64'),
10                  ('set', 'int'),
11                  ('map', 'int', ('map', ('vector', 'uint'), ('set', 'char'))),
12                  ('vector', ('vector', 'float64')),
13                  ('set', 'FooClassBar')
14                  ]
15
16 ## Which classes to create wrappers for.
17 classes = [('FooClass', 'Foo'),
18            ('FooClass', 'Bar', 'Foo', 'FooClassBar'),
19            ]
20
21 functions = [('FooFunc', 'Foo')]
22
23 Variables = [('barVar', 'Bar')]
```

- **Lines 1-3** Set the name of the Cython package, the output directory where the Cython wrapper will go, and the name of the directory where the C/C++ source lives.

- **Lines 5-9** This is an optional step where the user can specify which plugins should be run when `xdress` is executed. All but the last plugin (`'foopack.barplug'`) are built-in plugins that come with XDress. They perform the following functions:

  – `'xdress.stlwrap'`: Generates wrapper a for C++ STL objects (see next bullet for more information)

  – `'xdress.autoall'` and `'xdress.autodescribe'`: Parse all included files and enter all objects into the typesystem

  – `'xdress.cythongen'`: Use the generated typesystem to actually write out the Cython files that define the wrapper

  – `'foopack.barplug'`: Run the user supplied plugin `barplug` found in the package `foopack`.

  Note that if the `plugins` list is omitted from this file that XDress will automatically populate this list with the necessary plugins to create the Cython interface.

- **Lines 9-14** Specify which STL containers to create Cython wrappers for. These wrappers will be exposed to Python via custom NumPy dtypes that do all data sharing in memory (no copying). Notice that the specifications here can take on a nested form to accomidate arbitrary complexity. Also note that non-native C/C++ types can be specified here, with the restriction that the user-defined types be mentioned in the `classes` list below (see next bullet).

- **Lines 17-19**: An optional list of classes XDress should generate wrappers for. The `classes` object specified here should be a list of tuples. There are various formats for specifying the contents of each tuple, but the format used here is `('source_name', 'source_file', 'target_name', 'target_file')`. Where `source_name` is the name of the class in C++, `source_file` is the file where the class is defined in C++ and the `target` variants are the name and file for how and where the class should be defined in C++.

- **Lines 21-23**: Optional lists of functions and variables that should be wrapped. The syntax is similar to the syntax for classes.

2.4.2 XDress Plugins

As can be seen, specifying the API elements that need to be wrapped is straightforward and simple. To complement this simplicity XDress has a very easy to use plugin architecture that gives users absolute control over how the wrapping is handled. The plugin system is not a mere afterthought, but build into the core of how XDress operates. All the major functionality of XDress is modularized into distinct plugins that are executed using this archtecture. This means that user-supplied plugins will be given the same precedence as built-in plugins. To demonstrate some of the possibilities for XDress plugins, I will explain two of the plugins I have written to handle issues encountered with wrapping HBS.

The first of these plugins is now a part of XDress and lives in `xdress.descfilter`. This plugin allows the user to instruct XDress to "filter out" certain API components from the generated wrapper. This can be done in one of two ways: 1) specify that functions or methods with certain types in the function signature be excluded or 2) specify that certain methods of a class should be excluded. This flexibility can be useful when there are certain functions that shouldn't be exposed to Python. This is also a useful stop-gap for data types that have not yet been implemented into the XDress type system. In Section 3.1 I will present the actual `xdressrc.py` file currently being developed for HBS, which will provide a usage example for `xdress.descfilter`.

The second plugin is also part of XDress and lives in `xdress.doxygen`. This plugin uses dOxygen[10] to output an xml version of in-line documentation contained in the C/C++ source. The plugin then parses this xml, automatically generates Python docstrings, and inserts them into the Cython wrappers. These docstrings have many uses such as to provide information on methods, classes, or functions when these objects are inspected at the Python interpreter, or to be used by a tool like Sphinx[11] in conjunction with other content to produce stylized documentation.

Another important item to note is that because I got involved with XDress development at a very early stage and the HBS library utilizes a lot of advanced C++ language features, much of the recent and current development of XDress is being driven by the needs that arise in wrapping HBS. This, together with the ease and freedom XDress provides, caused me to choose XDress as the tool to use in constructing `hbspy`. As the HBS project moves forward, the relationship with the lead XDress developer, Anthony Scopatz, and the close integration between HBS and the

[10] dOxygen is a common documentation utility for C/C++ projects that gives the user the ability to have specially formatted comments in the source code become stylized documentation elements. dOxygen is free and open source. The code is hosted at `https://github.com/doxygen/doxygen` and the homepage for the project is `http://www.stack.nl/~dimitri/doxygen/`.

[11] Sphinx is a python package that can automatically create html or pdf (via latex) documentation using the reStructuredText markup language. Additionally, Sphinx can inspect docstrings and turn them into stylized documentation elements, much like dOxygen.

XDress development cycle will very helpful and should ensure long-term functionality.

3 Results and Discussion

3.1 XDress and HBS

TODO: This is a very fluid concept and I think I am going to hold off on writing it for a little

4 Conclusion

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

REFERENCES

[1] K. Judd, *Numerical Methods in Economics*. MIT Press, 1998, ISBN: 9780262100717. [Online]. Available: http://books.google.com/books/about/Numerical_Methods_in_Economics.html?id=9Wxk_z9HskAC.

A HBS (C++) Code Listings

Below are the code listings that are used as examples throughout section 2.

Listing 11: Portions of HKnotVector.h

```
#ifndef _H_KNOT_VECTOR_H_
#define _H_KNOT_VECTOR_H_

#include "common.h"
#include <vector>
#include <iostream>

using namespace std;
using namespace util;

namespace hbs
{
  class HKnotVector
  {
    /// A one-dimensional object which stores a knot vector of any degree.
    /// No geometric operations are performed using a knot vector, only basis
    /// function queries. This class is best used in connection with a HNURBS
        object which
    /// stores the geometric information. We do store the extra knot for open
    /// knot vectors. So a degree p knot vector will have p + 1 knots at the
        beginning
    /// and end of the knot vector. We currently don't support periodic knot
    /// vectors although this could be added pretty easily.
    public:

    /// Default constructor
    HKnotVector() : mDeg( 0 ) {}

    /// construct a knot vector from a vector of knots. We assume that p + 1
        repeated
    /// knots exists at the beginning and end of the knot vector.
    HKnotVector( uint degree, const DoubleVec &knots )
      : mDeg( degree ), mKnots( knots )
    {
      getKVecData( mKnots, mGroups, mReverseGroups, mMultipleCount );
    }

    /// A destructor
    ~HKnotVector() {}

    /// Returns the degree of this knot vector.
    uint degree() const { return mDeg; }

    /// Returns true if the knot vector is even.
    bool isEven() const { return degree() % 2 == 0; }

    /// Returns true if the knot vector is odd.
    bool isOdd() const { return !isEven(); }

    protected:

    uint mDeg;
```

```
    DoubleVec mKnots;
    IntVec mGroups;
    IntVecVec mReverseGroups;
    IntVec mMultipleCount;

    /// Returns group, multiplicity, zcount data for a vector of knots.
    void getKVecData( const DoubleVec &knots, IntVec &knot_groups,
              IntVecVec &reverse_knot_groups, IntVec &multiple_counts ) const
    {
      knot_groups.clear();
      reverse_knot_groups.clear();
      multiple_counts.clear();
      knot_groups.push_back( 0 );
      multiple_counts.push_back( 0 );
      uint group_index = 0;
      uint multiple_count = 0;
      IntVec group;
      group.push_back( 0 );
      for( uint iknot = 1; iknot < knots.size(); ++iknot )
      {
        if( equals( knots[ iknot - 1 ], knots[ iknot ], 1e-8 ) )
        {
          group.push_back( iknot );
          ++multiple_count;
        }
        else
        {
          ++group_index;
          multiple_count = 0;
          reverse_knot_groups.push_back( group );
          group.clear();
          group.push_back( iknot );
        }
        knot_groups.push_back( group_index );
        multiple_counts.push_back( multiple_count );
      }
      reverse_knot_groups.push_back( group );
    }
  };
}
#endif
```

Listing 12: Portions of common.h

```
#ifndef _UTIL_COMMON_H_
#define _UTIL_COMMON_H_

#include <climits>
#include <iostream>
#include <vector>
#include <set>
#include <map>
#include <cmath>
#include <string>
#include <assert.h>

/// common definitions needed throughout the hbs library

```

```
typedef unsigned int uint;
typedef unsigned long ulong;
typedef unsigned short ushort;
typedef unsigned char uchar;

using namespace std;

namespace util
{
  /// Clamps the values to a determined range. The values
  /// 'minimum' and 'maxmimum' must be of a type that can be
  /// cast to the same type as 'value', and must be less-than
  /// comparable with value's type as well.
  template< typename T, typename T2, typename T3 >
  inline T numClamp( T value, T2 minimum, T3 maximum )
  {
    if( value < minimum )
      return minimum;
    if( maximum < value )
      return maximum;
    return value;
  }

  /// This form is a little inconvenient, but is the basis of most other
  /// ways of measuring equality.
  inline bool equals( double a, double b, double tolerance )
  {
    // This method has been benchmarked, and it's pretty fast.
    return ( a == b ) ||
      ( ( a <= ( b + tolerance ) ) &&
      ( a >= ( b - tolerance ) ) );
  }

  typedef std::vector< double > DoubleVec;
  typedef std::vector< int > IntVec;
  typedef std::vector< IntVec > IntVecVec;
}
#endif
```