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

Spencer Lyon

July 30, 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

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

1 Introduction

1.1 Background

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.

1.2 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.

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.

[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/.

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: SWIG interface HKnotVector.i

```
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
- 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

[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.

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 spend 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

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.

2.4 XDRESS

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.

3 Results and Discussion

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.

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.

A HBS (C++) CODE LISTINGS

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

Listing 3: 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 4: 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
```