Extending PCL for use with Python: Bindings generation using Pybind11

(Proposal for Google Summer of Code, 2020)

Name and Contact Information

Name: Divyanshu Madan
Github: divmadan
Gitter: divmadan
Email: divyanshumadan99@gmail.com
Postal Address: 4-Cha-18, Jawahar Nagar, Jaipur, Rajasthan
Country: India
Phone: (+91) 9261700477
Primary Language: English

About the document

This document is very extensive and requires a section about itself, to stay sane. A few points:

- It documents various tools, existing solutions, other project's solutions etc. along with my proposal, how to tackle insufficiencies, timeline, etc.
- The "research" part (existing solutions, analysis, etc) is very extensive, difficult to structure in tabular form, and hard to summarize. Thus, it takes up a lot of space in the document.
- I have tried to make the sections independent of each other, so you can safely skip parts of the document, in case you want to.
- I have limited the headings up to 3 levels (H2, H3, H4).

A simplified skeleton or structure of the document:

Table of Contents

C, C++ and Python

As described by Boost.Python:

Python and C++ are in many ways as different as two languages could be: while C++ is usually compiled to machine-code, Python is interpreted. Python's dynamic type system is often cited as the foundation of its flexibility, while in C++ static typing is the cornerstone of its efficiency. C++ has an intricate and difficult compile-time meta-language, while in Python, practically everything happens at runtime.

These very differences mean that Python and C++ complement one another perfectly. Performance bottlenecks in Python programs can be rewritten in C++ for maximal speed, and authors of powerful C++ libraries choose Python as a middleware language for its flexible system integration capabilities.

Furthermore, the surface differences mask some strong similarities:

- 'C'-family control structures (if, while, for...)
- Support for object-orientation, functional programming, and generic programming (these are both multi-paradigm programming languages.)
- Comprehensive operator overloading facilities, recognizing the importance of syntactic variability for readability and expressivity.
- High-level concepts such as collections and iterators.
- High-level encapsulation facilities (C++: namespaces, Python: modules) to support the design of reusable libraries.
- Exception-handling for effective management of error conditions.
- C++ idioms in common use, such as handle/body classes and reference-counted smart pointers mirror Python reference semantics.

Given Python's rich 'C' interoperability API, it should in principle be possible to expose C++ type and function interfaces to Python with an analogous interface to their C++ counterparts.

However, the facilities provided by Python alone for integration with C++ are relatively meagre:

- Compared to C++ and Python, 'C' has only very rudimentary abstraction facilities, and support for exception-handling is completely missing.
- 'C' extension module writers are required to manually manage Python reference counts, which is both annoyingly tedious and extremely error-prone.
- Traditional extension modules also tend to contain a great deal of boilerplate code repetition which makes them difficult to maintain, especially when wrapping an evolving API.

These limitations have led to the development of a variety of wrapping systems to overcome these issues and utilise the complement ability of C++ and Python, for efficient and rapid development in the community.

Motivation: why create Python bindings for PCL?

- Bindings can be used to take advantage of a large, tested, stable library written in C++ in Python. A perfect reason to develop Python bindings for PCL.
- It encourages software reuse and reduces the requirement of reimplementing a library in several languages.
- Generating bindings for PCL will help rapid development in Python while utilising the speed and efficiency of the underlying C++ API.
- It will also enable python users to take advantage of the C++ based PCL, thus expanding the user base of the library.

Existing binding solutions (for PCL)

Pcl.py

Implementation

- Pcl.py implements a Cython library to process point clouds, combined with Scipy and Numpy.
- It uses CMake, has ROS support and uses PClPointCloud2 instead of PointCloud.

Issues

- The heavily templated part of PCL is not implemented in this library due to the limitation of Cython (Cython doesn’t support a template technique similar to “covariant”).
- It adds templated Cython headers to overcome this limitation and the user has to write that part in C++ and then wrap the I/O in Python which is a lot of work.

Lacking as PCL bindings

- It is an implementation of PCL in Cython, it doesn’t use any support tools for the binding, thus everything needs to be handled manually, almost impossible due to the active development of PCL itself.
- The library is under heavy construction, it is advised not to use it in production codes.

Pclpy

Implementation

- Pclpy uses CppHeaderParser to parse the header files and pybind11 to generate binding code for them.

Issues

- This library is in active development, the API is likely to change.
- The included modules do work, but tests are incomplete, and corner cases are still common.

Lacking as PCL bindings

- The repository is inactive (last commit 2 years ago, master branch).
- It only supports Windows officially.
- Incomplete: although it covers most of the modules, some modules are skipped.

Pypcd

Implementation

- Pypcd is a pure python, module implementation to read and write point clouds stored in the PCD file format.
- It parses the PCD header and loads the data as a Numpy array. That can then be used to perform some functions in the python environment.

Issues

- It is not production-ready code.
- Extra functionality like ASCII support is hackish and untested.
- There is no synchronization between the metadata fields and data in the `pc_data` array (synchronization issues).
- It doesn’t work on Python3.
- Slow while using ASCII.

Lacking as PCL bindings

- It is not a binding to the PCL I/O file format, rather a pure implementation of it.
- It only supports PCD file format and it’s functionalities.

Python-PCL

Implementation

- Python-pcl is the most popular 3rd party binding, supporting multiple versions and multiple OS, utilising the original PCL library itself.
- It uses Cython to generate Python bindings for a portion of the API.

Issues

- The heavily templated part of PCL is difficult to implement in this library due to the limitation of Cython.

- The result for python-PCL is a lot of code repetition, which is hard to maintain and to add features to, and incomplete bindings of PCL's classes and point types.

Lacking as PCL bindings

- It doesn't use any support tools for the binding, thus everything needs to be handled manually.
- The repository is inactive (last commit about 1 year ago).
- Incomplete: only supports some modules.

Analysis of other open-source projects' bindings

OpenCV-Python

(See documentation, source code)

- In OpenCV, all algorithms are implemented in C++ (like PCL).
- In a nutshell, OpenCV generates wrapper functions automatically from the C++ headers using some Python scripts.

Implementation

1. There is a CMake script which checks the modules to be extended to Python and grab their header files. These header files contain a list of all classes, functions, constants etc. for those particular modules.
2. These header files are passed to a Python bindings generator script `gen2.py`. It calls another Python header parser script `hdr_parser.py`. It splits the complete header file into small Python lists such that these lists contain all details about a particular function, class etc. The final list contains details of all the functions, enums, structs, classes etc. in that header file.
3. The header parser doesn't parse all the functions/classes in the header file. The developer has to specify which functions should be exported to Python. For that, there are certain macros added to the beginning of these declarations which enables the header parser to identify functions to be parsed.
4. The generator script `gen2.py` will create wrapper functions for all the functions/classes/enums/structs parsed by header parser (We can find these header files during compilation as `pyopencv_generated_*.h` files).
5. Some basic OpenCV data types like Mat, Vec4i, Size and some complex structs/classes/functions etc. need to be extended manually. For example, a Mat type should be extended to a Numpy array. All such manual wrapper functions are handled by `cv2.cpp`.
6. An example:
   `res = equalizeHist(img1,img2)` : We pass two numpy arrays which are converted to cv::Mat and then calls the equalizeHist() function in C++. For the result, res will be converted back into a Numpy array. Almost all operations are done in C++ which gives them almost the same speed as that of C++.

Issues

- There may be exceptional cases where generator scripts cannot create the wrappers. Such functions need to be handled manually by writing `pyopencv_*.hpp` which extends headers and then puts them into the misc/python subdirectory of the module.
- There are some methods that do not exist in C++ interface but are needed to handle some functionalities at the Python side, like `UMat::queue()` and `UMat::context()`for OpenCL. To handle this:
  - Macro `CV_WRAP_PHANTOM` is used which takes the method header as its argument.
  - We have to provide the method body in the `pyopencv_*.hpp` extension.

- If a default argument is needed, but it is not provided in the native C++ interface, we have to provide it for the Python side as the argument of macro `CV_WRAP_DEFAULT`.

Final notes

- An important point: this implementation was proposed about 10 years ago.
- The approach uses custom header parser and binding generator scripts for generating bindings and doesn't use any support tooling.
- As discussed on the PCL Gitter chat, it is not the preferred approach, at least for python bindings since well-supported tools such as pybind11 are available for use and in popular use. In addition, the custom script approach is not suitable for active development cases.

VTK-Python

(See FAQ, source code, documentation)

- **Wrapping as a multi-stage general process in VTK**:
    1. Identify header files that contain wrappable interfaces.
    2. Generate build rules that will create glue code from header files
    3. Parse the header files to generate a description of the interfaces
    4. Generate glue code from these descriptions
    5. Compile and link the glue code to create wrapper-language modules
- **Flow:**
    - Steps 1 and 2 are done by CMake.
    - Steps 3 and 4 are done by the VTK wrapping tools (often done by a single program).
    - Step 5 is done by a C++ compiler.

Implementation

- The binding generation process relies on the consistent coding style followed in the VTK header files and the use of the convenience macros.
- A target language-specific parser parses the header file, figures out suitable wrapper functions in the desired target language and generates the wrapper code that is used to generate the wrappings.
- **Files organization**:
    - `vtkWrapPython.c` actually does the code generation for Python.
    - When VTK is built, an executable named `vtkWrapPython` is built from this. It generates the Python wrappers for each wrappable VTK header.
    - The wrapper code makes use of `vtkPythonUtil.h` and several other C and C++ files in Wrapping/Python for various things. Also, note that `hints` is a mechanism for helping the wrapper code with hints to help the wrapping process along. Details of individual codes can be seen in`vtkParse.y` and `vtkParse.tab.c`
- **CMake Role**:
    - It is used to build the parser and organize the building of the generated code, link it correctly etc.
    - Macros: `WRAP_EXLUDE_PYTHON`, `WRAP_EXCLUDE`, `WRAP_LIST` are used to simplify the process.

Issues

- According to documentation, a method is not wrapped if:
    1. Its parameter list contains a pointer to anything other than a vtkObjectBase-derived object or a fundamental C++ type.

2. It returns a pointer to anything other than a vtkObjectBase-derived object unless the method returns a pointer to a fundamental C++ type and has an entry in the wrapping hints file, or the method is a vtkDataArray Get() method or the returned type is 'char *' or 'void *'
3. It is an operator method.

Final notes

- An important point: this implementation was proposed about 8 years ago.
- The file structure is very messy and hard to figure out. It has a file to wrap the code, an executable, utils file and several other files for "various things" as mentioned in the documentation. It uses hints to help the wrapping process along.
  In addition, it has `Hierarchy.txt` for modules, multiple CMakes for each language binding, etc.
- The whole process is very tailored and custom to use with no extensibility. A structure like this will be very hard to maintain.
- The source code also needs `yacc` C++ parser files and `lex.yy.c` helper files, along with preprocessor evaluators and lots of other custom files, not necessary for PCL use cases.
- As discussed on the PCL Gitter chat, it is not the preferred approach, at least for python bindings since well-supported tools such as pybind11 are available for use and in popular use. Even if we had to go with the custom approach, I believe the task can be performed in a more simplified way like OpenCV.
- A positive point that can be taken from their approach is the use of docstrings, which will enable easy `help()` function support in Python.

Open3D

(See publication, source code)

Implementation

- Developers use Python as a glue language to assemble components implemented in the backend.
- They use pybind11 as the helper tool for generating bindings.
- Codes are generated on a per-module basis keeping the interface clean. A final C++ interface file generated the code for all modules when compiled.
- CMake is used as the de facto tool for building purposes.

Issues

- The python bindings are not automated, i.e there is no script to parse the header files and generate the pybind11 interface code. Every file is manually coded for now (they are planning to move to code generation later), which is very hard to maintain.

Final notes

- The implementation using the Open3D Python interface is approximately half the length of the implementation using the Open3D C++ interface, and about five times shorter than the implementation based on PCL.
  (For reference, see Design section in the publication)
- It is a very good model of binding generation, using minimal tools (pybind11). The only downside is the lack of automation, which would require changes in the interface files whenever the underlying C++ API changes.

Analysis of available support tools and libraries for binding generation

Comparison links used in multiple headings: CL#1, CL#2, CL#3, CL#4, CL#5, CL#6, CL#7

Why use tools?

There is a performance gain making bindings by hand, but the gain will be very small compared to the effort required to do so. We will need to maintain the interface by hand, and this is not an option if the module is large (as in the case of PCL).

Some discussion points

- A general shortcoming of all wrapper generators is that many users eventually reach the limits of their capabilities, be it in terms of performance, feature support, language integration to one side or the other, or whatever.
- From that point on, users start fighting the tool in order to make it support their use case at hand, and many projects start over from scratch with a different tool.
- Therefore, most projects are better off starting directly with a manually written wrapper, at least when the part of the native API that they need to wrap is not prohibitively vast.
- If the use case is vast, this might not be an option especially if the library is still under active development.

Boost.Python

(See documentation)

- Boost.Python is an open-source C++ library which provides a concise IDL-like interface for binding C++ classes and functions to Python, using pure C++.
- Boost.Python attempts to maximize convenience and flexibility without introducing a separate wrapping language. It presents the user with a high-level C++ interface for wrapping C++ classes and functions, managing much of the complexity behind-the-scenes with static metaprogramming.
- Boost.Python also provides:
    - Support for C++ virtual functions that can be overridden in Python.
    - Comprehensive lifetime management facilities for low-level C++ pointers and references.
    - Support for organizing extensions as Python packages, with a central registry for inter-language type conversions.
    - A safe and convenient mechanism for tying into Python's powerful serialization engine (pickle).
    - Coherence with the rules for handling C++ lvalues and rvalues.

User-guided wrapping

- As much information is extracted directly from the source code to be wrapped as is possible within the framework of pure C++, and some additional information is supplied explicitly by the user.
- Mostly the guidance is mechanical and little real intervention is required.
- Because the interface specification is written in the same full-featured language as the code being exposed, the user has unprecedented power available when he/she does need to take control.

The advantage over Python C API

- The Python 'C' API only gives us a way of extracting signed integers. If the original function accepts an unsigned integer, the Boost.Python version will raise a Python exception if we try to pass a negative number to it.
- If the C++ function is called and it throws an exception, the exception propagates across the boundary with code generated by a 'C' compiler, and it will cause a crash. Functions wrapped by

Boost.Python automatically include an exception-handling layer which protects Python users by translating unhandled C++ exceptions into a corresponding Python exception.

Pros

- It's a very complete library. It allows the user to do almost everything that is possible with the C-API, but in C++. It has tons of tooling support like numpy to and from eigen, stdlib to and from python, etc.
- Very stringent, almost no bugs. It will either work really well or will not compile.

Cons

- Boost.Python's dynamic type conversion registry allows users to add arbitrary conversion methods.
- As mentioned on the Gitter chat: “verbose, old, slow to compile, uses more memory than pybind11 while providing a subset of features.”
- Modules generated are heavier as compared to pybind11.
- The repository is inactive (last commit on the master about a year ago).

Final notes

- Boost.Python requires more manual work to expose C++ object functionality, but it is capable of providing all major functions.
- The reason not to use it in this project are its cons, coupled by how extremely few projects use it for their bindings purpose, thus making it a not-so-popular choice even though it is very well designed and feature complete.

CFFI: C Foreign Function Interface for Python

(See documentation)

- CFFI is Python with a dynamic runtime interface to native code.
- CFFI is the dynamic way to load and bind to external shared libraries from regular Python code.

Pros

- It is similar to the ctypes module in the Python standard library, but generally faster and easier to use.
- Also, it has very good support for the PyPy Python runtime, still better than what Cython and pybind11 can offer.

Cons

- The runtime overhead prevents it from coming any close in performance to the statically compiled code that Cython and pybind11 generate for CPython.
- The dependency on a well-defined ABI (binary interface) means that C++ support is mostly lacking.

Final notes

- As long as there is a clear API-to-ABI mapping of a shared library, cffi can directly load and use the library file at runtime, given a header file description of the API.
- In the more complex cases (e.g. when macros are involved), cffi uses a C compiler to generate a native stub wrapper from the description and uses that to communicate with the library.
- That raises the runtime dependency bar quite a bit.

- In a nutshell, “If performance is not important, if dynamic runtime access to libraries is an advantage, and if users prefer writing their wrapping code in Python, then CFFI will do the job, nicely and easily.”
- Hence, it is not a very good approach for PCL use.

CLIF

(See source)

Implementation

- CLIF provides a common foundation for creating C++ wrapper generators for various languages.
- It consists of four parts:
    1. **Parser**: The parser converts a language-friendly C++ API description to the language-agnostic internal format and passes it to the matcher.
    2. **Matcher**: The matcher parses selected C++ headers with Clang (LLVM's C++ compiler) and collects type information. That info is passed to the Generator.
    3. **Generator**: The generator emits C++ source code for the wrapper. The generated wrapper needs to be built according to language extension rules. Usually that wrapper will call into the Runtime.
    4. **Runtime**: The runtime C++ library holds type conversion routines that are specific to each target language but are the same for every generated wrapper.
- It does not discover the API, we need to describe it in Python terms.
- CLIF will recompile the header with the latest LLVM/Clang compiler and produce C++ source code for a Python extension module.

Final notes

- Although it is a very vast tool, it has not been popularly used in the community (maybe was used for internal Google purposes). It is also very inactive (14 commits in total, the latest about 2 years ago).

Cython

- Cython is a Python compiler and a complete programming language that is used to implement actual functionality and not just bind to it.
- Cython is Python with native C/C++ data types, or it is a static Python compiler.
- For people coming from a Python background, it is much easier to express their coding needs in Python and then optimising and tuning them, than to rewrite them in a foreign language.
- Cython allows them to do that by automatically translating their Python code to C, which often avoids the need for an implementation in a low-level language.

Implementation

- Cython uses C type declarations to mix C/C++ operations into Python code freely, be it the usage of C/C++ data types and containers, or of C/C++ functions and objects defined in external libraries.
- There is a very concise Cython syntax that uses special additional keywords (cdef) outside of Python syntax, as well as ways to declare C types in pure Python syntax.

Pros

- When it comes to wrapping native libraries, Cython has strong support for designing a Python API for them.
- Being Python, it really keeps the developer focussed on the usage from the Python side and on solving the problem at hand, and takes care of most of the boilerplate code through automatic type conversions and low-level code generation.

Cons

- Requires learning a new language.
- Very verbose.

How to bind?

- We write code in special .pyx files.
- Those files are compiled (translated) into C code which in turn are compiled to CPython modules.
- Code in Cython files can call both pure Python functions but also C and C++ functions (and also C++ methods).
- For PCL use case, to keep the C++ API strictly separated from the exposure sources (.pyx extensions), the workflow has to be :

  > C++ API -> Compilation as shared object -> Cython extension(s) (importing and exposing C++ features) -> Compilation of the extension(s) -> Using the extension (extension to be added to your python path).

Final notes

- We would have to export all functionality of the C++ API in Cython. This means that we will not be using the underlying C++ API for Python use.
- Using Cython would make more sense when we want to implement a new library, not as a binding solution for a pre-existing C++ library.

CPPYY

(See documentation)

Implementation

- cppyy is an automatic, run-time, Python-C++ bindings generator, for calling C++ from Python and Python from C++.
- Run-time generation enables detailed specialization for higher performance, lazy loading for reduced memory use in large scale projects,
- It originated from ROOT bindings and is built on top of Cling. Additionally, it is JIT, so it can handle templates.

Final notes

- The header code runs in Cling, which is a downside.
- Furthermore, it has heavy user requirements.
- Due to unpopular usage, we’ll not be considering it. It has its upsides such as ease of work, but it won’t work to generate bindings.

SWIG

(See documentation)

I made some support documentation about it in my fork, here are some points from that:

- SWIG is a software development tool that connects programs written in C/C++ with high-level programming languages.
- 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 can also export its parse tree in the form of XML.
- SWIG can be used to turn common C/C++ libraries into components for use in popular scripting languages.

Implementation

- SWIG is an interface compiler. It works by taking the declarations found in C/C++ header files and uses them to generate the wrapper code that scripting languages need to access the underlying C/C++ code.
- In addition, SWIG provides a variety of customization features that let us tailor the wrapping process to suit our application.
- Some relevant points:
    - **ISO C/C++ syntax**:
        - SWIG parses ISO C++ that has been extended with a number of special directives.
        - As a result, interfaces are usually built by grabbing a header file and tweaking it a little bit.
        - This particular approach is especially useful when the underlying C/C++ program undergoes frequent modification.
    - **SWIG does not define a protocol nor is it a component framework**:
        - SWIG does not define mechanisms or enforce rules regarding the way in which software components are supposed to interact with each other. Nor is it a specialized runtime library or alternative scripting language API.
        - SWIG is merely a code generator that provides the glue necessary to hook C/C++ to other languages.
    - **Designed to work with existing C/C++ code**:
        - For the most part, it encourages keeping a clean separation between C/C++ and its scripting interface.

Pros

- We can generate bindings for many scripting languages.
- Robust, mature and can handle ISO C++ in its entirety.
- Very easy to use (just one interface file including the header, and we're done).
- SWIG provides control over most aspects of wrapper generation. Most of these customization options are fully integrated into the C++ type system--making it easy to apply customizations across inheritance hierarchies, template instantiations, and more.

Cons

- Because the process is very automatic, it sometimes requires extra hints to the swig parser.
- Some excerpts from the Gitter chat:
  "Cons of SWIG while holding my breath
    - apparently very slow in function calls

    - has issues with C++ parsing (similar to doxygen)
    - needs interface files, which adds tons of new files (unless they are generated automatically, say hello to a script which works on blobs to create them for you, which might not work perfectly, so complicate the script with more exceptions.... and blackhole)
    - another language (apart from the wrapped lang and C++)
    - no strong support for lambdas, auto-vectorization, numpy for eigen, stdlib for python in-built, etc.

  The pro of swig is multi-language support, ISO standard and that's about it.”

  “Apparently, projects found that less bothersome than swig; which is one of my reasons on hesitation towards swig (see OpenCV which literally generates bindings using python to write cpp code)”
- In order to wrap a template, we need to give SWIG information about a particular template instantiation : something that will not work in the case of PCL, which uses heavy templating (documentation unclear on this one).

Final notes

- It is very easy to install and use SWIG. It generates decent binding that is robust and versatile, and that one interface file allows the C++ code to be available from several other languages like LUA, C#, and Java. But unless we really need multi-language support, we can go for more tailored approaches.
- Its level of automacy (one interface file which includes the header, and that’s it) can be very pleasant to use but might give rise to hidden problems that cannot be solved later on.
- Thus, although being a very mature project and my initial approach, I shifted away from it because of its mixed reviews and projects migrating away from it.

PyBindGen

(See archive)

Implementation

- It generates the code dealing with the C-API.
- We can either describe functions and classes in a Python file, or let Pybindgen read the headers and generate bindings automatically (for this it uses pygccxml).

Pros

- PyBindGen generates Python C API calls directly, there is no speed-robbing indirection layer like SWIG.
- PyBindGen is highly portable:
    - The (optional) pygccxml-based scanner generates just Python code describing the API
    - PyBindGen as a module is just a bunch of .py files that can be shipped inside another project and used without installation
    - It generates C/C++ code that is self-contained at compile-time (only Python headers are required) and at run-time (no special library or Python module is required).
- Claims to be the fastest among peers.

Cons

- Callbacks for higher order functions are not supported.

- Not mature (still on version 0.2.0).
- It does not support Python versions 3.0, 3.1, and 3.2.
- The automatic header file scanning feature of PyBindGen does not work in any Python 3.x version, since the pygccxml package has not been ported to Python 3.

Final notes

- It has a great design, but unfortunately due to lack of maturity and inactive repository (last release about a year ago), we cannot use it.

Pybind11

(See limitations, source)

Implementation

- pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, mainly to create Python bindings of existing C++ code.
- Its goals and syntax are similar to the Boost.Python library: to minimize boilerplate code in traditional extension modules by inferring type information using compile-time introspection.
- It is a tiny self-contained version of Boost.Python with everything stripped away that isn't relevant for binding generation. Without comments, the core header files only require ~4K lines of code and depend on Python (2.7 or 3.x, or PyPy2.7 >= 5.7) and the C++ standard library.

Pros

- Supports lambda functions, slice-based access, assignment operations and uses move operators (for efficient transfer of custom data types) support.
- It's easy to expose the internal storage of custom data types through Pythons' buffer protocols. This is handy e.g. for fast conversion between C++ matrix classes like Eigen and NumPy without expensive copy operations.
- pybind11 can automatically vectorize functions so that they are transparently applied to all entries of one or more NumPy array arguments.
- Function signatures are pre computed at compile time (using constexpr), leading to smaller binaries.

Cons

- It lacks support for pointers to pointers.
- pybind11 casts away const-ness in function arguments and return values. This is in line with the Python language, which has no concept of const values. This means that some additional care is needed to avoid bugs that would be caught by the type checker in a traditional C++ program.
- The NumPy interface pybind11::array greatly simplifies accessing numerical data from C++ (and vice versa), but it's not a full-blown array class like Eigen::Array or boost.multi_array.

“These features could be implemented but would lead to a significant increase in complexity. I’ve decided to draw the line here to keep this project simple and compact.” - The author

Final notes

- Binaries are generally smaller by a factor of at least 2 compared to equivalent bindings generated by Boost.Python. A recent pybind11 conversion of PyRosetta, an enormous Boost.Python binding project, reported a binary size reduction of 5.4x and compile-time reduction by 5.8x.
- It is easily the most popular approach for generating Python bindings currently due to its lightweight nature. Projects like Tensorflow, LSST, etc. have already moved away from their old implementation in support of Pybind11, which makes it my favourite choice for developing bindings for PCL.

- The project is very active and mature, with good documentation and popular reviews. It is definitely the de facto tool right now for python bindings purposes.

Other Projects

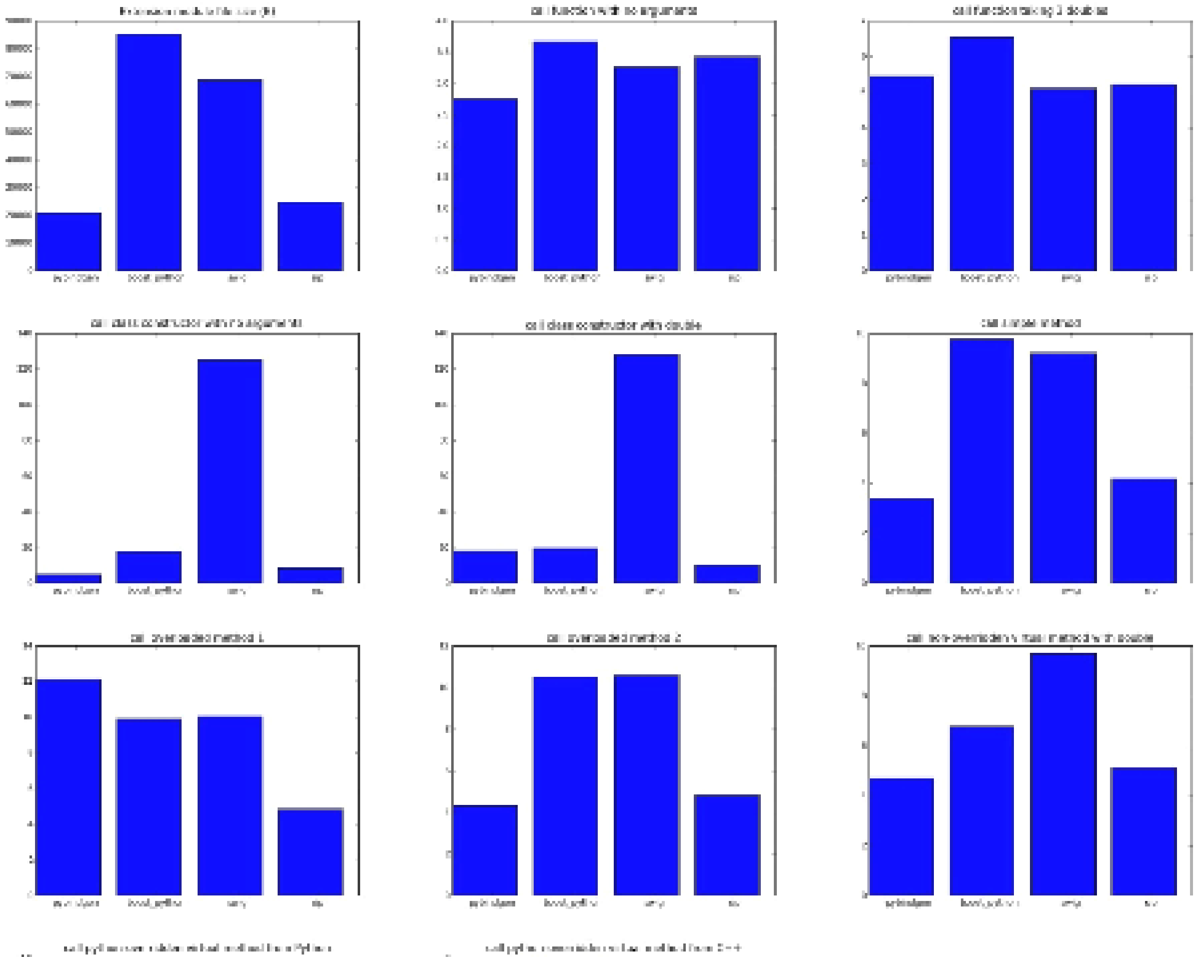
There are a plethora of projects out there, I cannot cover everything in detail. Some may be discontinued, others not popular, not for binding purposes, etc.

- **Ctypes**: It provides full access to the native C interface and support for loading and interfacing with dynamic libraries, such as DLLs or shared objects, at runtime. It brings along with it a whole host of types for interacting with system APIs, and allows you to rather easily define your own complex types, such as structs and unions, and allows you to modify things such as padding and alignment, if needed.
- **Py++**: Py++ is an object-oriented framework for creating a code generator for Boost.Python library and ctypes package. It uses a C++ parser to read the code and then generates Boost.Python code automatically.
- **PyCXX**: C++ facilities to make it easier to write Python extensions.
- **SIP**: SIP is a toolset for generating Python bindings that was developed for the PyQt project. It has a code generation tool and an extra Python module that provides support functions for the generated code.
- **Shiboken**: Shiboken is a tool for generating Python bindings that's developed for the PySide project associated with the Qt project.

Comparison figures

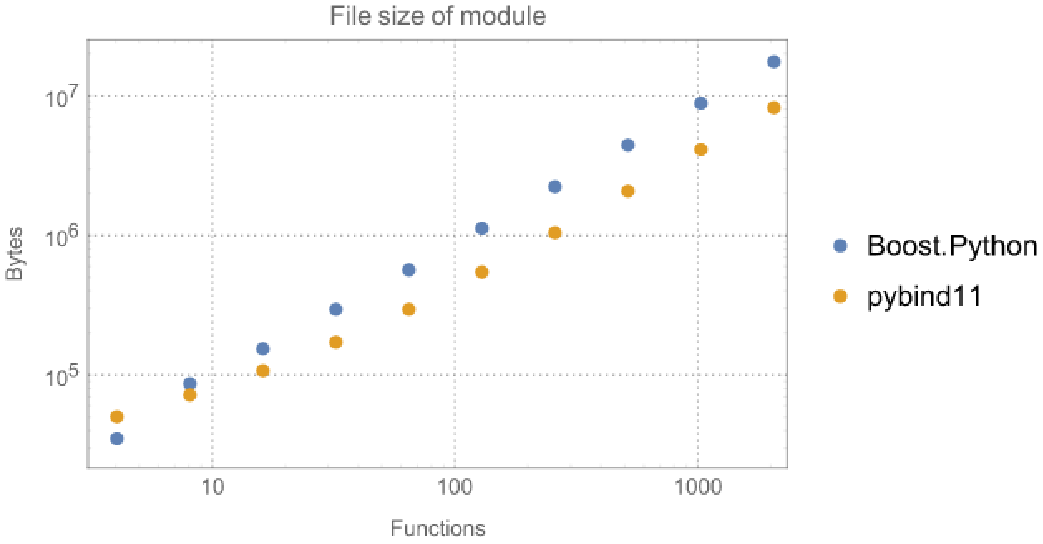
Compiled set of benchmarking graphs (source code by pybindgen)

(Compiled from: source)



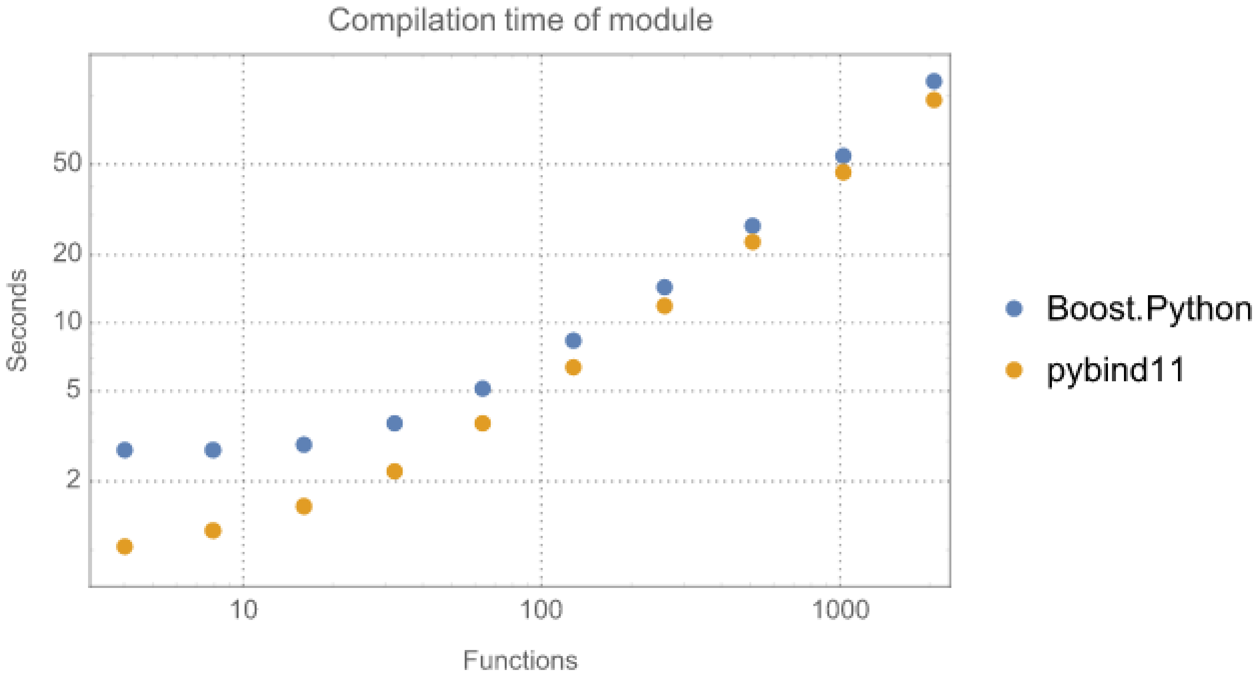
Compilation time benchmarking (by pybind11)

(source)



File size of module benchmarking (by pybind11)

(source)



Cppyy research paper benchmarking

(source)

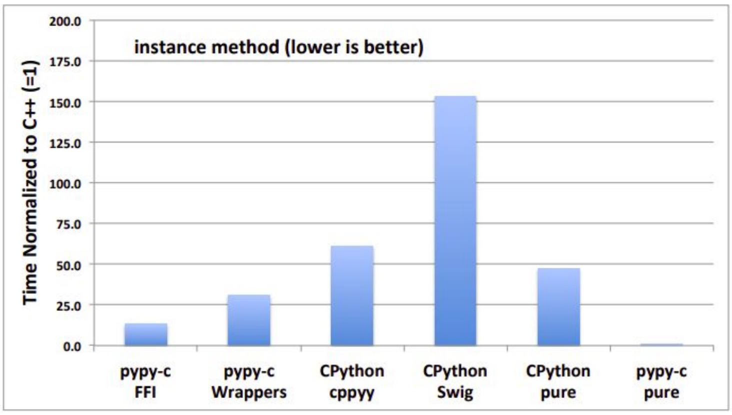


Fig. 1: *Relative performance (compared to C++) of the FFI and wrapped paths for instance method calls. CPython using cppyy, SWIG, and pure Python calls are shown for reference.*

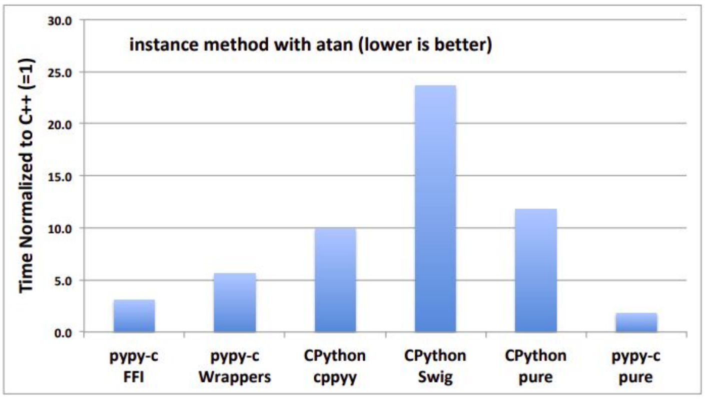


Fig. 2: *Relative performance (compared to C++) of the FFI and wrapped paths when calling a non-trivial function. CPython using cppyy, SWIG, and pure Python calls are shown for reference.*

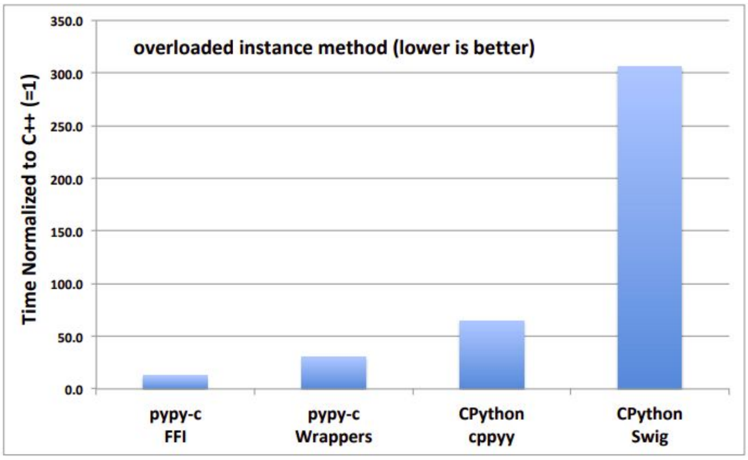


Fig. 3: *Relative performance (compared to C++) of the FFI and wrapped paths for overloaded instance method calls. CPython using cppyy and SWIG are shown for reference.*

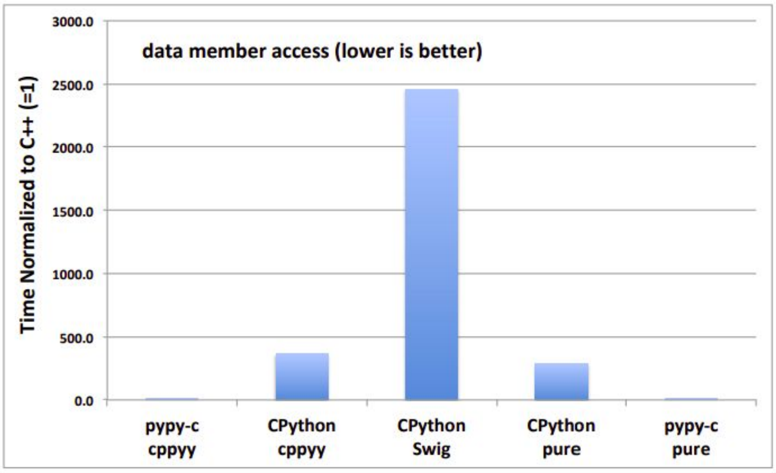


Fig. 4: *Relative performance (compared to C++) of data member access in* `pypy-c`*. CPython using cppyy, SWIG, and pure Python are shown for reference.*

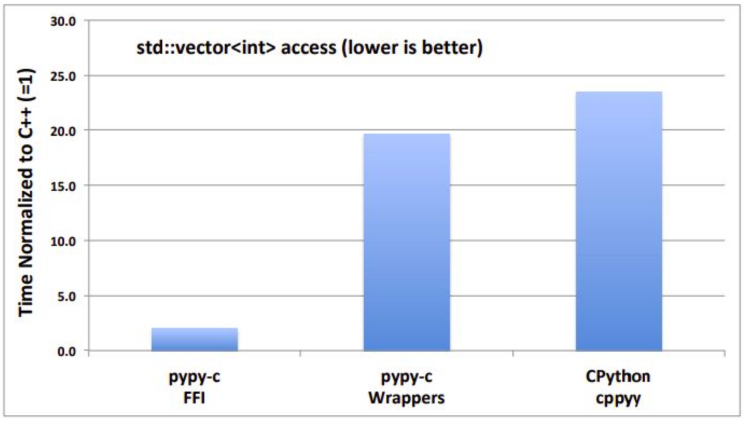


Fig. 5: *Relative performance (compared to C++) of looping over an STL vector in* `pypy-c` *and CPython using cppyy. SWIG not shown: it is ~ $350x$ slower.*

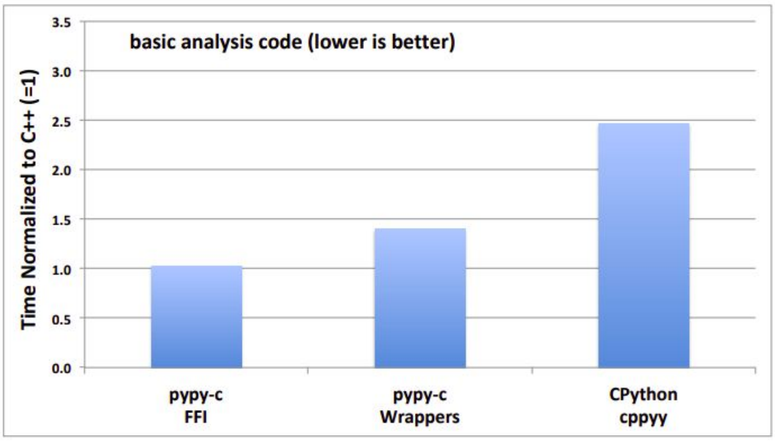


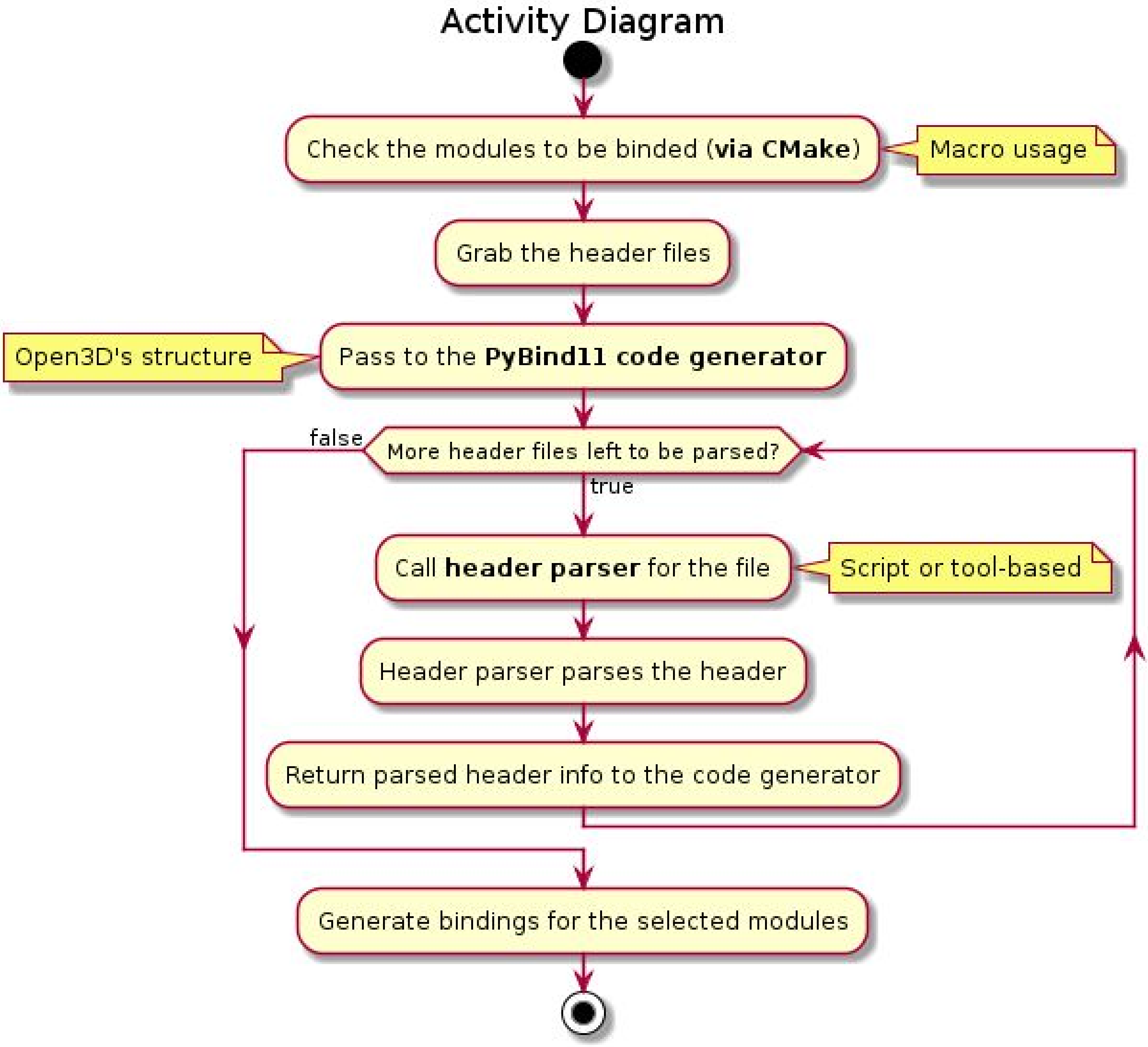
Fig. 6: *Relative performance (compared to C++) of a basic analysis code in* `pypy-c`*. CPython using cppyy is shown for reference.*

Proposal

(Finally!)

The idea

- My proposed idea for the generation of bindings takes inferences from the already applied and in-use techniques used by projects like Open3D and OpenCV.
- I basically want to replicate Open3D's model of binding generation with the addition of automation for header parsing (Open3D currently uses handwritten files).
- Oh, and yes, all of this will have to be compatible with the current PCL's CMake build system (of course).


Activity Diagram
Check the modules to be binded (**via CMake**)
Macro usage
Grab the header files
Open3D's structure
Pass to the **PyBind11 code generator**
false
More header files left to be parsed?
true
Call **header parser** for the file
Script or tool-based
Header parser parses the header
Return parsed header info to the code generator
Generate bindings for the selected modules

(See svg, png)

Choosing Pybind11

- Many projects are using or shifting towards pybind11. There are many reasons for it (already mentioned), which enhances my trust in using the tool for the PCL task.
- Open3D uses pybind11 to generate it's python bindings, and I will be modifying its pattern for this project.
- A thing to keep in mind is I want to avoid tools such as binder which although, would make my life very easy, but are not being used by any pybind11 using organisation due to its lack of extensibility and limited use case (only works for pybind11, cannot use its output for other languages' bindings)

Initial proof of concept using SWIG

- I should mention that my initial approach to generate bindings was to use SWIG, and it had some shortcomings.
- SWIG only required interface files (.i files) to generate bindings for multiple languages. This file was pretty customizable (support for macros, ifdef, etc.) which is both a pro and a con. If a certain feature was not supported by swig, we can just modify the interface file according to our needs.
- Keeping in mind the size of PCL, if a certain feature (like template instantiation) is not supported by just including the header file, we would have to manually handle that which would be very cumbersome.
- Along with these, all the cons mentioned in the SWIG portion in this document should also be kept in mind.
- The major reason to move away from the SWIG approach (as discussed on the Gitter chat) was the shift of projects from SWIG to pybind11 (TensorFlow, LSST to name a few) which reduced my confidence in it.

Open3D's model in detail

Folder skeleton

- Open3D's python bindings folder arrangement:

```
Open3d_pybind
        |_camera
              |_camera.cpp
              |_camera.h
        |_color_map
              |_camera.cpp
              |_camera.h
        |_geometry
              |_camera.cpp
              |_camera.h
        ...
        |_docstring.cpp
        |_docstring.h
        |_open3d_pybind.cpp
        |_open3d_pybind.h
```

A module in-depth

- The workings for a module (let's take camera) goes like this:
    - Camera.h
        - Include the main pybind header file

          ```
          #include "open3d_pybind/open3d_pybind.h"
          ```

        - Define the single function for generating bindings

```
void pybind_camera(py::module &m);
```

- Camera.cpp
  - Include the C++ header files

```
#include "Open3D/Camera/PinholeCameraIntrinsic.h"
#include "Open3D/Camera/PinholeCameraTrajectory.h"
```

  - Include the pybind support header files

```
#include "open3d_pybind/camera/camera.h"
#include "open3d_pybind/docstring.h”
```

  - Pybind11 code for the module camera

```
void pybind_camera_classes(py::module &m) {
   // handle default constructors, functions, enums, classes, templates, etc.
}
```

  - Define the main function

```
void pybind_camera(py::module &m) {
   // define camera as a module
   py::module m_submodule = m.def_submodule("camera");

   // call the class method
   pybind_camera_classes(m_submodule);
}
```

Binding all modules

- The main files for generation of bindings for all the modules:
  - Open3d_pybind.h
    - Include pybind11 header files

```
#include <pybind11/detail/internals.h>
#include <pybind11/eigen.h>
#include <pybind11/functional.h>
...
```

    - Include some Open3D support headers

```
#include "Open3D/Registration/PoseGraph.h"
#include "Open3D/Utility/Eigen.h"
```

    - Define namespaces and typedefs

```
namespace py = pybind11;
using namespace py::literals;
typedef std::vector<Eigen::Matrix4d, open3d::utility::Matrix4d_allocator>
        temp_eigen_matrix4d;
typedef std::vector<Eigen::Vector4i, open3d::utility::Vector4i_allocator>
```

```
    temp_eigen_vector4i;
```

- Use PYBIND11_MAKE_OPAQUE(T) to disable the template-based conversion machinery of types, thus rendering them opaque.
  The contents of opaque objects are never inspected or extracted, hence they can be passed by reference.

```
PYBIND11_MAKE_OPAQUE(std::vector<int>);
PYBIND11_MAKE_OPAQUE(std::vector<double>);
PYBIND11_MAKE_OPAQUE(std::vector<Eigen::Vector3d>);
...
```

- Some helper functions

```
namespace pybind11 {
namespace detail {

template <typename T, typename Class_>
void bind_default_constructor(Class_ &cl) {
    cl.def(py::init([]() { return new T(); }), "Default constructor");
}
template <typename T, typename Class_>
void bind_copy_functions(Class_ &cl) {
    cl.def(py::init([](const T &cp) { return new T(cp); }), "Copy constructor");
    cl.def("__copy__", [](T &v) { return T(v); });
    cl.def("__deepcopy__", [](T &v, py::dict &memo) { return T(v); });
}

}  // namespace detail
}  // namespace pybind11
```

- Open3d_pybind.cpp

  - Include header files for all modules

```
#include "open3d_pybind/open3d_pybind.h"
#include "open3d_pybind/camera/camera.h"
#include "open3d_pybind/color_map/color_map.h"
...
```

  - Define the PYBIND11_MODULE

```
PYBIND11_MODULE(open3d, m) {
    m.doc() = "Python binding of Open3D";

    // Register this first, other submodule (e.g. odometry) might depend on this
    pybind_utility(m);

   // call all single-functions for generating bindings, as specified in the module
header files
    pybind_camera(m);
    pybind_color_map(m);
    pybind_geometry(m);
    pybind_integration(m);
    pybind_io(m);
    pybind_registration(m);
    pybind_odometry(m);
    pybind_visualization(m);
}
```

Notes

- And that's it. It is very clean, elegant and maintainable. Separate modules contain code for each of them with individual header files, and a main C++ file just calls the binding generation function for each module. It gives us modularity and control over which module's binding to generate.
- In addition, there are also `docstring.cpp` and `docstring.h` files. They can be used to control the docstrings in Python (uses regex and maps) to make use of `help()` functionality in Python.
    - As discussed on my POC's Feedback page, this functionality is optional, since the IDE's intelligence will suffice for giving information about the return types, function arguments, etc.
    - Nevertheless, it can be an additional functionality to be implemented at a later stage.

Lack of automation: OpenCV's inference

- The repo and the documentation doesn't hint of any automation for this process, which will be needed, else maintainability will be a big issue.
- The automation will need 2 things:
    1. A header parser, which scans the header files for all modules, and extracts information about the classes, functions, templates,etc and makes it into some sort of structure.
    2. A script, which takes the header parsed structure and generates the code for all the pybind11 interface files (code for the binding function in each module) in a pybind11 format.

Header parsing solution

- This problem can be solved via either the script or the tool approach.
- **Script method**:
    - Taking inference from the OpenCV header parser, we can write a similar script for PCL
    - **OpenCV's header parser**:
        - The header parser splits the complete header file into small Python lists. So these lists contain all details about a particular function, class etc. For example, a function will be parsed to get a list containing function name, return type, input arguments, argument types etc.
        - The final list contains details of all the functions, structs, classes etc. in that header file.
        - The header parser doesn't parse all the functions/classes in the header file. The developer has to specify which functions should be exported to Python. For that, there are certain macros added to the beginning of these declarations which enables the header parser to identify functions to be parsed. These macros are added by the developer who programs the particular function. In short, the developer decides which functions should be extended to Python and which are not.
- **Tool based approach**:
    - Many tools can be used, to parse the header files:
        - **Cppheaderparser**
            - Cppheaderparser parses the C++ header files and generates a data structure representing the class.
            - Used by Pypcl
            - External modules required: PLY
        - **GCC_XML**

            - GCC_XML extension generates an XML description of a C++ program from GCC's internal representation.
            - Since XML is easy to parse, other development tools will be able to work with C++ programs without the burden of a complicated C++ parser.
            - Supports C++ in its entirety.
            - It is deprecated.
        - **Cast_XML**
            - CastXML is a C-family abstract syntax tree XML output tool.
            - Has limitations, such as no template support.

- **Note**: There is an active discussion currently going on regarding Script v/s Tools.

Why not binder?

(See source)

- Binder is the tool used for automatic generation of bindings using pybind11.
- Concerns for not using binder:
    - It requires a separate input file(s) that lists all header files. To keep it updated, we will need another scripting mechanism. (more scripting and more files to maintain)
    - Configuration:
        - It is too automatic for most projects. Thus, it requires fine-tuning via configuration.
        - It adds another configuration file to maintain.
    - Binder is an inextensible solution. It can only be used with pybind11. We will not be able to reuse its code/output for use in other language bindings.
- No major open source projects which are using pybind11 are using it, which means the task can be done without it. It gives rise to reliability concerns.

Scripting solution

- A script, which takes the header parsed structure and generates the code in a pybind11 format can be made following the OpenCV model.
- They use a CMake script which checks the modules to be extended to Python. It will automatically check all the modules to be extended and grab their header files.
- These header files are passed to a Python bindings generator script. This calls another script which parses each header file.
- In our case, the header parsing can be done via a script or some XML parser.
- The generator file in our case will generate pybind11's header-based code, rather than the custom code for binding generation, as in the case of OpenCV.

PyPi integration

The process for PyPi integration is very straightforward:

1. Clean the code: remove print statements, define `__main__`, etc.
2. Create a package (will already be done)
3. Create PyPi files:
    - `setup.py`: contains information about the package like its name, a description, the current version etc.
        - Download_url: link to source code of the latest release.
        - Install_requires: defining dependencies of the package (ex. numpy)
    - `setup.cfg`: metadata purposes
    - LICENCE and README
4. Create a PyPI account and a source distribution and upload the package (via `twine`).

And that's it.

Testing

(See Testing Your Code)

This is an area in which I have to research more extensively.

Although here are some approaches:

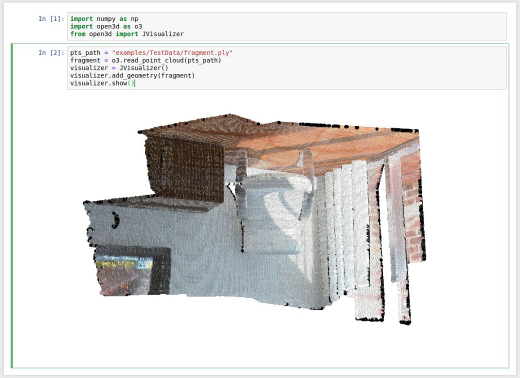
- **Unittest**: The included test module in the Python standard library.
- **Doctest**: The doctest module searches for pieces of text that look like interactive Python sessions in docstrings, and then executes those sessions to verify that they work exactly as shown.
- Some Tools:
  - **Py.test**: It is a no-boilerplate alternative to Python's standard unittest module.
  - **Hypothesis**: It is a library which lets us write tests that are parameterized by a source of examples. It then generates simple and comprehensible examples that make the tests fail, letting us find more bugs with less work.
  - **Tox**: It is a tool for automating test environment management and testing against multiple interpreter configurations.
  - **Mock**: unittest.mock is a library for testing in Python. As of Python 3.3, it is available in the standard library.

Stretch Goals

- I have kept the project very under-scoped (just bindings for Python) because I have done extensive research only for developing Python bindings (due to its high priority nature).
- Some extra functionalities/more bindings require some more research on my part, and that is why I have excluded them from my proposed approach.
- **Note**: I am planning on completing them along with the Python bindings, these are not “just in case” things. So, these are also a part of the timeline, they are not “stretch goals” in a literary sense.

Jupyter Visualization

- It takes influence from Open3D's jupyter visualisation (experimental) which makes use of WebGL.
- It takes point clouds and generates their visualisations in the Jupyter notebook.
- This functionality can be added only after we develop Python bindings, so it can be treated as an add-on feature.
- Open3D's Jupyter Visualization has some limitations:
  - Only point cloud geometry is supported.
  - Camera is initialized with fixed parameters, therefore, the initial view may not be optimal for the point cloud.
  - Performance is not optimized.
- We can follow a similar approach or look into some widgets supported by Jupyter.
- An example (source):



```
In [1]: import numpy as np
        import open3d as o3
        from open3d import JVisualizer
```

```
In [2]: pts_path = "examples/TestData/fragment.ply"
        fragment = o3.read_point_cloud(pts_path)
        visualizer = JVisualizer()
        visualizer.add_geometry(fragment)
        visualizer.show()
```

Docstrings addition

- Docstrings in Python is the documentation string which is string literal, and it occurs in the class, module, function or method definition, and it is written as a first statement.
- Docstrings are accessible from the doc attribute for any of the Python objects and also with the built-in help() function can come in handy.
- Through this, all of the documentation that is available in the header files can be made available through python.
- This feature is optional (as discussed on my POC's Feedback) as:
    - IDE's intelligence will suffice for end-users (gives hints about return types, function arguments, etc.)
    - This feature is good for developers but since the development happens in C++, this requirement is already satisfied.
- This makes it a low priority feature but nevertheless, it will be a nice add-on to have.
- VTK also supports it, along with Open3d. I found Open3d's model very elegant (just one CPP file to handle it), and that's probably what I'll be following.

Javascript Bindings

- The next bindings to consider are JS bindings.
- I have an inclination to use **emscripten**, much due to the suggestion of mentors.
    - Emscripten compiles C and C++ to WebAssembly using LLVM and Binaryen.
    - Emscripten output can run on the Web, in Node.js, and in wasm runtimes.
- I have to research more in the area to look for more approaches/alternatives (like WebIDL).

Timeline

Pre-GSOC (prior to May 4, 2020)

- Study Pybind11 and start on the proof of concept
    - The pybind11, although a lightweight library, is very extensive and requires time to be studied.
    - Study pybind11's in-detail workings and also how other organizations are using it in detail.
    - Start on the proof of concept cum commencing solution.
- Start with the header parsing problem
    - Study OpenCV's header parser, understand it's workings and what can/cannot be used for our case.
    - Look into cppheaderparser, CastXML and GCC_XML, their functionalities, overheads, usage, etc.
    - Finalise to use the script approach or the tool-based approach.
- Study CMake
    - A significant part of the project is dealing with the integration with CMake.
    - Brush up on CMake, look into how PCL uses it.
    - Use CMake for selecting modules for generation of bindings.

Community Bonding (May 4, 2020 - June 1, 2020)

- Although I have been active in the PCL community and learned a lot, there is still a huge amount of PCL unexplored from my side, I'll continue my contributions, try solving issues, read up on documentation etc.
- I'll also try to help out fellow students who haven't yet worked on the repo or are just starting their PCL journey.
- On the project's side:
    - Finish off the work I started in the pre-GSoC phase.

    - Start work on the script to generate pybind11 bindings. This will also include details such as structuring the generated files, etc.

Coding (June 1, 2020 - August 24, 2020)

- **Note**: Documentation will be done alongside implementation.
- **Week 1 and 2 (June 1, 2020 - June 14, 2020)**
    - Continue work on scripting for generation of bindings for all modules.
- **Week 3 and 4 (June 15, 2020 - June 28, 2020)**
    - Finish pybind11 bindings generation.
    - Start the testing procedure: study test-tools that can be leveraged for our case.
- **Week 5 and 6 (June 29, 2020 - July 12, 2020)**
    - Continue with the tests (I believe they will take a significant amount of time)
    - Do PyPI integration for the module.
    - Parallelly, start research on the stretch goals.
- **Week 7 and 8 (July 13, 2020 - July 26, 2020)**
    - Start with Jupyter visualisation and it’s testing (experimental)
    - Start with docstrings/javascript bindings (depending on priority)
- **Week 9 and 10 (July 27, 2020 - August 9, 2020)**
    - Complete all the proposed stretch goals.
    - Try testing for stretch goals (if applicable)
- **Week 11 and 12 (August 10, 2020 - August 23, 2020)**
    - This will serve as a buffer period for the worst-case scenario, for work pending in that case.
    - Most probably everything will be wrapped up before this period, as planned. In that case, start work on some extra stretch goals/ feature implementations after necessary discussions.

Post-GSoC

- Although I’ll try my best to develop a bug-free solution, nothing like that exists. So, I’ll be active within the community for maintaining the python bindings.
- Any new cool features that extend the ability of the current solution, I’ll work on them.
- Personally, GSOC is a great way to get involved in a community, and PCL will serve as the robotics project I’ll be involved in for a long time. I would like to dwell deeper into PCL’s workings, and actively improve and enhance its functionalities.

Biographical Information

I am currently in my third year of study in Information Technology (B.Tech.) at Manipal Institute of Technology, Manipal, India.

I worked on the driverless car at Formula Manipal Driverless, a student project working on Formula style race cars. I developed a SLAM solution for the car (used Error State Extended Kalman Filter) which is now being tested on simulation environments.

I did significant work on Kalman and particle filters during my tenure along with the FastSLAM algorithms and worked with Python, C++, ROS and some simulation environments like CARLA.

Bindings generation is a new project for me and I hope to learn a lot from it.

- **Which city will you be spending this summer in?:** Jaipur, India
- **Do you have a part-time or full-time job/internship planned?:** No
- **How many hours per week can you give for the project?:**
    - My college vacations have been shifted, college is closed till May 31, 2020. After that, my college will be operational for the rest of the GSOC timeline.
    - During the Pre-GSOC (prior to May 4, 2020) and Community Bonding (May 4, 2020 - June 1, 2020) period, I’ll be able to give 40-45 hours per week for the project.

    - During the Pre-GSOC (prior to May 4, 2020) period (college operational), I’ll be able to contribute 25-35 hours weekly on average.
    - I’ll be having end-semester exams from July 06, 2020 - July 18, 2020, during which the working hours will fall significantly.
- **Are you applying for any other organizations/projects this GSOC?:** No
- **Why did you choose PCL?:** GSOC is a way of getting more involved in the PCL community, which I have been wanting to do as a way of contributing more in the open-source robotics area. Plus, it had really cool projects.
- **Contributions to PCL?:** Worked mainly on the issues: pull requests

---