Convert-XY: type-safe interchange of C++ and Python containers for NumPy extensions

Damian Eads (eads@soe.ucsc.edu) – *University of California, 1156 High Street, Santa Cruz* USA
Edward Rosten (er258@cam.ac.uk) – *University of Cambridge, Trumpington Street, Cambridge* UK

We present Convert-XY: a new, header-only template library for converting containers between C++ and Python with a simple, succinct syntax. At compile-time, template-based recursive pattern matching is performed on the static structure of the C++ type to build dynamic type checkers and conversion functions.

Note: proper C++ syntax encloses template parameters with angle brackets. We omit the extra space placed between two adjacent right brackets for space purposes, e.g. `vector<map<int, vector<int>>>` *instead of* `vector<map<int, vector<int> > >`.

Introduction

We present **Convert-XY**: a library for converting arbitrary containers between C++ and Python. Here is a simple "hello world" example of **Convert-XY** in action; it converts a Python dictionary `x` to a C++ object `y`:

```
void hello_world(PyObject *x) {
   map <string, vector<int>> y;
   convert(x, y);
}
```

This function can be called from Python with a dictionary (or any object obeying the mapping protocol) with key objects that can be converted to strings and sequence protocol objects containing objects that can be converted to numbers:

```
hello_world({"a": [1,2,3], "b": (4,5),
             "c": np.array([6,7]),
             "d": (x for x in (8,9))})
```

At compile time, pattern matching is applied to the structure of the C++ type of `y`. This yields two pieces of information: how to check for compatible structure in the Python type (e.g. a dictionary mapping strings to lists, tuples, arrays or sequences is compatible with `map<string, vector<int>>`) and which set of converters is applicable (e.g. sequences vs. dictionaries). Dynamic type checkers and converter functions can be built with this static type information at compile time. Further, the library statically determines when an array buffer pointer can be reused, preventing unnecessary copying which can be very useful for programs operating on large data sets. LIBCVD, a popular computer vision library used in our work, provides a `BasicImage` class, which can be used to wrap two-dimensional image addressing around a buffer pointer. **Convert-XY** avoids copying and allocation by reusing the NumPy buffer pointer whenever it can:

```
void example(PyObject *x) {
   map <string, BasicImage<float>> y;
   convert(x, y);
}
```

In this example, the type of the target object for conversion is known at compile time; it's an STL map. The compiler's template pattern matching algorithm matches on the structure of this type to build a dynamic type checker to ensure every part of the source Python object `x` follows compatible protocols, e.g. the outermost part of `x` follows the mapping protocol. In Python, we construct a dictionary of NumPy arrays containing integers.:

```
def example():
   hello = np.zeros((50,50), dtype='i')
   x = {'hello': hello}

   # Use your favorite wrapping tool to call C++
   # (e.g. Boost, Weave, Python C API)
   cpp.example(x);
```

Since the buffer is reused in this example and the NumPy array contains elements of a different type than the `BasicImage` objects in the STL map, a run time exception results, "`Expected a contiguous NumPy array of type code (f)`".
Unlike `BasicImage` objects, LIBCVD's `Image` objects allocate and manage their own memory. In this situation, the source `x` can be discontiguous and the array elements can be of a different type than in the source `x`.

Background

NumPy offers a succinct syntax for writing fast, vectorized numerical algorithms in pure Python. Low-level operations on large data structures happen in C, hidden from the user. The broader SciPy toolset (SciPy, NumPy, Mayavi, Chaco, Traits, Matplotlib, etc.) is a framework for developing scientific applications, simulations, GUIs, and visualizations [Jon01]. The simplicity and expressiveness of Python makes scientific computation more accessible to traditionally non-programmer scientists.
Writing extensions is tedious and inherently error-prone, but sometimes necessary when an algorithm can't be expressed in pure Python and give good performance. Additionally, a significant number of physics and computer vision codes are already written in C++ so Python/C++ integration is an unavoidable problem in need of good solutions.
Interfacing between C/C++ and Python can mean several different things; the figure illustrates three cases. First, wrapping C/C++ functions exposes their

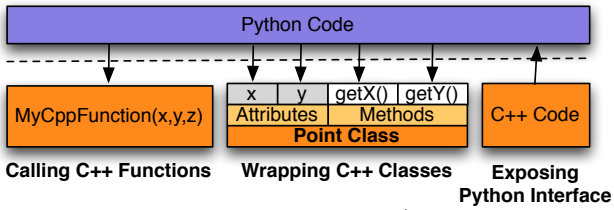

Figure 1 *Existing tools expose C/C++ functions, structures, and objects from Python and Python functions and data from C/C++.* ***Convert-XY*** *converts between arbitrary C++ and Python containers.*

interface so they can be called from Python. This is a well-explored problem with several mature tools in mainstream use including Boost Python [Abr03], SWIG [Bea95], Cython [Ewi08], Weave [Jon01], ctypes [Hel00], Py++ [Yak09], and the Python C API [GvR92] to name a few. Second, wrapping C++ structs and classes exposes user-defined data structures in Python. Boost, SWIG, and Py++ are very well-suited to interface problems of this kind. Third, exposing Python from C++ enables C++ code to manipulate Python objects. PyCXX serves this purpose. It wraps the Python Standard Library with C++ classes, performing reference counting and memory management automatically using C++ language features [Sco04].

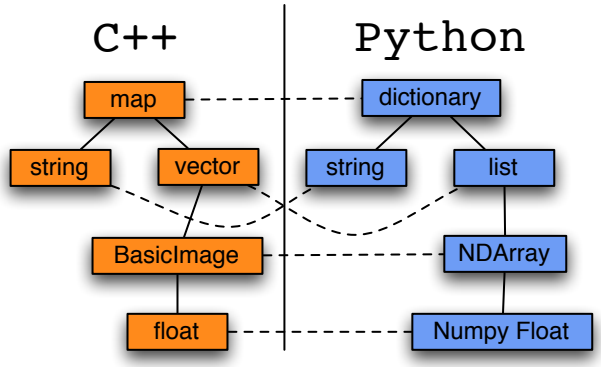

Figure 2 *Helper templates deduce sensible mappings and conversion behavior between Python and C++ types at compile time.*

Convert-XY is not meant to replace any of the other extension tools because it addresses a different problem altogether: how to convert objects between Python and C++ automatically.

SWIG supports typemaps which enable the developer to define how data is converted from Python to C++ and vice versa.:

```
namespace CVD {
  template<class T> class BasicImage;
  %typemap(in) BasicImage<float> {
     // Conversion code goes here.
  }
}
```

However the task of conversion must be specified manually. **Convert-XY** is a solution which can be used to automatically build converters for SWIG typemaps.

Differences in Type-checking

Python and C++ use fundamentally different type systems. In *dynamic typing* (Python), types are not prespecified and can change as the program runs. Thus, checks for type errors happen at *run time*. In *static typing* (C++), types never change throughout the execution of a program so type errors are found at compile time but type checks are omitted at run time because type correctness guarantees are made before the program starts. The template facility within C++ is a language in itself because templates are just compile time programs. Templates can be used to express sophisticated code generation and compile time logic. **Convert-XY** exploits the template facility to build conversion functions and dynamic type checkers for converting between C++ and Python objects at compile time.

Convert-XY

Convert-XY's names and functions are declared within the `ConvertXY` namespace. For the rest of this paper, we assume its symbols have been imported into the local namespace with `using namespace ConvertXY;`. We will be explicit when introducing a **Convert-XY** class or function for the first time.

The standard API for converting from a Python object to a C++ object is:

```
// From Python to C++
void ConvertXY::convert(PyObject *x, CPPType &y);
```

Reference counts are updated to ensure a consistent state in cases where the flow of execution returns to Python during conversion.

The `ConvertXY::convert` function is also used to convert from a C++ object back to a Python object; it returns an *unowned* reference to the Python object it creates:

```
// From C++ to Python
PyObject *ConvertXY::convert(CPPType &x);
```

In cases where an object of type `CPPType` does not support a default constructor, the `ConvertXY::Holder` class can be used:

```
void convert(PyObject *x, Holder<CPPType> &y);
```

The `Holder` class defers construction until more information is known about the source `x`, e.g. its dimensions, contiguity, or striding. A specialized `ConvertXY::ObjectFactory<CPPType>` class invokes a non-default constructor of `CPPType` with the information needed for construction. For example, an image size and buffer pointer are both needed to construct a `BasicImage` object. The `Holder` class conveniently enables us to declare a target without immediately constructing its innards. For example,

```
void example(PyObject *x) {
   Holder<BasicImage <float>> y;
   convert(x, y);
   BasicImage <float> &yr(y.getReference());
   display_image(yr);
}
```

The `BasicImage` object in the `Holder` object is not created until `convert` is called. The `getReference()` method returns a `BasicImage` reference created in `convert`. `Holder` 's destructor destroys the `BasicImage` object it contains. The `display_image` function opens up a viewer window to display the image. We assume that the control of execution never returns to Python while the viewer is still open. In a later section, we explain how to handle cases when this cannot be safely assumed.

`ToCPP` and `ToCPPBase` classes

`ConvertXY::ToCPPBase<CPPType, Action>` and `ConvertXY::ToCPP<CPPType, PyType, Action>` are the most fundamental classes in **Convert-XY**. Inheritence enables us to manage conversion between statically-typed C++ objects and dynamically-typed Python objects. The base class `ToCPPBase` assumes a C++ type but makes no assumptions about a Python type or protocol. Its derived class `ToCPP` assumes the Python object follows a protocol or has a specific type. There is a separate derived class for each possible (C++, Python) type pair. This effective design pattern mixes static polymorphism (templates) and dynamic polymorphism (inheritence) to properly handle two different typing paradigms. The `Action` template argument specifies how the conversion takes place; providing it is optional as an action specification is automatically generated for you by default.

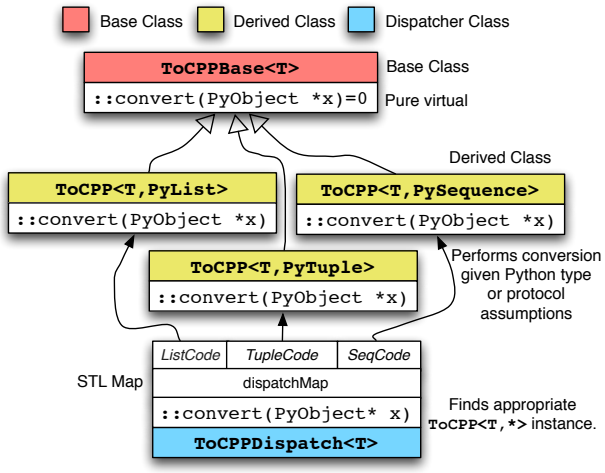

Figure 3 *The fundamental design pattern of* ***Convert-XY****: mixing dynamic polymorphism (inheritence) and static polymorphism (templates). The* `ToCPPDispatch` *class determines which types and protocols are applicable, and finds the appropriate* `ToCPP` *converter instance.*

`ConvertXY::ToCPPDispatch<CPPType, Action>` maintains an associative array between Python object types/ protocols and `ToCPP` derived classes. When the call `convert(x,y)` is compiled, pattern matching is applied on the type of `y` to recursively deduce which base converters could be needed at run-time. The `convert` function calls `ToCPPDispatch<CPPType,Action>::getConverter(x)`, which examines at run time the type and protocol information of the Python object `x` and returns a `ToCPP<CPPType,T,Action>` instance as a `ToCPPBase<CPPType>` reference, call it `z`. Then, the code `z.convert(x,y)` converts `x` to `y` with type assumptions of `x` (encoded with the type `T`) deduced at run-time.

At compile time, a Python object is converted to a C++ object of type `CPPType`, `ToCPP` classes are instantiated recursively for each subpart of `CPPType`, and this is repeated until no more containing types remain. The end result from this recursive instantiation is a compiled chain of converters for interchanging C++ objects of a specific type. Figure 4 illustrates how recursive instantiation works on a `map<string, vector<BasicImage<float>>>`.

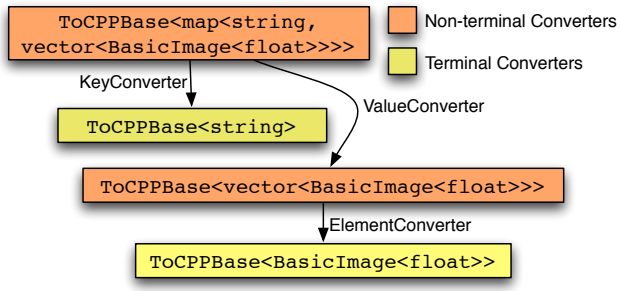

Figure 4 *Illustrates the process of instantiating* `Convert` *classes recursively. In orange are non-terminal converters which instantiate either non-terminal or terminal converters, and in yellow, terminal converters.*

In most cases, the basic user will never need to use to `ObjectFactory`, `ToCPP`, `ToCPPBase`, `ToCPPDispatch`, or `DefaultToCPPAction` classes directly.

Default Conversion Actions

Some C++ array classes, such as LIBCVD's `BasicImage`, do not manage their own memory and can reuse existing buffers such as ones coming from a different language layer (e.g. Python) or special memory from a device (e.g. a digital camera). These are called reference array classes. There are two different ways to convert a NumPy array to a reference array object: copy the buffer pointer or copy the data. `ConvertXY::DefaultToCPPAction` is a templated class for defining how each part of the conversion chain happens; it serves a purpose only at compile time. The `typedef`:

```
DefaultToCPPAction<map<string,
   vector<BasicImage<float>>>>::Action
```

is expanded recursively, mapping the STL `std::map` to the type:

```
Copy<Copy, Copy<Reuse>>
```

This compound type specifies that the string keys and vector elements be copied but the buffer pointer to the arrays should be reused. Figure 5 illustrates how this recursive instantiation works.

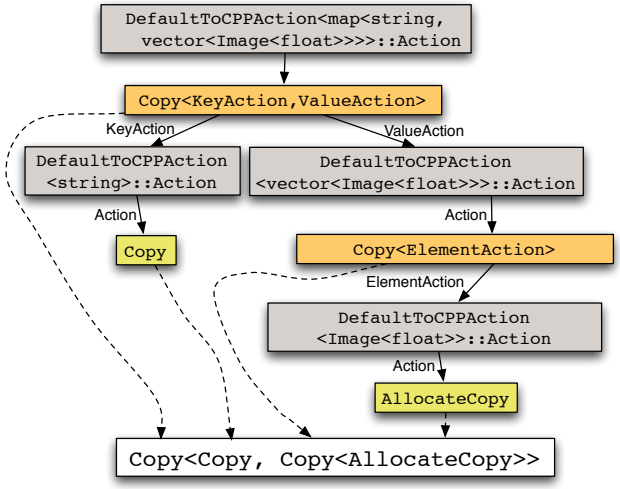

Figure 5 *The process of conversion is deduced at compile time via recursive instantiation of* `DefaultToCPPAction`.

The process of conversion can be overridden by passing as a template argument a different `Action` type. For example, suppose the target `BasicImage` was allocated from elsewhere (e.g. a special `malloc` function that properly aligns buffers), we can override the default conversion action (`Reuse`) so `Copy<>` is used as the `Action` instead. The `ConvertXY::convert_override` function can be used for this purpose:

```
void example(PyObject *x, BasicImage<float> &y) {
   convert_override<Copy<>>(x, y);
}
```

Suppose a C++ object `y` of a compound container type already exists, e.g. an object of type `map<string, BasicImage<float>>`. In this case, the keys should not be copied but we must ensure that the C++ map contains exactly the same set of keys as the Python dictionary. This is done by first checking that the size of the dictionary is the same as the `std::map` and then making sure each unique key in the dictionary is equivalent to some unique key in the map.

The contents of a dictionary of arrays can be copied into a `std::map` of existing `BasicImage` buffers as follows:

```
void example(PyObject *x,
             map<string, BasicImage<float>> &y) {
  convert_override
    <CheckSize<CheckExists, Copy<>>>(x, y);
}
```

The `CheckSize` simple checks that the size of the `std::map` and Python map are equivalent. `CheckExists` ensures each key in the Python map is also in the STL map.

Reference counting is automatically handled during conversion with PyCXX. In cases where the flow of execution may return to Python after conversion, PyCXX can be used to ensure objects that are in use by C++ routines don't get prematurely destroyed.:

```
void example(PyObject *x,
             BasicImage<float> &y) {

  // The line below is safe.
  convert(x,y);

  // However, if it can't be guaranteed that program
  // control flow will never return to Python
  // for the rest of this function, use PyCXX's
  // Py::Object to increment the reference count.
  Py::Object obj(x);


  // If the flow of execution returns to Python while
  // executing the function below, y will always point
  // to a valid buffer.
  compute(y);
}
```

The Other Way Around: from C++ to Python

When converting from C++ to Python, the type structure of the target Python object must be prespecified at compile time (e.g. the type of the Python object to be created is static). This is not straightforward since there is sometimes more than one compatible Python type for a C++ type. For example, an STL vector can be converted to a Python list, tuple, or NumPy object array. In **Convert-XY** this is deduced with a compile-time-only `Structure` metatype deduced by recursive instantiation of `DefaultToPythonStructure<CPPType>::Structure`. Figure 6illustrates how `DefaultToPythonStructure` is recursively instantiated to yield a default Python target type.

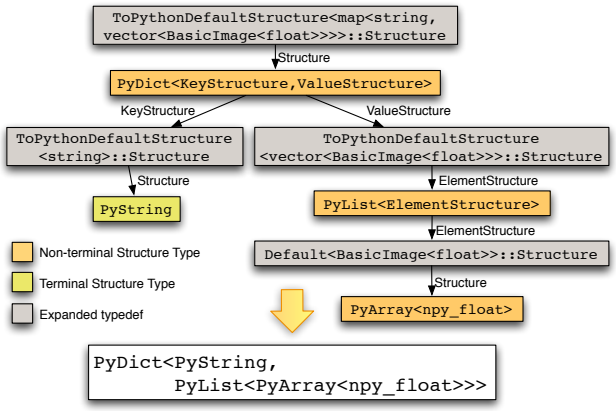

Figure 6 *The default Python type when converting from C++ to Python is deduced by recursive instantiation of* `ToPythonDefaultStructure<CPPType>`.

As the figure illustrates, the default type for a Python target given a source object that's an STL map mapping string to vectors of images is `PyDict<PyString, PyList<PyArray<npy_float>>>`. If a tuple is preferred over a list, the default structure can be overriden as follows:

```
PyObject *y
  = convert_override
      <PyDict<PyString,
              PyTuple<PyArray<npy_float>>>>(x)
```

If an attempt is made to convert a C++ object to a Python object that's incompatible, meaningful compiler messages can be generated via a trick involving incomplete types:

```
in not_found.hpp:88:
  'unsupport_type_or_structure'
  has incomplete type
  CannotConvertToPython<map<int,vector<int>>,PyInt>.
```

Allocating Result Arrays

One approach to allocating result arrays is to copy the result (e.g. image or matrix) into a new NumPy array. This overhead can be avoided using the `ConvertXY::allocate_numpy_array` function. This function is templated, and pattern matching is performed on the C++ type to generate an allocator at compile time. In this example, when the `allocate_numpy_array` function is called, NumPy array and `BasicImage` object are allocated at the same time, sharing the same buffer pointer. The image is smoothed using the LIBCVD function `convolveGaussian`, as illustrated on figure 7:

```
PyObject *smooth_image(PyObject *x, double radius) {

    // First convert the NumPy array input image
    // to a BasicImage<float>.
    Holder<BasicImage <float>> y;
    convert(x, y);

    // Now allocate enough memory to store the
    // result. Use the same buffer pointer for
    // both the NumPy array and BasicImage.
    Holder<BasicImage<float>> result;
    PyObject *pyResult =
      allocate_numpy_array(result,
       y.getReference().size());

    // Using LIBCVD, convolve a Gaussian on the
    // converted input, store the result in the
    // buffer pointed to by both pyResult and
    // result.
    CVD::convolveGaussian(y.getReference(),
                          result.getReference(),
                          radius);

    return pyResult;
}
```

Libraries Supported

Convert-XY supports conversion between Python and objects from several C++ libraries including TooN, LIBCVD, and STL. The library is split into three different headers, all of which are optional.

- `ConvertXY/TooN.hpp`: for converting between NumPy arrays and TooN matrices and vectors.
- `ConvertXY/STL.hpp`: for converting between STL structures and Python objects.

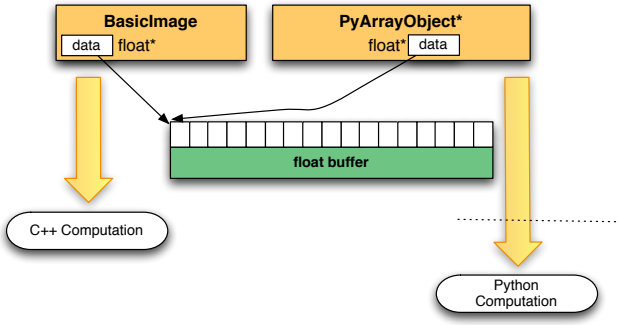

Figure 7 *`allocate_numpy_array` constructs a new Python and C++ array at the same time, to wit, a NumPy array and a `BasicImage` object. The NumPy array owns the buffer allocated so when its reference count goes to zero, its memory buffer will be freed.*

- `ConvertXY/CVD.hpp`: for converting between NumPy arrays and `CVD::Image` C++ objects.

The compiler can convert combinations of STL, CVD, and TooN structures (e.g. `vector<pair<BasicImage<float>,Matrix<>>>`) only when their respective headers are included together.

Tom's object-oriented Numerics Library (TooN)

TooN is a header-only linear algebra library that is very fast on small matrices making it particularly useful for real-time Computer Vision applications [Dru03]. TooN provides templated classes for static and dynamically-sized vectors and matrices, and uses templated-based pattern matching to catch for common linear algebra errors at compile time. For example, when converting the result of an a matrix multiplication involving two matrices `X` and `Y` of incompatible dimensions:

```
Matrix <3,5> X;
Matrix <6,3> Y;
PyObject *Z;
Z = convert(X * Y);
```

an easy-to-understand compiler error results, "`dimension_mismatch has incomplete type SizeMismatch<5,6>`".

Dimension checking is performed at compile time even when some dimensions are not known beforehand as the following example shows:

```
Matrix <Dynamic,5> X;
Matrix <6,Dynamic> Y;
PyObject *Z;
Z = convert(X * Y);
```

If a dimension check passes at compile time, it is omitted at run time. A buffer pointer of a NumPy array can be reused in a `TooN::Matrix` by specifying a layout type as the third template argument to `Matrix` (which is matched with `DefaultAction`):

```
void example(PyObject *x) {
  Holder<Matrix <Dynamic,
                 Dynamic,
                 Reference::RowMajor>> y;
  convert(x, y);
  cout << "The matrix is: "
       << y.getReference() << endl;
}
```

Calling the function above from Python with a very large NumPy array incurs practically no overhead because the NumPy array buffer pointer is copied.

Cambridge Video Dynamics Library (LIBCVD)

LIBCVD is a C++ library popular in the framerate, real-time computer vision community [Ros04]. The three main classes of the library are `BasicImage`, `Image`, and `ImageRef`. The base class `BasicImage` does not manage its own memory and can reuse buffer pointers from other objects such as NumPy arrays. The `Image` class inherits from the `BasicImage`, allocates its own memory, keeps a reference count of the number of `Image` objects referring to it, and deletes its buffer when the reference count drops to zero. `ImageRef` is simply a location in an image, i.e. an `(x,y)` pair. `Image` inherits from `BasicImage` so algorithms can be written to generically operate on both `Image` and `BasicImage` objects. Invoking `size()` on an image returns an `ImageRef`. The `ImageRef` objects are also used to index the image.

LIBCVD contains many of the most common image processing algorithms including convolution, morphology, camera calibration and connected components. It can load and process streaming video in many formats as well as PNG, JPEG, BMP, and PNM image formats.

The example below converts a NumPy array to a BasicImage, finds all local maxima within a 3 by 3 window above a threshold, converts the result to a Python object, and returns it:

```
template <class T>
PyObject* large_local_maxima(PyObject *pyImage,
                             PyObject *pyThresh) {
  Holder<BasicImage<T>> image;
  double thresh;

  convert(pyImage, image);
  convert(pyThresh, thresh);

  BasicImage<T> &I(image.getReference());

  vector<ImageRef> maxima;
  for(int y=1; y < I.size().y-1; y++)
    for(int x=1; x < I.size().x-1; x++) {
      T ctr = I[y][x];
      if(ctr > thresh      && ctr > I[y-1][x-1] &&
         ctr > I[y-1][x]   && ctr > I[y-1][x+1] &&
         ctr > I[y][x-1]   && ctr > I[y][x+1]   &&
         ctr > I[y+1][x-1] && ctr > I[y+1][x] &&
         ctr > I[y+1][x+1] ) {
        maxima.push_back(ImageRef(x, y));
      }
    }
  }
  return convert(maxima);
}
```

The example shows that with LIBCVD's lightweight, simple design, computer vision codes can be written in C++ succinct with good run-time performance. **Convert-XY** in combination with an existing wrapping tool greatly simplifies the task of interfacing Python with a C++ library.

The function is templated so it works on array elements of any numeric type. Templated functions are harder to wrap from Python. One simple, flexible approach involves writing a function that walks across a list of types specified as a template argument. In this case, the list of types is called `List`:

```
template <class List>
PyObject*
   large_local_maxima_walk(PyObject *pyImage,
                           PyObject *pyThresh) {
  typedef List::Head HeadType;
  typedef List::Tail TailList;
  const bool listEnd = List::listEnd;
  if (listEnd) {
     throw Py::Exception("type not supported!");
  }
  try {
     return large_local_maxima<HeadType>(pyImage,
                                         pyThresh);
  }
  catch (ConvertXY::TypeMismatch &e) {
     return large_local_maxima_walk<TailList>(pyImage,
                                              pyThresh);
  }
}
```

Then write a C wrapper function:

```
extern "C"
PyObject* large_local_maxima_c(PyObject *pyImage,
                               PyObject *pyThresh) {
  return large_local_maxima_walk
     <ConvertXY::NumberTypes>(pyImage, pyThresh);
}
```

that calls the walk function on a list of types. This C function can easily be called with SWIG, the Python C API, PyCXX, boost, or Cython. When debugging new templated functions, each error is usually reported by the compiler for each type in the type list. This can be avoided by first debugging with a list containing only a single type:

```
extern "C"
PyObject* large_local_maxima_c(PyObject *pyImage,
                               PyObject *pyThresh) {
  return large_local_maxima_walk
         <ConvertXY::SingletonList<float>>
            (pyImage, pyThresh);
}
```

Improvements and PyCXX Integration

Convert-XY has been refactored considerably since its first introduction at the SciPy conference in August 2009. The most significant improvements include customizable actions and structures, greatly simplified semantics, as well as being fully integrated with the mature C++ package PyCXX. It handles Python referencing and dereferencing in **Convert-XY** is handled to make conversion safe to exceptions and return of execution control to Python.

Packages Using Convert-XY

The authors have written several packages that already use **Convert-XY** .

- `GGFE`: grammar guided feature extraction is a technique for generating image features from generative grammars. GGFE is a tool for expressing generative grammars in pure Python. (see http://ggfe.googlecode.com). Some image features are implemented in C++ using LIBCVD with conversion handled by **Convert-XY**.
- `ACD`: a Python package for anomalous change detection in hyperspectral imagery. All algorithms are written in pure Python but optional, performance-enhanced C++ versions of functions make heavy use of **Convert-XY**. `ACD` is property of the Los Alamos National Laboratory and its release is still being considered by management.
- `ETSE`: a package for performing dynamic time warping and time series clustering in Python. (see http://www.soe.ucsc.edu/~eads/software.shtml)
- `Tesla`: a new object localisation system under development as part of our research. The software will be released upon publication of our algorithm.

Conclusion

Convert-XY is a powerful tool that facilitates automatic conversion of arbitrarily structured containers between C++ and Python with a succinct syntax, `convert(x,y)`. By exploiting template-based pattern matching in C++, dynamic type checkers and converters can be recursively built based on the static structure of a C++ object. At run-time, the dispatcher class decodes the type of the Python object and deduces the protocols it obeys. Additionally, conversion is customizable by specifying different action or structure meta-types. Large data sets can be converted between Python and C++ with minimal copying. When possible, erroneous conversions are caught at compile time but otherwise caught at run time. **Convert-XY** integrates with PyCXX to improve exception safety during conversion. It can also be used to automatically facilitate conversion for SWIG typemaps.

License

Convert-XY is offered under the terms of the General Public License version 2.0 with a special exception.

> As a special exception, you may use these files as part of a free software library without restriction. Specifically, if other files instantiate templates or use macros or inline functions from this library, or you compile this library and link it with other files to produce an executable, this library does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

Future Work

The primary focus of **Convert-XY**'s development until now has been on greatly improving the safety, simplicity, and flexibility of the interface. Moving forward, we plan to focus efforts on improving the documentation, finishing a regression test suite, and writing a tutorial on how to write custom converters for other libraries.

References

[Abr03] D. Abrahams. *Building Hybrid Systems with Boost Python*. PyCon 2003. 2003.

[Bea95] D. Beazley. *Simplified Wrapper and Interface Generator*. http://www.swig.org/. 1995--.

[Dru03] T. Drummond, E. Rosten, et al. *TooN: Tom's Object-oriented Numerics*. http://mi.eng.cam.ac.uk/~twd20/TooNhtml/. 2003.

[Ewi08] M. Ewing. *Cython*. 2008--.

[Hel00] T. Heller. *ctypes*. 2000--.

[Jon01] E. Jones, T. Oliphant, P. Peterson, et al. "SciPy: Open Source Scientific tools for Python". 2001--.

[GvR92] G. van Rossum. *Python*. 1991--.

[Ros04] E. Rosten, et al. *LIBCVD*. 2004--

[Sco04] B. Scott, P. Dubois. *Writing Python Extensions in C++ with PyCXX*. http://cxx.sf.net/. 2004--.

[Yak09] R. Yakovenko. Py++: Language Binding Project. 2009.