LA-UR- 08-5862

Approved for public release; distribution is unlimited.

Title: Experiences Using Scipy for Computer Vision Research

Author(s): Damian Eads
Edward Rosten

Intended for: Proceedings of the 2008 Scipy Conference
California Institute of Technology
Pasadena, CA

Los Alamos
NATIONAL LABORATORY
EST. 1943

Experiences Using SciPy for Computer Vision Research

Damian Eads (eads@lanl.gov) – *Los Alamos National Lab, MS D436* USA
Edward Rosten (edrosten@lanl.gov) – *Los Alamos National Lab, MS D436* USA

SciPy is an effective tool suite for prototyping new algorithms. We share some of our experiences using it for the first time to support our research in object detection. SciPy makes it easy to integrate C code, which is essential when algorithms operating on large data sets cannot be vectorized. The universality of Python, the language in which SciPy was written, gives the researcher access to a broader set of non-numerical libraries to support GUI development, interface with databases, manipulate graph structures, render 3D graphics, unpack binary files, etc. Python's extensive support for operator overloading makes SciPy's syntax as succinct as its competitors, MATLAB, Octave, and R. More profoundly, we found it easy to rework research code written with SciPy into a production application, deployable on numerous platforms.

Introduction

Computer Vision research often involves a great deal of effort spent prototyping new algorithms code. A highly agile, unstructured, iterative approach to code development takes place. Development in low-level languages may be ideal in terms of computational efficiency but is often time consuming and bug prone. MATLAB's succinct "vectorized" syntax and efficient numerical, linear algebra, signal processing, and image processing codes has led to its popularity in the Computer Vision community for prototyping algorithms. Last year, we started a completely new research project in object detection using the SciPy+Python [Jon01] [GvR92] framework without any extensive experience developing with it but having substantial knowhow with MATLAB and C++. The software had to run on Windows and be packaged with an installer. The research problem was very open-ended so a large number of prototype algorithms needed development but eventually the most promising among them had to be integrated into a production application. The project sponsor imposed short deadlines so the decision to use a new framework was high risk as we had to learn the new tool set while keeping the research on pace. Postmortem, we found SciPy to be an excellent choice for both prototyping new code and migrating prototypes into a production system. Acquiring proficiency with SciPy was quick: completing useful, complicated tasks was achievable within a few hours of first installing the software. In this paper, we share some noteworthy reflections on our first experience with SciPy in a full-scale research project.

A Universal Language

One of the strengths of SciPy is that it is a library for Python, a universal and pervasive language. This has two main benefits. First, there is a separation of concerns: the language (Python) is developed independently of the SciPy tool set. The Python community focuses strictly on maintaining the language and its interpreter while the SciPy community focuses on the development of scientific tool sets. The efforts of both groups are not spread thinly across both tasks freeing more time to focus on good design, maintenance, reliability, and support. MATLAB [Mwc82], R [Rcd04], and Octave [Eat02] must instead accomplish several tasks at once: designing a language, implementing and maintaining an interpreter, and developing numerical codes. Second, the universality of Python means there is a much broader spectrum of self-contained communities beyond scientific computation, each of which solely focuses on a single kind of library (e.g. GUI, database, network I/O, cluster computation). Third-party library communities are not as common for highly specialized numerical languages so additional effort must be spent developing tertiary capabilities such as GUI development, database libraries, image I/O, etc. This further worsens the "thin spread" problem: there are fewer time and resources to focus on the two core tasks: developing the language and developing numerical and scientific libraries.

SciPy does not suffer from the "thin spread" problem because of the breadth of libraries available from the many self-contained Python communities. As long as a library is written in Python, it can be integrated into a SciPy application. This was very beneficial to our research in computer vision because we needed capabilities such as image I/O, GUI development, etc. This enabled a more seamless migration into production system.

Operator Overloading: Succinct Syntax

Python's extensive support for operator overloading is a big factor in the success of the SciPy tool set. The array bracket and slice operators give NumPy great flexibility and succinctness in the slicing of arrays (e.g. `B=A[::-1,::-1].T` flips a rectangular array in both directions then transposes the result.)

Slicing an array on the left-hand side of an assignment performs assignments in-place, which is particularly useful in computer vision where data sets are large and unnecessary copying can be costly or fatal. If an array consumes half of the available memory on a machine, an accidental copy will likely result in an `OutOfMemory`

error. This is particularly unacceptable when an algorithm takes several weeks to run and large portions of state cannot be easily check-pointed.

In NumPy, array objects either own their data or are simply a view of another array's data. Both array slicing and transposition generate array views so they do not involve copying. Instead, a new view is created as an array object with its data pointing to the data of the original array but with its striding parameters recalculated.

Extensions

Prior to the project's start, we wrote a large corpora of computer vision code in C++, packaged as the Cambridge Video Dynamics Library (LIBCVD) [Ros04]. Since many algorithms being researched depended on these low-level codes, a thorough review of different alternatives for C and C++ extensions in Python was needed. Interestingly, we eventually settled on the native Python C extensions interface after trying several other packages intended to enhance or replace it.

A brief description is given of the `Image` and `ImageRef` classes, the most pervasive data structures within LIBCVD. An `Image<T>` object allocates its own raster buffer and manages its deallocation while its subclass `BasicImage<T>` is constructed from a buffer and is not responsible for the buffer's deallocation. The `ImageRef` class represents a coordinate in an image, used for indexing pixels in an `Image` object.

ctypes

`ctypes` [Hel00] seems the easiest and quickest to get started but has a major drawback in that `distutils` does not support compilation of shared libraries on Windows and Mac OS X. We also found it cumbersome to translate templated C++ data structures into NumPy arrays. The data structure would first need to be converted into a C-style array, passed back to Python space, and then converted to a NumPy array. For example, a set of (x, y) coordinates would be represented using `std::vector<ImageRef>` where the coordinates are defined as `struct ImageRef {int x, y;};`. The function for converting a vector of these `ImageRef` structs into a C array is:

```
int *convertToC(vector <ImageRef> &xy_pairs,
                int *num) {
  int *retval = new int[xy_pairs.size()*2];
  *num = xy_pairs.size();
  for (int i = 0; i < xy_pairs.size(); i++) {
    retval[i*2] = xy_pairs[i].x;
    retval[i*2+1] = xy_pairs[i].y;
  }
  return retval;
}
```

Not knowing in advance the size of output buffers is a common problem in scientific computation. In the example, the number of (x, y) pairs is not known *a priori* so the NumPy array cannot be allocated prior to calling the C++ function generating the pairs. One could use the NumPy array allocation function in C++-space but this defeats one of the main advantages of `ctypes`: to interface Python-unaware code.

Once the C-style array is returned back to Python-space, the next natural step is to use the pointer as the data buffer of a new NumPy array object. Unfortunately, this is not easy as it seems because three problems stand in the way. First, there is no function to convert the pointer to a Python buffer object, which is required by the `frombuffer` constructor; second, the `frombuffer` constructor creates arrays that do not own their data; third, even if an array can be created that owns its own data, there is no way to tell NumPy how to deallocate the buffer. In this example, the C++ operator `delete` is required instead of `free()`. In other cases, a C++ destructor would need to be called.

We eventually worked around these issues by creating a Python extension with three functions: one that converts a C-types pointer to a Python buffer object, one that constructs an nd-array that owns its own data from a Python buffer object, and a hook that deallocates memory using C++ `delete`. Even with these functions, each C++ function needs to be wrapped with a C-style equivalent:

```
int* wrap_find_objects(const float *image,
                       int m, int n, int *size) {
  BasicImage <float> cpp(image, ImageRef(m, n));
  vector <ImageRef> cpp_refs;
  find_objects(cpp, cpp_refs);
  *size = cpp_refs.size();
  return convertToC(cpp_refs);
}
```

This function takes in an input image and size, which it converts to a `BasicImage` and calls the `find_objects` routine, which is used to find the (x, y) pairs corresponding to the locations of objects in an image, which it returns as a C-style array. Since `ctypes` does not implement C++ name mangling, there is no function signature embedded in the shared library. Thus, type checking is not performed in Python so a core dump may result when not invoked properly. To avoid these bugs, we needed to create a Python wrapper to do basic type checking of arguments and conversion of input and post-processing of output. `ctypes` is intended to eliminate the need for wrappers, yet two were needed for each C++ function being wrapped. We found `ctypes` inappropriate for our purposes: wrapping large amounts of C++ code safely and efficiently. We did, however, find `ctypes` appropriate for wrapping:

- numerical C codes where the size of output buffers is known ahead of time and can be done in Python-space to avoid ownership and object lifetime issues.
- wrapping non-numerical C codes, particularly those with simple interfaces that use basic C data structures (e.g. encrypting a string, opening a file, or writing a buffer to an image file.)

Weave

With SciPy `weave`, C++ code can be embedded directly in a Python program as a multi-line string in a Python program. MD5 hashes are used to cache compilations of C++ program strings. Whenever the type of a variable is changed, a different program string results, causing a separate compilation. `weave` properly handles iteration over strided arrays. Compilation errors can be cryptic and the translation of a multi-line program string prior to compilation is somewhat opaque. Applications using `weave` need a C++ compiler so it did not fit the requirements of our sponsor. However, we found it useful for quickly prototyping "high risk" for-loop algorithms that could not be vectorized.

Boost Python

Boost Python is a large and powerful library for interfacing C++ code from Python. Learning the tool set is difficult so a large investment of time must be made up front before useful tasks can be accomplished. Boost copies objects created in C++-space, rather than storing pointers to them. This reduces the potential of a dangling reference to an object from Python space, a potentially dangerous situation. Since our computer vision codes often involve large data sets, excessive copying can be a show-stopper.

SWIG

SWIG is a tool for generating C and C++ wrapper code. Our time budget for investigating different alternatives for writing extensions was limited. Several colleagues suggested using the SWIG library to perform the translation and type checking. The documentation of more complicated features is somewhat lacking. The examples are either the "hello world" variety or expert-level without much in between. When deadlines neared, we decided to table consideration of SWIG. However, we encourage those in the SciPy community who have had success with SWIG to document their experiences so others may benefit.

Cython

Cython is a Python dialect for writing C extensions. Its development has been gaining momentum over the past six months. Python-like code is translated into C code and compiled. It provides support for static type checking as well as facilities for handling object lifetime. Unfortunately, its support for interfacing with templated C++ code is limited. Given the large number of templated C++ functions needing interfacing, it was unsuitable for our purposes.

Native Python C Extensions

As stated earlier, we eventually settled native Python C extensions as our extension framework of choice. A small suite of C++-templated helper functions made the C wrapper functions quite succinct, and performed static type checking to reduce the possibility of introducing bugs.

We found that all the necessary type checking and conversion could be done succinctly in a single C wrapper function and that in most cases, no additional Python wrapper was needed. A few helper functions were written to accommodate the conversion and type checking:

- `BasicImage<T> np2image<T>(img)` converts a rectangular NumPy array with values of type `T` to a `BasicImage` object. If the array does not contain values compatible with `T`, an exception is thrown.
- `PyArrayObject *image2np<T>(img)` converts an `Image` object to a NumPy array of type `T`.
- `PyArrayObject *vec_imageref2np(v)` converts an `std::vector<ImageRef>` of N image references to a N by 2 NumPy array.
- `pair <size_t, T*> np2c(v)` converts a a rectangular NumPy array to a `std::pair` object with the size stored in the first member and the buffer pointer in the second.

Shown below is a boilerplate of a wrapper function. C++ calls go in the `try` block and all errors are caught in the `catch`. All of the helper functions throw an exception if an error occurs during conversion, allocation, or type check. By wrapping the C++ code in a `try/catch`, any C++ exceptions thrown as a `std::string` are immediately caught in the wrapper function. The wrapper function then sets the Python exception string and returns. This solution freed us from having to use Python error handling and NumPy type checking constructs in our core C++ code:

```
PyObject* wrapper(PyObject* self,
                  PyObject* args) {
  try {
    if(!PyArg_ParseTuple(...)) return 0;
    // C++ code goes here.
  }
  catch(string err) {
     PyErr_SetString(PyExc_RuntimeError, err.c_str());
      return 0;
  }
}
```

Shown below is an example of the `np2image` helper function, which converts a `PyArrayObject` to a `BasicImage`. If any errors occur during the type check, a C++ exception is thrown, which gets translated into a Python exception:

```
template<class I>
BasicImage<I> np2image(PyObject *p,
                       const std::string &n="") {

  if (!PyArray_Check(p)
      || PyArray_NDIM(p) != 2
      || !PyArray_ISCONTIGUOUS(p)
      || PyArray_TYPE(p) != NumpyType<I>::num) {
    throw std::string(n + " must be "
      + "a contig array of " + NumpyType<I>::name()
      + "(typecode " + NumpyType<I>::code() + ")!");
  }
  PyArrayObject* image = (PyArrayObject*)p;
  int sm = image->dimensions[1];
  int sn = image->dimensions[0];
  CVD::BasicImage <I> img((I*)image->data,
                          CVD::ImageRef(sm, sn));
  return img;
}
```

Parsing arguments passed to a wrapper function is remarkably simple given a single, highly flexible native extensions function `PyArg_ParseTuple`. It provides basic type checking of Python arguments, such as verifying an object is of type `PyArrayObject`. More thorough type checking of the underlying type of data values in a NumPy array is handled by our C++ helper functions.

Many of the functions in C++ are templated to work across many pixel data types. We needed a solution for passing a NumPy array to an appropriate instance of a templated function without needing a complicated `switch` or `if` statement for each templated function being wrapped. A special wrapper function must be written that generically calls instances of the templated function. We call this a "selector", which is encapsulated in a templated struct:

```
template<class List>
struct convolution_ {
  static PyObject* fun(PyArrayObject *image,
                       double sigma) {
    // Selector code goes here.
  }
}
```

An example of a selector for a convolution function is shown below. It works generically across multiple pixel data types. An instance of the struct is generated for each type in a type list. We iterate through the type list via a form of template-based pattern matching, checking the type of the array with the type of the type in the list's head. If it matches, we call `convolveGaussian`:

```
typedef typename List::type type;
typedef typename List::next next;
if (PyArray_TYPE(image) == NumpyType<type>::num) {
  BasicImage <type> input
    (np2image<type>(image, "image"));
  PyArrayObject *py_result;
  BasicImage <type> result
    (alloc_image<type>(input.size(), &py_result));
  convolveGaussian(input, result, sigma);
  return (PyObject*)py_result;
}
```

Otherwise, we invoke the tail selector:

```
else {
  return _convolution<next>::fun(image, sigma);
}
```

If the type is not supported because none of the types matched, an exception must be thrown:

```
template<>
struct convolution_ <PyCVD::End> {
  static PyObject* fun(PyArrayObject *image,
                       double sigma) {
    throw string("Can't convolve with type: "
                 + PyArray_TYPE(image));
  }
};
```

Finally, the native C wrapper function calls the convolution selector. The selector is templated with a type list of supported types:

```
extern "C" PyObject *convolution(PyObject *self,
                                 PyObject *args) {
  try {
    PyArrayObject *_image;
    double sigma;
    if (!PyArg_ParseTuple(args, "O!d",
            &PyArray_Type, &_image,
            &sigma)) { return 0; }
    return convolution_<CVDTypes>::fun(_image, sigma);
  }
  catch(string err) {
    PyErr_SetString(PyExc_RuntimeError, err.c_str());
    return 0;
  }
}
```

The `TypeList` construct is now defined. All type lists terminate with a special `End` (sentinel) struct:

```
struct End{};
template<class C, class D> struct TypeList {
  typedef C type;  typedef D next;
};
```

Type lists can now be defined to specify supported data types for a selector. The first type list shown is for most numeric data types while the second one is for floating point types:

```
typedef TypeList<char,
                 TypeList<unsigned char,
                 TypeList<short,
                 TypeList<unsigned short,
                 TypeList<int,
                 TypeList<long long,
                 TypeList<unsigned int,
                 TypeList<float,
                 TypeList<double, End>
                 > > > > > > > > CVDTypes;
typedef TypeList<float,
                 TypeList<double, End>
                 > CVDFloatTypes;
```

Lastly, in order to support translation between native C data types and NumPy type codes, we need a macro for defining helper structs to perform the translation:

```
#define DEFNPTYPE(Type, PyType) \
template<> struct NumpyType<Type> {\
  static const int num = PyType; \
  static std::string name(){ return #Type;} \
  static char code(){ return PyType##LTR;} \
}
template<class C> struct NumpyType {};
```

Next, we instantiate a helper struct for each C data type, specifying its corresponding NumPy type code:

```
DEFNPTYPE(unsigned char , NPY_UBYTE );
DEFNPTYPE(char          , NPY_BYTE );
DEFNPTYPE(short         , NPY_SHORT );
DEFNPTYPE(unsigned short, NPY_USHORT);
DEFNPTYPE(int           , NPY_INT   );
DEFNPTYPE(long long     , NPY_LONGLONG);
DEFNPTYPE(unsigned int  , NPY_UINT  );
DEFNPTYPE(float         , NPY_FLOAT );
DEFNPTYPE(double        , NPY_DOUBLE);
```

Comparison with mex

Prior to this project, the authors had some experience working with MATLAB's External Interface (i.e. `mex`). `mex` requires a separate source file for each function. No function exists with the flexibility as Python's `PyArg_ParseTuple`, making it difficult to parse input arguments. Nor does a function exist like `PyBuildValue` to succinctly return data. Opening `mex` files in `gdb` is somewhat cumbersome, making it difficult to pin down segmentation faults. When a framework lacks succinctness and expressibility, developers are tempted to copy code, which often introduces bugs.

Object-oriented Programming

Many algorithms require the use of data structures than other just rectangular arrays, e.g., graphs, sets, maps, and trees. MATLAB's usually encodes such data structures with a matrix (e.g. the `treeplot` and `etree` functions). MATLAB supports object-oriented programming so in theory one can implement a tree or graph class. The authors have attempted to make use of this facility but found it limited: objects are immutable so changes to them involve a copy. Since computer vision projects typically involve large data sets, if not careful, subtle copying may swamp the system. Moreover, it is difficult to organize a suite of classes because each class must reside in its own directory (named `@classname`) and each method in its own file. Changing the name of a method requires renaming a file and the method name in the file followed by a traversal of all files in the directory to ensure all remaining references are appropriately renamed. Combining three methods into one involves moving the code in the two files into the remaining file and then deleting the originating files. This makes it cumbersome to do agile object-oriented development. To get around these shortcomings, most programmers introduce global variables, but this inevitably leads to bugs and makes code hard to maintain. After many years of trial and error, we found Python+SciPy to be more capable for developing both prototype code and production scientific software.

Python has good facilities for organizing a software library with its *modules* and *packages*. A module is a collection of related classes, functions, and data. All of its members conveniently reside in the same source file. Objects in Python are mutable and all methods of a class are defined in the same source file. Since Python was designed for object-orientation, many subcommunities have created OO libraries to support almost any software engineering task: databases, GUI development, network I/O, or file unpacking. This makes it easy to develop production code.

Data structures such as maps, sets, and lists are built into Python. Python also supports a limited version of a continuation known as a *generator function*, permitting lazy evaluation. Rich data structures such as graphs can easily be integrated into our algorithms by defining a new class. Workarounds such as global variables were not needed. Development with Python's object-oriented interface was remarkably seamless.

In MATLAB, variables are passed by value with copy-on-write semantics. Python's support for pass-by-reference gives one more flexibility by allowing one to pass large arrays to functions and modify them. While these semantics are not as easy to understand as pass-by-value, they are essential for developing production applications as well as for computing on large data sets.

Conclusion

We started a new research project using SciPy without having any previous experience with it. SciPy's succinct, vectorized syntax and its extensive support for slicing makes it a good prototyping framework. The universality of Python gives one access to a wide variety of libraries, e.g. GUI toolkits, database tools, etc., to support production development. Its modules and object-orientation allows for a clean organization of software components. Pass-by-reference semantics permit efficient and safe handling of large data sets. With the flexibility of Python's C extension interface, one can interface with a large corpora of existing C++ code. Our design permits core C++ algorithms to be Python-unaware but with support for error reporting back to the Python environment. Using C++ generics combined with a small suite of macros and helper functions, instances of templated algorithms can be called in a manner that is generic to pixel data type. Overall, we found the Python+SciPy to be an excellent choice to support our research.

References

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

[Eat02] J. Eaton. *GNU Octave Manual.* Network Theory Limited Press. 2002.

[Hel00] T. Heller. *ctypes.* `http://sf.net/projects/ctypes`. 2000--.

[Hun02] J. Hunter. *matplotlib: plotting for Python.* `http://matplotlib.sf.net/`. 2002--.

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

[Mwc82] The Mathworks Corporation. *MATLAB.* `http://www.mathworks.com`. 1984--.

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

[Rcd04] The R Core Development Team. *R Reference Manual.* 2004--.

[Ros04] E. Rosten, et al. *LIBCVD.* http://svr-www.eng.cam.ac.uk/~twd20/. 2004--