LA-UR- 08-5491

Approved for public release; distribution is unlimited.

Title:

First Time 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

First Time Experiences Using SciPy for Computer Vision Research

Damian Eads (eads@lanl.gov) – *Los Alamos National Lab, MS B244, Los Alamos, NM* USA
Edward Rosten (edrosten@lanl.gov) – *Los Alamos National Lab, MS D436, Los Alamos, NM* 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. Python's extensive support for operator overloading makes SciPy's syntax as succinct as its competitors, MATLAB, Octave, and R. 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. 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. Complicated tasks often demand an iterative approach to code development. Developing such codes 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. 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 project sponsor imposed short deadlines so the decision to use a new framework was somewhat "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 reliability, maintenance, and design.

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 with highly specialized numerical languages like MATLAB and Octave so additional effort must be spent developing GUI capabilities, etc. This further worsens the "thin spread" problem: there is less time and resources to focus on 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 it is written in Python, it can be integrated into a SciPy application. Computer Vision research often requires image I/O, GUIs for basic tasks, etc. We were pleasantly surprised by the wealth of Python libraries available to us, and 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 gives 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. Array objects can either own their data or be a view of another array's data, so slicing and transposition does not require 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 so the view can be used properly in further computation.

Extensions

Prior to the project's start, the authors had written 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 extensions in Python was needed. Interestingly, we eventually settled on the original Python C interface (call it PythonExt) after trying several other packages intended to enhance or replace it.

Note that the primary data structures in LIBCVD are the Image and ImageRef classes. An Image<T> object allocates its own raster buffer and manages its deletion 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, which can be used to index a pixel 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 somewhat 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.

By way of 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;
}
```

Since the number of (x, y) pairs is not known *a priori* 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 be independent of PythonExt.

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; 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, the function signature is not embedded in the shared library so a core dump may result when not invoked properly since the call is not safe-guarded with type checking. To avoid these bugs, we needed to create a Python wrapper to do basic type checking of arguments, conversion of input, and conversion of output. ctypes is intended to eliminate the need for wrappers. However, for each C++ function, one wrapper function and conversion code was needed in C to wrap C++ code, and one wrapper in Python to wrap C code with type checking. Thus, we found ctypes inappropriate for our purposes: wrapping large amounts of C++ code safely and efficiently. In summary, our experience shows that ctypes is 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

The SciPy weave package allows embedding C++ code 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 new program string is generated, causing a separate compilation. weave properly handles iteration over strided arrays. Compilation errors can be somewhat cryptic and it is not obvious how a multiline program string is translated prior to compilation. 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 we could not vectorize prior to investing in a `PythonExt` implementation.

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, to avoid 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 could be quite costly.

Python C Extensions (`PythonExt`)

As stated earlier, we eventually settled on `PythonExt` as our C 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 accomodate the conversion and type checking:

- `BasicImage<T> to_image<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*> nparray2c(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.

All of these 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 translated into a Python exception. Thus, all C++ code could be free of Python exception constructs, as the example illustrates:

```
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 one of our helper functions, which converts a `PyArrayObject` to a `BasicImage`. If any errors occur during the type check, an exception is thrown, which gets translated into a Python exception. Significant effort was invested in `to_image` because it is used so much. However, the version below has been shortened for brevity (e.g. we omitted how the name of the offending variable gets included in the exception message.):

```
BasicImage<float> to_image(void* p){
  if(!PyArray_Check(p) || PyArray_NDIM(p) != 2
     || !PyArray_ISCONTIGUOUS(p)
     || PyArray_TYPE(p) != NPY_FLOAT)
     throw "Bad PyArray";
  PyArrayObject* image = (PyArrayObject*)p;
  int sm = image->dimensions[1];
  int sn = image->dimensions[0];
  BasicImage <I> img((I*)image->data,
                     ImageRef(sm, sn));
  return img;
}
```

Parsing arguments passed to a wrapper function is remarkably simple given a single, highly flexible `PythonExt` 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 the C++ helper functions.

SWIG

Ed do you want to write something here? I don't really have experience in this regard.

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 succintness 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 support for these data structures is rudimentary as most of them are encoded with a matrix (e.g. the `treeplot` and `etree` functions accept input as a matrix). 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 severly limited: objects are immutible 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 in MATLAB 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 in MATLAB. To get around these shortcomings, inevitably most programmers introduce global variables but this lends itself to introducing bugs and making code hard to maintain. After many years of trial and error, overall it is very difficult to develop highly capable production scientific software codes in MATLAB.
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. Unlike MATLAB, 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 could be easily coded as a class and integrated into our algorithms. No work arounds, such as global variables were needed. Development with Python's object-oriented interface, aside from the `self` annoyance of which any seasoned Python programmer is aware, was remarkably seemless.

Caveat

Python is a dynamically typed language, and as such, does not offer many static guarantees offered by other languages such as C++, Java, and Ocaml. Prior to programming with Python, the authors had extensive experience programming in a multitude of statically typed languages including some study in typing theory. It is generally good advice to the new programmer to learn as many languages as possible, particularly those with strong typing as they help develop good programming practice, which will serve well when programming in Python.

Plotting

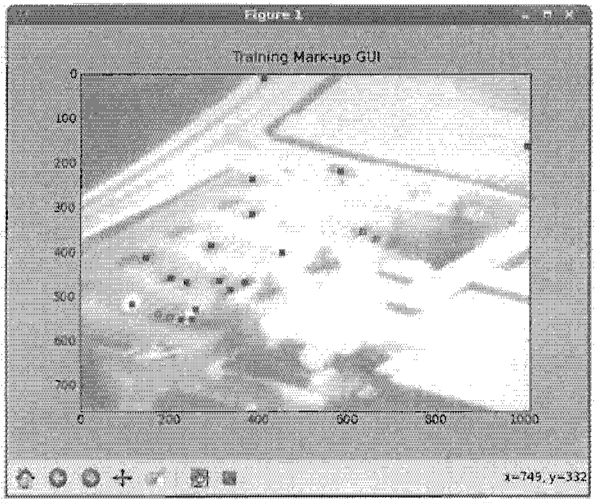
`matplotlib` [Hun02] is the premiere plotting package for the SciPy tool family provides an advanced plotting facility with a syntax similar to MATLAB's. Thus, many users already familiar with MATLAB plotting should find `matplotlib` easy to learn.

The `imshow` command provided by `matplotlib` is similar to both the `scaleim` and `imshow` commands.

Computer Vision research often involves collecting higher-level meta-data from an image. Time-consuming approaches requiring significant overhead in terms of code are not worth it. `matplotlib` provides some very simple call back mechanisms for developing very simple annotation tools. Consider the problem of marking (x, y) locations in an image with a mouse. The following class acheives this in `matplotlib` with minimal code.:

```
class MarkImage:

 def __init__(self, img):
   mplp.title("Mark-up GUI")
   mplp.imshow(img)
   mplp.connect('button_press_event',
               lambda x: self.click_add(x))
   mplp.show()
   self.L = []

 def plot_point(self, x, y):
   mplp.plot([x],[y])

 def click_add(self, event):
   tb = mplp.get_current_fig_manager().toolbar
   if event.button==1 and
      event.inaxes and
      tb.mode == '':
     x,y = event.xdata,event.ydata
     self.L.append([x, y])
     self.plot_point(x, y)
     mplp.draw()

 def get_points(self):
   return np.asarray(L)
```

A screenshot of the annotated GUI is shown in the following figure.



References

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

[Hel00] T. Heller. *ctypes: An Advanced Foreign Functions Interface for Python.* http://python.net/crew/theller/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, T. Drummond, et al. *Cambridge Video Dynamics Library (LIBCVD).* http://svr-www.eng.cam.ac.uk/~twd20/. 2004--