First Time Experiences
Using SciPy
for Computer Vision Research

Damian Eads and Edward Rosten
ISR Division
Los Alamos National Laboratory
Los Alamos, New Mexico

{eads, edrosten}@lanl.gov
Research Problem

Find the cars
Algorithm Workflow

Learning Algorithm

Object Detector
New Research Project

- New government research project in 2007
- Learn object detectors from example data
- Explore new algorithms
- Requirements: short deadlines, must work on Windows and Linux, algorithms exploitation, and production system.
- Extensive knowhow with MATLAB and C++
- No experience with SciPy
- Chose SciPy: risk
Postmortem

- SciPy: a superior choice
- nice learning curve: useful in a few hours
- effective for research and production codes
- universal language (Python)
- easy to rework prototypes into deployable applications
SciPy: good for prototyping

- Easy to vectorize
- Succinct syntax (thanks to Python's extensive support for operator overloading)
- Slicing with views: avoids copying!
- Unlike MATLAB, R and Octave: Python is a universal language
  - Separation of concerns:
    - Python group: the language
    - SciPy group: scientific codes
  - Larger corpora of libraries, more subcommunities: GUI, database, file unpacking, etc.
What this talk is about...

_topic 1. Extensions
- Have large data sets
- Can't always vectorize

_topic 2. C++
- Lots of anti-C++ people
- Static efficiency
- How to interface?
Why we need C++?

✎ A lot of Computer Vision code can't be vectorized
  ● Python “for” loops: cost prohibitive for very large data sets.

✎ C++:
  ● “for” loops are efficient
  ● lots of serial algorithms and data structures, e.g. sets, queues, heaps, multimaps, etc.
  ● static efficiency
  ● you can do more in-place
Computer Vision Codes

- Large data sets and significant computation
  - Efficiency is important
  - Avoid unnecessary duplication
    - Can slow things down,
    - Or hose you!
- Used LIBCVD: a C++ library
  - Cambridge Video Dynamics Library
  - Frame-rate real time implementations of many computer vision algorithms
  - Essential for our work
  - Need to interface C++ library with Python
Basic LIBCVD Data Structures

- **BasicImage<T>:** an image object that does not manage its memory
- **Image<T>:** a new image object whose memory is allocated when created
- **SubImage<T>:** region of an image
- **ImageRef:** coordinates in an image; has two members: x and y.
What we want?

- Call LIBCVD function, pass a numpy array and get back a numpy array.
- Hide the LIBCVD infrastructure!
C++ and Python

- Semantic differences can be painful
  - Both want to manage their own memory
  - Example: when resizing an array, there is no way to tell Python to look at a different buffer
  - Fortunately, LIBCVD has numpy-like semantics
  - Can't always preallocate: size of the buffer might not be known \textit{a priori}
- Hard to examine C++ data structures from Python, e.g. std::vector

UNCLASSIFIED // LA-UR-08-05860 - Eads, Rosten
Call functions by name from shared libraries

*Distutils won't compile* shared libraries properly on windows and Mac OS X

Does not understand C++ name mangling or template instantiation

- Hard to translate C++ data structures into Python ones
ctypes

- C wrapper function. Can call it like a Python function with C-types.

```c
extern C 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);
}
```

UNCLASSIFIED // LA-UR-08-05860 - Eads, Rosten
Converts a C++ vector of (x,y) points to a C-array so it can be understood by c-types

```cpp
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;
}
```
**ctypes**

- Type checking cannot always be done
  - can cause core dumps.
  - Python wrapper may be needed
- **three wrappers per C++ function!**
- **more wrappers to write, more bugs**
- **ctypes inappropriate for our purposes!**
ctypes

* Appropriate for wrapping
  - numerical C codes where buffer sizes are known \textit{a priori}
  - non-numerical C codes with simple interfaces
* Not appropriate for C++.
weave

can write C++ and C programs in Python as multi-line strings!
hashes C++ program strings to map to compiled code
properly handles iteration over strided arrays
pseudo-templated: changing types of input variables causes recompile
weave

Pros
● Great for prototyping “high risk” code
● Seems to work on both platforms

Cons
● Compiler errors can be somewhat cryptic.
● Code translation: somewhat opaque
● Released binary requires compiler
Boost::Python

- Large, powerful, and mature library for interfacing C++ code with Python.
- Steep learning curve: Large investment of time up-front
- Protection can be annoying
  - C++ objects are copied prior to being returned to Python space: avoid problems
  - Hard to avoid copying
  - Excessive copying: either quite costly or a show stopper!
Eventually settled on PythonExt
Conversion from Numpy to CVD and vice versa is easy: helper functions
Error handling is easy!
- Aggressive type checking with templated helpers
- Throw exception
Only a single wrapper function needed.
- Wrapper in Python space was unnecessary
Easy to parse complicated argument tuples!
Great framework!
PythonExt

- Wrote suite of C++-templated helper functions
  - Numpy to C++/CVD
    - `BasicImage <T> to_cvd<T>(PyObject *np)`
    - `void np_to_irvec<T>(PyArrayObject *obj, vector<ImageRef> &out)`
  - C++/CVD to Numpy
    - `PyArrayObject *from_cvd<T>(BasicImage <T> &img)`
    - `PyArrayObject *irvec_to_np<T>(vector<ImageRef> &points)`
```c
#define CODE(Type, PyType) \ 
template<> struct Code<Type>\ 
{\ 
  static const int type = PyType;\ 
  static string name(){ return #Type;}\ 
  static char code(){ return PyType##LTR;}\ 
}
```
PythonExt: type checking

CODE(unsigned char , NPY_UBYTE );
CODE(char          , NPY_BYTE  );
CODE(short         , NPY_SHORT );
CODE(unsigned short, NPY_USHORT);
CODE(int           , NPY_INT   );
CODE(unsigned int  , NPY_UINT  );
CODE(float         , NPY_FLOAT );
CODE(double        , NPY_DOUBLE);
template<class I, class P> BasicImage<I>
pyobject_to_basic_image(P* p, const string& n="") { 
  if (!PyArray_Check(p) || PyArray_NDIM(p) != 2 
    || !PyArray_ISCONTIGUOUS(p) 
    || PyArray_TYPE(p) != Code<I>::type) 
    throw string(n + " must be a contiguous array of " + 
          Code<I>::name() + " (type code " + Code<I>::code() + 
          ")!");

  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;
}
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;
    }
}
Type Generality: no if statements!

```cpp
struct End{};

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

typedef TypeList<char,
    TypeList<unsigned char,
    TypeList<short,
    TypeList<unsigned short,
    TypeList<int,
    TypeList<unsigned int,
    TypeList<float,
    TypeList<double, End> > > > > > > > CVDTypes;
```
Type Generality: no if statements!

template<class List> struct load_image_by_type_letter
{
    static PyObject* load(const string& fname, char type_letter)
    {
        typedef typename List::type type;
        typedef typename List::next next;

        if(type_letter == Code<type>::code())
            return image_load_by_type<type>(fname);
        else
            return load_image_by_type_letter<next>::load(fname, type_letter);
    }
};
Type Generality: no if statements!

```cpp
template<> struct load_image_by_type_letter<End> {
    static PyObject* load(const string& fname,
                           char type_letter) {
        char L[2] = {type_letter, 0};
        throw string("Can't load image in to unknown type: ") + L;
    }
};
```
C++ Extensions

- ctypes
  - Three wrappers needed per function
  - Bug prone
  - Conversion code messy
- Boost::Python
  - Object lifetime issues
- PythonExt
  - templated greatness: type checking, type generality, clean conversion functions
  - easy error handling: throw an exception, catch in one place
  - is around to stay!
Comparison with mex

- separate source file for each function!
- No `PyArg_ParseTuple` or equivalent
- Opening mex with gdb
  - Cumbersome
  - Difficult to pin down segmentation faults
- Lacks succinctness and expressibility
  - Temptations to copy code: leads to bugs
Use Linux or Mac OS X whenever possible
- Windows: not the best scientific computing OS
- Memory manager is *wimpy*
  - Allocation of large buffers: very problematic
  - Not aggressive about cleaning up data
- *Processing* does not work as well
  - Memory leaks
- Hard to get optimizations right
  - Core dumps optimized code requiring aligned memory – not a problem on Linux
- Nice installers with *distutils*
MATLAB

- object-oriented infrastructure
- objects are immutable
- one directory per class
- one file per method
- Pass-by-value: global variables
- Not really good for production systems
- Richer data structures often encoded with matrix
  - graphs
  - trees
Python: Production Capable

- Can code richer data structures
  - Graphs, trees, lists
- Good for organizing larger code bases
  - Production systems
- Universal language
  - Lots of GUI toolkits, networking libraries, database suites, etc.
- MATLAB-like: simple calling conventions
Conclusion

**SciPy**
- A good choice!
- Easy to implement extensions to handle large data sets
- Python provides a nice extension framework
- C++ templated helpers and exceptions do the job!
- Easy to write prototype code in Python+Weave
- Universality and Separation of Concerns
  - Lots of libraries out there when your app becomes more sophisticated!
  - Good quality code!