A User's Perspective On Pyrex

William Stein
http://modular.fas.harvard.edu

Massachussetts Python Meeting: November 11, 2004

Abstract

Pyrex is a language for writing Python extension classes that run at C speed. In this 15 minute talk I will briefly describe Pyrex, why it useful for the software I'm currently writing, and how it compares to other systems for my application.

The Pyrex manual describes Pyrex as

A smooth blend of the finest Python with the unsurpassed power of raw C.

1 Pyrex

- The author of Pyrex is **Greg Ewing**, of University of Canterbury, New Zealand. The Pyrex web page is:

  `http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/`

- There is an active mailing list:

  `http://lists.copyleft.no/mailman/listinfo/pyrex`

- Pyrex is a language very similar to Python for writing Python extension modules. It is easy to use C functions from Pyrex, and much Python code also works in Pyrex.
- I'm just an enthusiastic Pyrex user. I have no connection with Pyrex development, and have not read the source code, which appears to be about 13000 lines of pure Python.

2 MANIN: A Python Modular Forms package

MANIN: I'm writing a Python package for number theorists; it's like the SciPy package, but for number theory (and cryptography?). The first version is aimed mainly at computing with "modular forms", which are holomorphic functions whose power series expansion encodes deep arithmetic information. I will say nothing further about modular forms here, except to remark that computing them involves sparse and dense linear algebra over exact base fields (e.g., finite fields, the rational numbers, and number fields). For my application, **speed** and **memory efficiency** are crucial, and must meet or exceed what is already available in the competing modular forms programs that are partly written in C or C++.

MAGMA: Currently the non-profit Australian computer algebra system MAGMA dominates in number theory computation. It's expensive, and has several drawbacks (e.g., closed source, no "Pickle" of objects, no user defined classes, no graphics). MAGMA is over 2 million lines of C code and a few hundred thousand lines of code written in the MAGMA language. Thus MAGMA is written in a hybrid of C and an interpreted language. This hybrid approach seems to be the best way to write a fast sophisticated computer algebra system.

NZMATH: A Japanese mathematician is trying to make a serious number theory package in **pure** Python. See `http://tnt.math.metro-u.ac.jp/nzmath/`. Pure Python seems like a bad idea to me, since it is too slow.

3 How Pyrex is Helpful

- It's difficult to create a number theory package in pure Python, because Python is too slow for **certain** things. For example, multiplying two matrices over the integers modulo a small prime p is fast in C and slow in pure Python, as we will see below.
- I will now give a complete example of the simple matrix-multiplication algorithm in pure Python and Pyrex. (There are better asymptotically fast "divide and conquer" matrix multiplication algorithms that grew out of work of Strassen, which I will not discuss.)
- Excellent support for exception handling in extension modules.

4 Pure Python Implementation of Matrix Multiply

We multiply two square matrices modulo a small prime p. I have not included error checking.

```
# mult1.py
class Matrix:
    def __init__(self, p, n, entries=None):
        """p -- prime
           n -- positive integer
           entries -- entries of the matrix (defaults to None, which means 0 matrix).
        """
        self.__n = n; self.__p = p
        if entries != None:
            self.__entries = [x % p for x in entries]
        else:
            self.__entries = [0 for _ in range(n*n)]

    def __repr__(self):
        s = ""; n = self.__n
        for i in range(n):
            for j in range(n):
                s += "%s, "%self.__entries[n*i + j]
            s += "\n"
        return s

    def __mul__(self, other):
        ans = []; n = self.__n
        for i in range(n):
            for j in range(n):
                v = [self.__entries[i*n+k] * other.__entries[k*n+j] for k in range(n)]
                ans.append(sum(v) % self.__p)
        return Matrix(self.__p, n, ans)
```

Example usage:

```
>>> import mult1
>>> a = mult1.Matrix(101,3,range(9))
>>> a
0, 1, 2,
3, 4, 5,
6, 7, 8,
>>> a*a
15, 18, 21,
42, 54, 66,
69, 90, 10,
```

5 Pyrex Implementation of Matrix Multiply

I copied `mult1.py` to `mult2.pyx`, and made the appropriate modifications.

```
# mult2.pyx
cdef extern from "Python.h":    # we will use these C functions below
    void* PyMem_Malloc(int)
    void PyMem_Free(void *p)

cdef class Matrix:
    cdef int *entries
    cdef int p, n

    def __new__(self, int p, int n, entries=None):
        self.p = p; self.n = n
        self.entries = <int*> PyMem_Malloc(sizeof(int)*n*n)   # cast to int pointer

    def __dealloc__(self):
        PyMem_Free(self.entries)         # using a C function

    def __init__(self, int p, int n, entries=None):
        """ p -- prime
            n -- positive integer
            entries -- entries of the matrix (defaults to None, which means 0 matrix).
        """
        cdef int i, j, k, x
        if entries != None:
            for i from 0 <= i < self.n:         # Pyrex's version of for loop;
                for j from 0 <= j < self.n:     # get converted to C with no Python calls
                    k = i*self.n + j
                    x = entries[k] % p
```

```
                if x<0: x = x + p
                self.entries[k] = x
    else:
        for i from 0 <= i < self.n:
            for j from 0 <= j < self.n:
                self.entries[i*self.n + j] = 0

def __repr__(self):
    cdef int i, j, n
    s = ""
    n = self.n
    for i from 0 <= i < n:
        for j from 0 <= j < n:
            s = s + "%s, "%self.entries[n*i + j]
        s = s + "\n"
    return s

cdef Matrix __mul__(Matrix self, Matrix B):
    cdef int s, i, j, k, n
    cdef Matrix ans
    ans = Matrix(self.p, self.n)
    n = self.n
    for i from 0 <= i < n:
        for j from 0 <= j < n:
            # The i,j entry of the product
            s = 0
            for k from 0 <= k < n:
                s = (s + (self.entries[i*n+k] * B.entries[k*n+j])) % self.p
                if s < 0: s = s + self.p
                ans.entries[i*n+j] = s
    return ans
```

We first convert `mult2.pyx` to a C program, then compile it to a shared object extension library:

```
# pyrex mult2.pyx         # convert pyx file to a 677 line pure C file.
# ls -lh mult2.c
-rw-r--r--  1 was was 23K Nov 11 15:16 mult2.c
# gcc -shared -O3 -fPIC -I/home/was/local/include/python2.4 mult2.c -o mult2.so
```

```
mult2.c: In function ‘__pyx_f_5mult2_6Matrix___mul__’:
mult2.c:302: warning: use of cast expressions as lvalues is deprecated   # yikes!
mult2.c:316: warning: use of cast expressions as lvalues is deprecated   # many warnings...
mult2.c:362: warning: use of cast expressions as lvalues is deprecated
mult2.c: At top level:
mult2.c:427: warning: initialization from incompatible pointer type      # yikes!
```

```
# ls -lh mult2.so
-rwxr-xr-x  1 was was 17K Nov 11 15:16 mult2.so
```

Now we can use the mult2 module, which has the same interface as mult.

```
>>> import mult2
>>> a = mult2.Matrix(101,3,range(9))
>>> a*a
15, 18, 21,
42, 54, 66,
69, 90, 10,
```

6 Some Generated C Code

```
/* Generated by Pyrex 0.9.3 on Thu Nov 11 15:16:11 2004 */
#include "Python.h"
#include "structmember.h"
typedef struct {PyObject **p; char *s;} __Pyx_InternTabEntry; /*proto*/
...
/* Declarations from mult2 */
staticforward PyTypeObject __pyx_type_5mult2_Matrix;
struct __pyx_obj_5mult2_Matrix {
  PyObject_HEAD
  struct __pyx_vtabstruct_5mult2_Matrix *__pyx_vtab;
  int (*entries);
  int p;
  int n;
};
...
static int __pyx_f_5mult2_6Matrix___new__(PyObject *__pyx_v_self, PyObject *__pyx_args, PyObject *__pyx_kwds); /*proto*/
static int __pyx_f_5mult2_6Matrix___new__(PyObject *__pyx_v_self, PyObject *__pyx_args, PyObject *__pyx_kwds) {
  int __pyx_v_p;
  int __pyx_v_n;
  PyObject *__pyx_v_entries = 0;
  int __pyx_r;
  static char *__pyx_argnames[] = {"p","n","entries",0};
  __pyx_v_entries = __pyx_k1;
  if (!PyArg_ParseTupleAndKeywords(__pyx_args, __pyx_kwds, "ii|O", __pyx_argnames, &__pyx_v_p, &__pyx_v_n, &__pyx_v_entries)) return -
  Py_INCREF(__pyx_v_self);
  Py_INCREF(__pyx_v_entries);
  /* "/home/was/talks/pyrex/mult2.pyx":22 */
  ((struct __pyx_obj_5mult2_Matrix *)__pyx_v_self)->n = __pyx_v_n;
...
```

7 Comparing Timings

I also wrote a **Psyco** version `mult3.py`, which is `mult1.py`, but with the line `import pysco; psyco.full()` appended at the top.
I used the following function to test the speed of each matrix multiply routine.

```
import timeit
def matmul(tries=10, p=97, n=20):
    T0 = timeit.Timer("a=mult1.Matrix(%s,%s,range(%s*%s)); a*a"%\
                      (p,n,n,n), "import mult1")
    T1 = timeit.Timer("a=mult2.Matrix(%s,%s,range(%s*%s)); a*a"%\
                      (p,n,n,n), "import mult2")
    T2 = timeit.Timer("a=mult3.Matrix(%s,%s,range(%s*%s)); a*a"%\
                      (p,n,n,n), "import mult3")
    t0 = T0.timeit(tries)
    t1 = T1.timeit(tries)
    t2 = T2.timeit(tries)
    print "Pure Python: %s\n\nPyrex: %s (Speedup: %s)\n"%(t0, t1, t0/t1)
    print "Psyco: %s (Speedup: %s)"%(t2, t0/t2)
```

Here are some timing results (under Python 2.3, since my Pysco install only supports Python 2.3), on my Pentium-M 1.6Ghz laptop.

```
>>> prof.matmul(1000)
Pure Python: 4.03225588799
Pyrex: 0.380643129349 (Speedup: 10.5932711695)
Psyco: 3.22930121422 (Speedup: 1.24864657104)
>>> prof.matmul(200, n=30)
Pure Python: 2.62041187286
Pyrex: 0.219752788544 (Speedup: 11.9243623265)
Psyco: 2.07065701485 (Speedup: 1.26549778842)
>>> prof.matmul(200, n=50)
Pure Python: 11.7379591465
Pyrex: 0.95321393013 (Speedup: 12.3140868754)
Psyco: 9.31631708145 (Speedup: 1.25993555649)
```

We gain a full order of magnitude (base 10) improvement by using Pyrex! That's very significant for my application, where matrix multiply is at the core of many algorithms. Also, Psyco doesn't seem to help much (this could be my ignorance about how best to use Pysco!).

8 Comparison with Psyco, SWIG, Boost, C

- **[Psyco]** Psyco is a "just in time compiler". As the above timings indicate, it is not as fast as Pyrex in our matrix multiplication example. Also, the Pysco web page says "Psyco currently uses a lot of memory. It only runs on Intel 386-compatible processors (under any OS) right now." These are both major obstructions for my application.

- **[SWIG]** SWIG wraps C++ classes as C-extension modules; the classes are also wrapped by pure Python classes, which can be a performance hit. For example, imagine implementing integers modulo p in C++/SWIG versus implementing them using Pyrex. Every multiply would be costly in SWIG. It is also painful to access Python objects from C++/SWIG (one uses "typemaps"). Finally, it is difficult **for me** to constantly switch between programming in C++ and programming in Python.

  The documentation for SWIG is *excellent*, it is a rapidly evolving system, and SWIG supports exposing C++ code to many other languages besides Python.

- **[Boost]** Boost.Python seems really neat when I read about it, but I found it way too complicated for my needs when I actually use it. I just couldn't get my head around it.

- **[C]** Implementing extensions directly in C is painful because of all the explicit reference counting. Also, again it is confusing (to me) to constantly switch between programming in C and programming in Python.

9 Drawbacks to Using Pyrex

- Important to be familiar with the books that come with Python called **The Python/C API** and **Extending and Embedding**.
- Several basic Python constructs are not yet supported in Pyrex. None of the following work:

```
a += 2
a = [i*i for i in range(10)]
```

- No direct support for C++ libraries or C++ code.
- Pyrex does not parse C header files. That's why we had

```
cdef extern from "Python.h":
    void* PyMem_Malloc(int)
    void PyMem_Free(void *p)
```

  at the beginning of `mult2.pyx`, instead of just "`cdef include "Python.h"`"

- Debuging can be confusing.
- It is sometimes tricky to divide Pyrex code into many files and call Pyrex C functions from one file in another file.