Pyrex, Pyrex, Pyrex, Pyrex, Pyrex, **SageX**

Martin Albrecht (malb@informatik.uni-bremen.de)

February 18, 2007

Outline

Outline

Basic Facts about Pyrex

Pyrex lets you write code that mixes Python and C data types any way you want, and compiles it into a C extension for Python.

(`http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`)

- Written by Greg Ewing of New Zealand.
- `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`
- Python-like code converted to C code that is compiled by a C compiler. All non-C memory management done automatically.
- Easy way to implement C extension modules for Python and to interface Python to C and C++ libraries.

What is SageX? I

The version of Pyrex shipped with SAGE is heavily patched. To reflect the huge difference to vanilla Pyrex it got renamed to SageX. Btw. The UrbanDictionary defines SageX as:

SageX *Amazingly cool. Omnipotent.*

(`http://www.urbandictionary.com/define.php?term=SageX`)

main changes:

- allow `cimports` across directories.
- several patches so that Pyrex works with Python 2.5, upstream ?
- SageX has list comprehension
- Code inspection mostly works

What is SageX? II

Time-critical SAGE code gets implemented in Pyrex, which **is (as fast as) C** *code, but easier to read (e.g., since all variables and scopes are explicit).*

(`http://modular.math.washington.edu/talks/2006-07-09-cnta/2006-07-09-cnta.pdf`)

"is (as fast as) C"

This is not necessarily true, you need to write almost C for this

lots of code in SAGE like library interfaces and basic arithmetic types already implemented in SageX

Possible Alternatives: Psycho

In short: run your existing Python software much faster, with no change in your source. Think of Psyco as a kind of just-in-time (JIT) compiler.
[. . .]
Benefits *2x to 100x speed-ups, typically 4x, with an unmodified Python interpreter and unmodified source code, just a dynamically loadable C extension module.* *(Not backed by our experience, M.A.)*
Drawbacks *Psyco currently uses a lot of memory. It* **only runs on Intel 386-compatible processors** *(under any OS) right now. There are some* **subtle semantic differences (i.e. bugs)** *with the way Python works; they* **should not be apparent in most programs**.

Possible Alternatives: CTypes

- you can actually use C datatypes in pure Python using CTypes
- won't speed up your existing code but you can call C functions to speed things up, interface an interesting library etc.
- shipped with Python 2.5
- How cool is that?

```
import ctypes

libc = ctypes.CDLL("/lib/libc.so.6", ctypes.RTLD_GLOBAL)
print libc.strlen("Hello!")
6
libm = ctypes.CDLL("/usr/lib/libm.so", ctypes.RTLD_GLOBAL)
libm.sin.argtypes = [ctypes.c_double]
libm.sin.restype = ctypes.c_double
libm.sin(1.0)
0.8414709848078965
```

- ask Josh

Possible Alternatives: SWIG/Boost

- ???

Outline

Getting Started with SageX

How do I port my code to Pyrex/SageX?

Just,

don't!

panic

Seriously, how do I port my code?

Seriously, don't! (for now)

A real example which was about to be *sagexed*:

Old Code (2 s)

```
def old_bit_vector(self):
  v = self.vertices()
  n = len(v)
  nc = n*(n - 1)/2
  bit_vector = '0'*int(nc)
  for e in self.edge_iterator():
    a = min(v.index(e[0]), v.index(e[1]))
    b = max(v.index(e[0]), v.index(e[1]))
    p = b*(b - 1)/2 + a
    bit_vector = bit_vector[:p] + '1' \
                 + bit_vector[p+1:]
  return bit_vector
```

New Code (0.48 s)

```
def new_bit_vector(self):
  v = self.vertices()
  n = len(v)
  nc = int(n*(n - 1))/int(2)
  bit_vector = set() # a python set!
  for e,f,g in self.edge_iterator():
    c = v.index(e)
    d = v.index(f)
    a,b = sorted([c,d])
    p = int(b*(b - 1))/int(2) + a
    bit_vector.add(p)
  bit_vector = sorted(bit_vector)
  s = []
  j = 0
  for i in bit_vector:
    s.append( '0'*(i - j) + '1' )
    j = i + 1
  s = "".join(s)
  s += '0'*(nc-len(s))
  return s
```

Premature SageXification

is the Root of all Evil

Before you port your class to SageX profile and test it!

- the iPython profiler frontend

```
sage: %prun for i in range(10000): _ = a+b
```

- cProfile

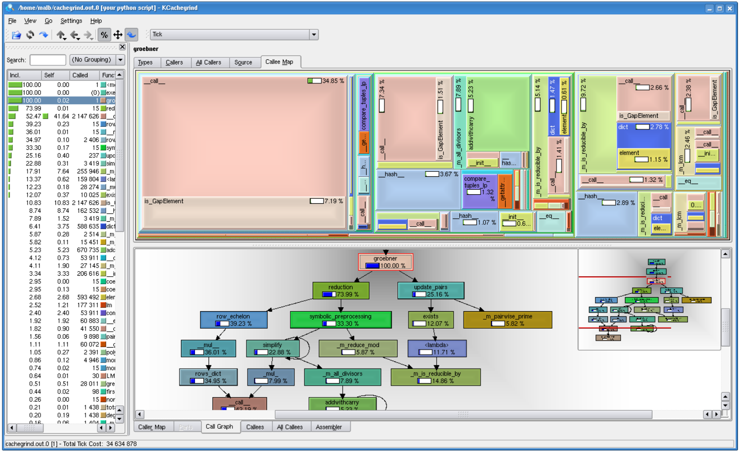
```
sage: import cProfile

# some setup
n = graphs.CompleteGraph(23)
g = Graph(n)

# profile!
sage: cProfile.run('g.show()')
```

- hotshot

```
sage: import hotshot
sage: filename = "pythongrind.prof"
sage: prof = hotshot.Profile(filename, lineevents=1)
sage: prof.run('g.show()')
sage: prof.close()
```

Hotshot

You may convert the output of hotshot using `hotshot2calltree` and view the result in `kcachegrind`.



Summary/Emphasize

- identify the portion of your code which needs to be speeded up carefully
- benchmark it for bottlenecks
- does any C library provide the functionality, can we call it?
- go ahead and have fun with SageX

Getting Started with SageX in SAGE

Now, how do I really port my code to Pyrex/SageX?

You may start writing SageX code by

- writing an `.spyx` file and loading/attaching it,
- put `%sagex` on top of a notebook cell, it will get compiled and executed, or
- write a `.pyx` file and add it to `setup.py`.

Now write your almost Python code, besides some exceptions:

SageX and Python Differences I

- No `__le__`, `__eq__`, `__ne__`, etc. but `__cmp__` and `__richcmp__`
- In `Class.__add__(left,right)` `left` doesn't need to be of type `Class`; no `__radd__` etc.
- Pickling (saving and loading objects) doesn't "just works", implement `__reduce__`
- no `yield`: Write an iterator class and implement `__next__` there.
- You can go around many limitations (like no yield) by calling `eval(' ')` in your Pyrex code (I don't like it, William does)

SageX and Python Differences II

- cdef you class to allow access from C but that invalids **AttributeError** programming like this:

```
try:
  return self.__cached_result
except AttributeError:
 self. __cached_result = self._calculate_result() #won't work
 return self.__cached_result
```

- Instead all members must be known at compile time:

```
cdef class MyClass
  cdef object __cached_result
   ...
  def calculate_result(MyClass self):
    if self.__cached_result is not None: #Note the 'is not'
      return self.__cached_result
    else:
      self.__cached_result = self._calculate_result()
      return self.__cached_result
```

Example I

inspired by a true story

Robert wanted to optimize a function which takes a string of 1s and 0s and returns some compressed format used in graph theory (similar to base64 encoded bytes). The function looked somewhat like this. Please note: This is not the original one, I made this one up for educational purposes.

```
def bitstringtosomerandomformat1(s):
  from sage.rings.integer_ring import ZZ
  if s == None: return "0"
  if not isinstance(s,str): raise TypeError, "need to get string
  s = s + "0" * (6-len(s)%6)

  res = ""

  for i in range(len(s)/6):
    res += chr(ZZ( s[i:i+6] , 2) + 63)
  return res


sage: time r = bitstringtosomerandomformat1(s)
CPU time: 1.33 s,  Wall time: 1.34 s
```

Example II

inspired by a true story

Let us SageX this:

```
%sagex

def bitstringtosomerandomformat2(s):
  from sage.rings.integer_ring import ZZ

  if s == None: return "0"

  if not isinstance(s,str): raise TypeError, "need to get string"

  s = s + "0" * (6-len(s)%6)

  res = ""

  for i in range(len(s)/6):
    res += chr(ZZ( s[i:i+6] , 2) + 63)
  return res


sage: time r = bitstringtosomerandomformat2(s)
CPU time: 4.95 s,  Wall time: 5.09 s
```

Lesson: It can actually slow you down

Example III

inspired by a true story

Let us optimize the Python code.

```
def bitstringtosomerandomformat3(s):
  from sage.rings.integer_ring import ZZ

  if s == None: return "0"

  if not isinstance(s,str): raise TypeError, "need to get string"

  s = s + "0" * (6-len(s)%6)

  res = []

  for i in range(len(s)/6): #fast string concatenation
    res.append( chr(ZZ( s[i:i+6] , 2) + 63) )
  return "".join(res)
```

```
sage: time r = bitstringtosomerandomformat3(s)
CPU time: 1.32 s,  Wall time: 1.43 s
```

Not much of a difference

Example IV

inspired by a true story

Let's SageXify the optimized version and apply some SageX tricks:

```
%sagex

from sage.rings.integer_ring import ZZ

def bitstringtosomerandomformat4(s):
   cdef int i, m # c ints

   if s is None: return "0"

   if not PY_TYPE_CHECK(s,str): raise TypeError, "need to get string"

   s = s + "0" * (6-len(s)%6)

   res = []

  m = len(s)/6
   for i from 0 <= i < m: # c for loop
      res.append(chr(int( s[i:i+6] , 2) + 63))
   return "".join(res)


sage: time r = bitstringtosomerandomformat4(s)
CPU time: 0.13 s,  Wall time: 0.13 s
```

Strike!

Example V

inspired by a true story

Is there a nicer way to do this, equally fast or faster? The silver bullet:

```
%sagex

from sage.rings.integer_ring import ZZ

def bitstringtosomerandomformat5(s):
  if s is None:
    return "0"

  if not PY_TYPE_CHECK(s,str): raise TypeError, "need to get string"

  s = s + "0" * (6-len(s)%6)

  res = [chr(int( s[i:i+6] , 2) + 63) for i in range(len(s)/6)]
  return "".join(res)


sage: time r = bitstringtosomerandomformat5(s)
CPU time: 0.11 s,  Wall time: 0.12 s
```

Lesson: Rober Bradshaw's list comprehension just rocks!

Outline

Tips to Gain Speed I

- SageX tries to make things easy for you which may interfear with speed.
- `cdef` all integers as `int` if possible
- Use `int` **for**-loops:

```
cdef int i #this is important!
for i from 0 <= i < n:
  # do something
```

  or use list comprehension
- SageX knows `cdef f()` functions/methods and `def f()` functions/methods. The later are callable from Python but calling them is much more expensive than calling a `cdef` function/method.
- Avoid Python! If you basically call heaps of Python code things won't be faster

Tips to Gain Speed II

- Use the macros in `stdsage.pxi` like `PY_NEW` and `PY_TYPE_CHECK`
- SageX plays safe when it comes to list, tuple, dict access: `t[0]` gets translated to:

```
__pyx_1 = PyInt_FromLong(0);
if (!__pyx_1) {
  __pyx_filename = __pyx_f[0];
  __pyx_lineno = 10;
  goto __pyx_L1;
}
__pyx_3 = PyObject_GetItem(__pyx_v_t, __pyx_1);
if (!__pyx_3) {
  __pyx_filename = __pyx_f[0];
  __pyx_lineno = 10;
  goto __pyx_L1;
}
Py_DECREF(__pyx_1); __pyx_1 = 0;
Py_DECREF(__pyx_3); __pyx_3 = 0;
```

Tips to Gain Speed III

This is faster:

```
cdef extern from "Python.h":
    void* PyTuple_GET_ITEM(object p, int pos)

w = <object> PyTuple_GET_ITEM(t,0)
```

The macro `FAST_SEQ_UNSAFE` is even faster, as it allows access using a C array.

- So use Python C API directly, but be carefull with **refcounting**

Outline

C++ I

it's only str().replace("some string","some other string")

SageX knows no classes but it knows structs and function pointers. Those "look" like methods in classes when feed to a C++ compiler.

```
cdef extern from "linbox/field/givaro-gfq.h":

    ctypedef struct GivaroGfq "LinBox::GivaroGfq":
        #attributes
        int one
        int zero

        # methods
        int (* mul)(int r, int a, int b)
        ...
        unsigned int (* characteristic)()
        ....
    GivaroGfq *gfq_factorypk "new LinBox::GivaroGfq" (int p, int k)
    GivaroGfq *gfq_factorypkp "new LinBox::GivaroGfq" (int p, int k, intvec poly)
    #actually, 'orig[0]' does the same
    GivaroGfq  gfq_deref "*"(GivaroGfq *orig)
    void delete "delete "(void *o)
    int gfq_element_factory "LinBox::GivaroGfq::Element"()
```

C++ II

it's only str().replace("some string","some other string")

This class may now be used like this:

```
def some_function():
  cdef GivaroGfq *k
  cdef int e
  k = gfq_factorypk(2,8)
  e = k.mul(e,k.one,k.zero)
  delete(k)
```

To ensure that the resulting C++ code is feed to a C++ compiler specify `language='c++'` in setup.py:

```
linbox_gfq = Extension('sage.libs.linbox.finite_field_givaro',
                      sources = ["sage/libs/linbox/finite_field_givaro.pyx"],
                      libraries = ['gmp', 'gmpxx', 'm', 'stdc++', 'givaro', 'linbox'],
                      language='c++'
                      )
```

C++ III

it's only str().replace("some string","some other string")

- **Templates** are **not supported** but “C name specifiers” allow to deal with templates:

```
cdef extern from "linbox/integer.h":
   ctypedef struct intvec "std::vector<LinBox::integer>":
       void (* push_back)(int elem)

   intvec intvec_factory "std::vector<LinBox::integer>"(int len)
```

- **Overloading** of functions/methods is **not supported**. Create a C alias for every combination.
- If everything else fails: You can always wrap the C++ code in a **C function** and call this from SageX. However this introduces a function call as overhead (need to check if this can be inlined). LinBox wrapper written this way, function call costs neglectable, much easier to write in C++ than in SageX.

Outline

Inclusion of a C Data Structure Library

I still propose **libcprops**

- http://cprops.sourceforge.net/
- pro: ANSI-C (which both SageX and I understand much better than C++)
- pro: data structures: **linked_list**, heap, priority_list, **hashtable**, hashlist, **avltree**. red-black tree ...
- pro: thread safe (!!!)
- pro: easy to read, I could adapt it

...I started using it during the MSRI workshop for polynomial representation.

Questions?

Thank You!