Speeding up Python with C/C++

- Fernando Pérez, Grad. Student, Physics Dept.
- A background of Pascal, C/C++, Perl, Python (and many others), roughly in that order.
  - Recent Python project: IPython, a better interactive interpreter (http://www-hep.colorado.edu/~fperez/ipython/)
- Speed and computers: *"Early Optimization is the root of all evil"* - Donald Knuth.
  - Speed of execution: C/C++, Fortran, assembly
  - Speed of development: Perl, Python (Java).
  - Good software design: a balancing act.
- In *many* cases, Python's speed is enough.

How to speed it up when you need to

- By hand: cumbersome, tricky, time-consuming.
- SWIG: http://www.swig.org
  - Good for wrapping big existing C/C++ libraries.
- Boost.Python: http://www.boost.org/libs/python/doc/
  - Similar to SWIG, more C++ oriented.
- Weave - part of SciPy: http://scipy.org/
  - Direct inlining of C/C++ code in Python.
- PyInline: http://pyinline.sourceforge.net/ and Pyrex: http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex/.
  - Related to weave in spirit, still far from production-ready.

Weave - Part of SciPy

- weave.ext_tools()
  - Easier building of extension modules (SWIG).
- weave.inline()
  - Inlining of C++ code within Python code.
- weave.blitz()
  - Auto-compilation for Numeric expressions.
- weave.accelerate()
  - Automatic acceleration of Python code - *NEW*

Often Python *is* fast enough

- Consider the following two trivial functions

```
def py_print(input):
    print "Input:",input

def c_print(input):
    code = """printf("Input: %i \\n",input);"""
    weave.inline(code,['input'])
```

Timing results

```
In [15]: time_test (5000,py_print, 42)
Out[15]: 0.13
In [17]: time_test (5000,c_print, 42)
Out[17]: 0.21
```

- C is *slower* than Python ???
- There is some overhead involved in weave.
- Python's internal functions are fairly efficient and well tied into the core.
- Don't optimize unless you *really* need to.

Sometimes, you *do* need speed

- Consider building a matrix of the form [1]:

$$M_{kl} = \frac{1}{\sqrt{N}} \exp\left(i\left[\frac{2\pi}{N}\left(k^2 - kl + l^2\right) + \frac{N}{2\pi}\kappa \sin\left(\frac{2\pi}{N} l\right)\right]\right)$$

- First a pure Python solution

```
def quantum_cat_python(N,kappa):
    # First initialize complex matrix with NxN elements
    mat=zeros((N,N), Complex)
    # precompute a few things outside the loop
    sqrt_N_inv = 1.0/sqrt(N)
    alpha = 2.0*pi/N
    kap_al = kappa/alpha
    # now we fill each element
    for k in range(0,N):
        for l in range(0,N):
            mat[k,l] = sqrt_N_inv * \
                       cmath.exp(1j*(alpha*(k*k-k*l+l*l) + \
                       kap_al*sin(alpha*l)))
    return(mat)
```

[1] Arnd Bäcker, Ulm University: http://www.physik.uni-ulm.de/theo/qc/baec/qmaps.html

Using Numeric Python

- High-level, array-oriented package (like IDL)
- Very well optimized, extensive library.

```
def quantum_cat_numeric(N,kappa):
    alpha = 2.0*pi/N
    mat_fn = lambda k,l: alpha*(k*k-k*l+l*l)
    phi = fromfunction(mat_fn,(N,N)) + \
           (kappa/alpha)*sin(alpha*arange(N))
    return (1.0/sqrt(N))*exp(1j*phi)
```

Using weave.inline(). Inner loop in C

```
def quantum_cat_weave(N,kappa):
    phi = zeros((N,N), Float)    # Initialize phase matrix
    support = "#include <math.h>"
    code = """
float alpha = 2.0*pi/N;
float kap_al = kappa/alpha;

for (int k=0;k<N;++k)
  for(int l=0;l<N;++l)
    phi(k,l) = alpha*(k*k-k*l+l*l) + kap_al*sin(alpha*l);
"""
    # Call weave to fill in phi
    weave.inline(code,['N','kappa','pi','phi'],
                 type_converters = converters.blitz,
                 support_code = support,libraries = ['m'])
    return (1.0/sqrt(N))*exp(1j*phi)
```

Timing results

```
In [32]: N
Out[32]: 300

In [33]: kappa
Out[33]: 0.29999999999999999

In [34]: time_test(1,quantum_cat_python,N,kappa)
Out[34]: 4.7399999999999984

In [35]: time_test(1,quantum_cat_numeric,N,kappa)
Out[35]: 0.32000000000000028

In [36]: time_test(1,quantum_cat_weave,N,kappa)
Out[36]: 0.19999999999999929

In [37]: _34/_35
Out[37]: 14.812499999999982

In [38]: _34/_36
Out[38]: 23.700000000000077
```

Some lessons learned

- Manual optimization is often unnecessary.
- Look for good libraries for your problem first.
- Python function calls are expensive.
  - If you need to optimize in C/C++, try to avoid calling back into Python.
- Straightforward optimizations: tight loops over large data structures.
- Lots of work is being done
  - It's easier every day (weave, Pyrex, PyInline, ...)
- Python has a bright future for scientific computing (SciPy, NumArray, others...)