Speeding up statistical computations in Python using numexpr and cython

In the age of "big data", many are interested in optimizing the performance of the high-level languages that we use to analyse data. NumPy gets us part of the way there on Python:

- Storage of multidimensional data
- Efficient data access
- Efficient in-memory storage
- Fast methods and functions for data manipulation

Fast array expression evaluation with numexpr

numexpr allows array expressions to be evaluated far faster than what can be achieved in Python using Numpy arrays. numexpr parses a string expression and optimizes and compiles the code on the fly, using a virtual machine that includes a JIT compiler. In addition, numexpr offers direct support for parallel multi-threaded computations, since Python's global interpreter lock is bypassed.

Part the reason Python can be slow for array calculations is that it creates temporary arrays to store intermediate results from array element calculations, which wastes memory and cache. numexpr handles such calculations in manageable chunks, which accelerates computation.

Example: computing a polynomial

```
In [76]: import numpy as np
x = np.linspace(-1, 1, 1e7)
0.25*x**3 + 0.75*x**2 - 1.5*x - 2

Out[76]: array([ 0.00000000e+00, -4.50000045e-07, -9.00000090e-07,...,
-2.50000030e+00, -2.50000015e+00, -2.50000000e+00])

In [77]: timeit 0.25*x**3 + 0.75*x**2 - 1.5*x - 2
1 loops, best of 3: 551 ms per loop

In [78]: import numexpr as ne
ne.set_num_threads(1)

In [79]: ne.evaluate('0.25*x**3 + 0.75*x**2 - 1.5*x - 2')

Out[79]: array([ 0.00000000e+00, -4.50000045e-07, -9.00000090e-07,...,
-2.50000030e+00, -2.50000015e+00, -2.50000000e+00])

In [80]: timeit ne.evaluate('0.25*x**3 + 0.75*x**2 - 1.5*x - 2')
10 loops, best of 3: 90.7 ms per loop

In [81]: ne.set_num_threads(4)
```
The CPU spends a lot of time waiting for memory to give it computations.

![Graph showing the performance gap between processor and memory over time.](image)

CPU caches are often used to make up for this difference. CPU caches are more effective when the data are optimally located in memory to take advantage of cache performance. `numexpr` does this by moving contiguous blocks of data from memory to the CPU cache, reusing them as much as possible within the cache to more quickly give the CPU access to the data.

**Limitations**

`numexpr` only implements element-wise operations. So, \(a \times b\) becomes:

```python
for i in range(N):
    c[i] = a[i] * b[i]
```

Similarly, it cannot index other parts of arrays in the same expression:

```python
for i in range(N):
    c[i] = a[i-1] + a[i] * b[i]
```

**Extending Python with Cython**

Cython is a programming language based on Python with extra syntax to provide static type declarations. This takes advantage of the benefits of Python while allowing one to achieve the speed of C. It actually generates C extensions that are then compiled. It has the advantage of integrating well with NumPy and supports multicore processing.

**Example: numerical integration**

```python
timeit ne.evaluate('0.25*x**3 + 0.75*x**2 - 1.5*x - 2')
```

10 loops, best of 3: 52.9 ms per loop
The function $f(x)$ (in blue) is approximated by a linear function (in red).

Trapezoidal rule
This article is about the quadrature rule for approximating integrals. For the implicit trapezoidal rule for solving initial value problems, see Trapezoidal rule (differential equations). For the explicit trapezoidal rule for solving initial value problems, see Heun's method.

In numerical analysis, the trapezoidal rule (also known as the trapezoid rule or trapezium rule) is an approximate technique for calculating the definite integral

$$\int_{a}^{b} f(x) \, dx.$$ 

The trapezoidal rule works by approximating the region under the graph of the function $f(x)$ as a trapezoid and calculating its area. It follows that

$$\int_{a}^{b} f(x) \, dx \approx (b - a) \frac{f(a) + f(b)}{2}.$$ 

Example: Gibbs sampling

We wish to estimate the parameters of the joint distribution of $x$ and $y$:

$$f(x, y) = x \exp(-xy^2 - y^2 + 2y - 4x)$$

Gibbs sampling does this by conditional sampling of each stochastic parameter in turn, conditional on the current values of the other.
parameters. Here, the appropriate conditional distributions are:

\[
\begin{align*}
 x|y & \sim \text{Gamma}(3, y^2 + 4) \\
 y|x & \sim \text{Normal}\left(\frac{1}{1+x}, \frac{1}{2(1+x)}\right)
\end{align*}
\]

In [ ]:

```python
import numpy as np
from numpy import zeros, random, sqrt
gamma = random.gamma
normal = random.normal

def gibbs(N=20000, thin=200):
    mat = zeros((N,2))
    x, y = mat[0]
    for i in range(N):
        for j in range(thin):
            x = gamma(3, y**2 + 4)
            y = normal(1./(x+1), 1./sqrt(2*(x+1)))
        mat[i] = x, y
    return mat
```

In [103]:

```
timeit gibbs(20000, 200)

1 loops, best of 3: 26.9 s per loop
```

cython code:

```cython
from math.h:
cdef extern "sqrt(double)"

cdef from "gsl/gsl_rng.h"
    ctypedef struct gsl_rng_type
    ctypedef struct gsl_rng
    gsl_rng_type *gsl_rng_mt19937
    gsl_rng *gsl_rng_alloc(gsl_rng_type * T) nogil

cdef from "gsl/gsl_randist.h"
    double gamma "gsl_ran_gamma"(gsl_rng * r, double, double)
    double gaussian "gsl_ran_gaussian"(gsl_rng * r, double)

cdef gsl_rng *r = gsl_rng_alloc(gsl_rng_mt19937)

@cython.wraparound(False)
@cython.boundscheck(False)
def gibbs(int N=20000,int thin=500):
    cdef:
        double x=0
        double y=0
        int i, j
    ndarray[float64_t, ndim=2] samples
    samples = np.empty((N,thin))
    for i from 0 <= i < N:
        for j from 0 <= j < thin:
            x = gamma(r,3,1.0/(y*y+4))
```

http://127.0.0.1:8888/8242bf4e-8373-4ed3-bb65-69696578d398/print
```python
y = gaussian(r, 1.0/sqrt(x+1))
samples[i, 0] = x
samples[i, 1] = y
return samples
```

```
In [106]: import simplegibbs_cython

In [108]: timeit simplegibbs_cython.gibbs(20000, 200)
```

```
1 loops, best of 3: 550 ms per loop
```

```
In [ ]:
```