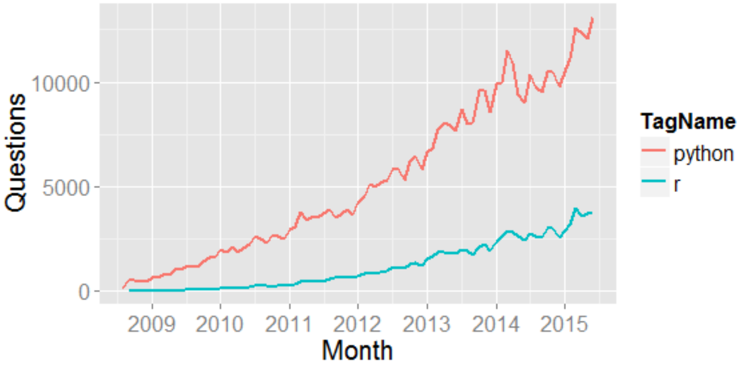
How to code like Bruce Lee fights

Some things I have learned about computation

Tea talk, Heiko

10th May 2016

Computation & Science



From blog entry ‘In celebration of 100,000 R questions on StackOverflow‘

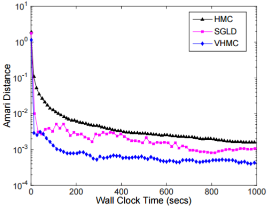
Matlab, Python, R, Julia & co

- High level programming is convenient
  - No explicit control over memory
  - Limited control over computation
  - Type-free
  - Readable code (?)

Promise: The language developers will sort it out

A typical NIPS paper needs this plot

(Marginal) improvements in performance/time



Scalable Variational Gaussian Process Classification

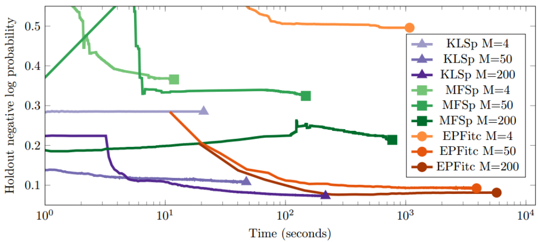


Figure 2: Temporal performance of the different methods on the *image* dataset.

Problems

- Automatic memory managment comes at a cost
- Runtime type inference comes at a cost
- Affects readability
  - function calls & indexing become expensive
  - compensate using "flattened" and "vectorised" code

- Most (research) codes do not nearly exploit the hardware
- Giving away the control might make that impossible

Solutions (I would call hacks)

- Write critical parts in C
- Things like Cython (type/compile system for Python)
- Impossible to read, write, maintain ...
- ... and more critical: to validate and reproduce

A story about π

6 July 1997	Yasumasa Kanada and Daisuke Takahashi	HITACHI SR2201 (1024 CPU) [20]		**51,539,600,000**
5 April 1999	Yasumasa Kanada and Daisuke Takahashi	HITACHI SR8000 (64 of 128 nodes) [21]		**68,719,470,000**
20 September 1999	Yasumasa Kanada and Daisuke Takahashi	HITACHI SR8000/MPP (128 nodes) [22]		**206,158,430,000**
24 November 2002	Yasumasa Kanada & 9 man team	HITACHI SR8000/MPP (64 nodes), Department of Information Science at the University of Tokyo in Tokyo, Japan [23]	600 hours	**1,241,100,000,000**
29 April 2009	Daisuke Takahashi et al.	T2K Open Supercomputer (640 nodes), single node speed is 147.2 gigaflops, computer memory is 13.5 terabytes, Gauss–Legendre algorithm, Center for Computational Sciences at the University of Tsukuba in Tsukuba, Japan[24]	29.09 hours	**2,576,980,377,524**

8 months later

Fabrice Bellard beats previous world record:

- $2.6 \cdot 10^9$ digits
- Using **a single** Intel i7 quad core
  - 46.9 gigaflops
  - 3000 USD
  - 131 days
- Takahashi: 640 quad cores, roughly 2000x faster
  - 94.2 Tflops (trillion floating point operations per second)
  - Multi-million USD
  - 29 hours
- Bellard only 96 times slower, speedup is 20x

http://bellard.org/pi/pi2700e9/faq.html

The π algorithms are:

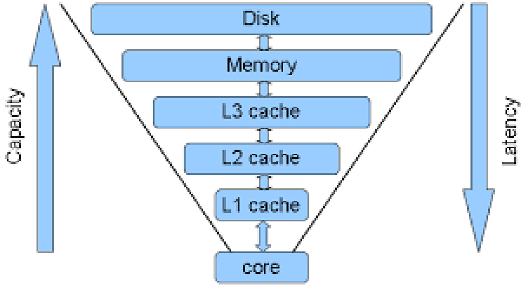
- IO bound – very heavy communication between the nodes

Bellard's algorithm:

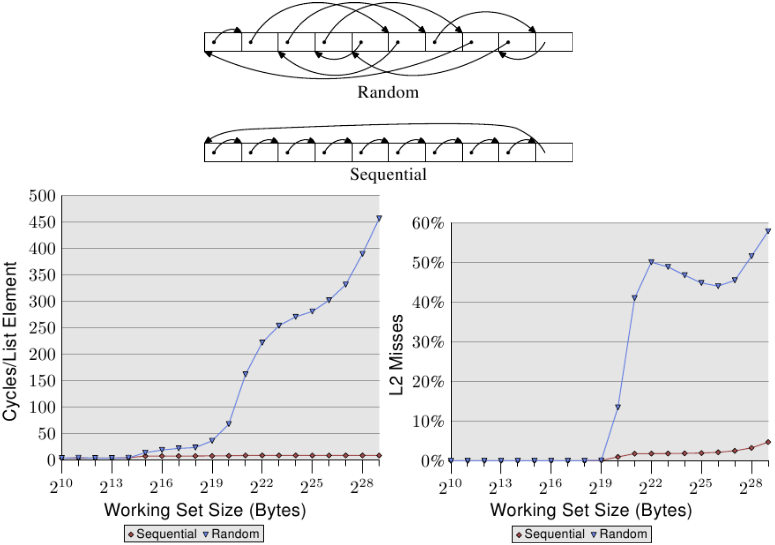
- Chudnovsky series evaluated using binary splitting
- **Asymptotically slower** than Arithmetic-Geometric Mean by Takahashi

Asymptotics seem to be saturated at 10^{12} digits. Why faster?

CPU cache


Disk
Memory
L3 cache
L2 cache
L1 cache
core
Capacity
Latency

Locality matters when accessing memory

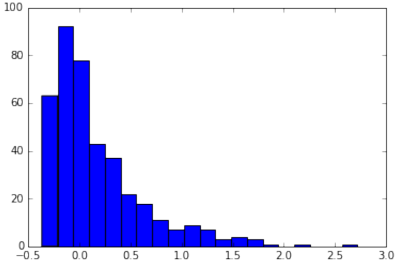


Example: MMD permutation test

- Recall Arthur's kernel two-sample test.
- Each n samples $x_i \sim p$ and $y_i \sim q$

$$n^2 \mathrm{MMD}^2 = \sum_{i,j} k(x_i, x_j) + k(y_i, y_j) - 2k(x_i, y_j)$$

- Testing requires the distribution of MMD^2 under $p = q$
- Analytically hard, so simulate empirical version



Pseudo-code:

```
N = 1000; X = randn(N); Y = laplace(N)
XY = stack(X,Y)

null = zeros(100)
for rep in 1..100
        p = index_permutation(2*N)
        XY = XY[p]
        X, Y = split(XY)

        for i,j in 1..N
                null[rep] = null[rep]
                          + k(X[i], X[j])
                          + k(Y[i], Y[j]
                          - 2*k(X[i], Y[j])
        end for
end for
```

MATLAB (E.g. the code in Gretton et al.):

```
N = 1000; X = randn(N); Y = laplace(N)
XY = stack(X,Y)

null = zeros(100)
for rep in 1..100
        p = index_permutation(2*N)
        XY = XY[p] % CREATES COPY
        X, Y = split(XY) % CREATES COPY

        for i,j in 1..N % EXTREMELY SLOW
                null[rep] = null[rep]
                            + k(X[i], X[j])
                            + k(Y[i], Y[j]
                            − 2*k(X[i], Y[j])
        end for
end for
```

Comparison

- $N = 2000$ (moderate)
- 200 samples from null
- Precomputed kernel matrix

Implementation	**Seconds**	**Comment**
Matlab	230	copy

Python:

```
N = 1000; X = randn(N); Y = laplace(N)
XY = stack(X,Y)

null = zeros(100)
for rep in 1..100
        p = index_permutation(2*N)
        XY = XY[p] % CREATES VIEW
        X, Y = split(XY) % CREATES VIEW

        for i,j in 1..N % EVEN SLOWER
                null[rep] = null[rep]
                          + k(X[i], X[j])
                          + k(Y[i], Y[j]
                          - 2*k(X[i], Y[j])
        end for
end for
```

Comparison

- $N = 2000$ (moderate)
- 200 samples from null
- Precomputed kernel matrix

Implementation	**Seconds**	**Comment**
Matlab	230	copy
Python	200	view rather than copy

C/C++:

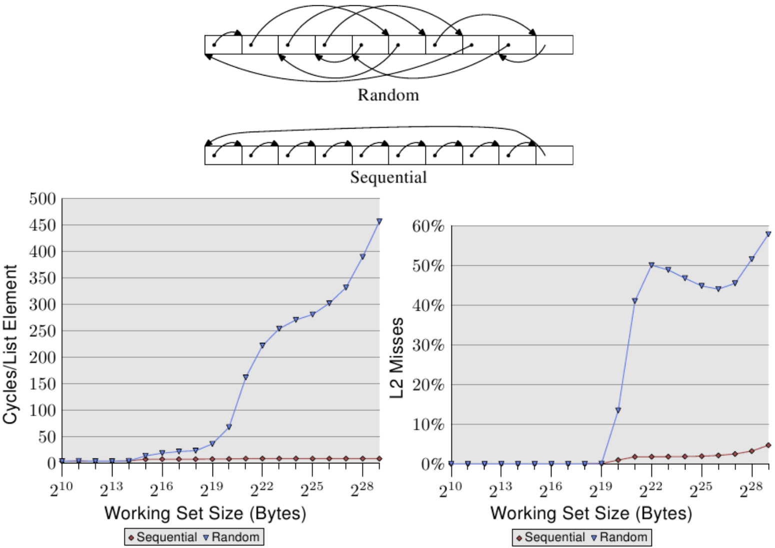
```
N = 1000; X = randn(N); Y = laplace(N)
XY = stack(X,Y)

null = zeros(100)
for rep in 1..100
        p = index_permutation(2*N)

        for i,j in 1..N
                null[rep] = null[rep]
                  + k(XY[p[i]], XY[p[j])
                  + k(XY[p[i+N]], XY[p[j+N]]
                  - 2*k(XX[p[i]], XY[p[j+N]])
        end for
end for
```

Comparison

- $N = 2000$ (moderate)
- 200 samples from null
- Precomputed kernel matrix

Implementation	**Seconds**	**Comment**
Matlab	230	copy
Python	200	view rather than copy
C/C++	120	random access

Locality matters when accessing memory



C/C++ (Rahul):

```
N = 1000; X = randn(N); Y = laplace(N)
XY = stack(X,Y)

null = zeros(100)
for rep in 1..100
        p = index_permutation(2*N)

        k_xx, k_yy, k_xy = 0

        for i,j in 1..N
                compute k(XY[i], XY[j+N])
                decide_which_term(p, i, j)
                update k_xx, k_yy, k_xy
        end for
end for
```

Comparison

- $N = 2000$ (moderate)
- 200 samples from null
- Precomputed kernel matrix

Implementation	**Seconds**	**Comment**
Matlab	230	copy
Python	200	view rather than copy
C/C++	120	random access
C/C++	60	sequential access

C/C++:

```
N = 1000; X = randn(N); Y = laplace(N)
XY = stack(X,Y)

null = zeros(100)
ps = 100_index_permutations(2*N)
k_xx, k_yy, k_xy = 0

for i,j in 1..N
        compute k(XY[i], XY[j+N])

        for rep in 1..100
                decide which terms
                update k_xx, k_yy, k_xy
                update null[rep]
        end
end for
```

Comparison

- $N = 2000$ (moderate)
- 200 samples from null
- Precomputed kernel matrix

Implementation	**Seconds**	**Comment**
Matlab	230	copy
Python	200	view rather than copy
C/C++	120	random access
C/C++	60	sequential access
C/C++	30	sequential & single sweep

Single sweep does not require to pre-compute kernel matrix
$\mathcal{O}(N^2) \Rightarrow O(N)$ memory

C/C++ and multicore:

```
N = 1000; X = randn(N); Y = laplace(N)
XY = stack(X,Y)

null = zeros(100)
ps = 100_index_permutations(2*N)
k_xx, k_yy, k_xy = 0

#pragma omp parallel for
for i,j in 1..N
        compute k(XY[i], XY[j+N])

        for rep in 1..100
                decide which terms
                update k_xx, k_yy, k_xy
                update null[rep]
        end
end for
```

Comparison

- $N = 2000$ (moderate)
- 200 samples from null
- Precomputed kernel matrix

Implementation	Seconds	Comment
Matlab	230	copy
Python	200	view rather than copy
C/C++	120	random access
C/C++	60	sequential access
C/C++	30	sequential & single sweep
C/C++	15	sequential sweep & dual-core

Single sweep does not require to pre-compute kernel matrix
$\mathcal{O}(N^2) \Rightarrow O(N)$ memory

Why this matters

A Fast, Consistent Kernel Two-Sample Test

Arthur Gretton
Carnegie Mellon University
MPI for Biological Cybernetics
arthur.gretton@gmail.com

Kenji Fukumizu
Inst. of Statistical Mathematics
Tokyo Japan
fukumizu@ism.ac.jp

Zaid Harchaoui
Carnegie Mellon University
Pittsburgh, PA, USA
zaid.harchaoui@gmail.com

Bharath K. Sriperumbudur
Dept. of ECE, UCSD
La Jolla, CA 92037
bharathsv@ucsd.edu

Abstract

A kernel embedding of probability distributions into reproducing kernel Hilbert spaces (RKHS) has recently been proposed, which allows the comparison of two probability measures P and Q based on the distance between their respective embeddings: for a sufficiently rich RKHS, this distance is zero if and only if P and Q coincide. In using this distance as a statistic for a test of whether two samples are from different distributions, a major difficulty arises in computing the significance threshold, since the empirical statistic has as its null distribution (where $P = Q$) an infinite weighted sum of χ^2 random variables. Prior finite sample approximations to the null distribution include using bootstrap resampling, which yields a consistent estimate but is computationally costly; and fitting a parametric

Why this matters

- The spectral test is theoretically quite complicated
- Motivated with its speed
- "our new distribution estimate is [...] computationally less costly than the bootstrap"
- "[...] due the requirement to repeatedly re-compute the test statistic"
- 64 citations on Google scholar

Why this matters

- $N = 2000$ (moderate)
- Eigendecomposition. Can't be optimised or parallelised.
- Scales $\mathcal{O}(N^3)$, so gets worse quickly

Implementation	**Seconds**	**Comment**
Matlab	230	copy
Python	200	view rather than copy
C/C++	120	random access
C/C++	60	sequential access
C/C++	30	sequential & single sweep
C/C++	15	sequential sweep & dual-core
Spectral	**60**	**single low-level call**

Conclusion

Machine Learning heavily focusses on computation

- Better be careful with statements à la
  - "Our algorithm is a X% speedup over the state-of-the-art"
  - "We provide an implementation in [R/Python/etc], with critical parts written in C"
  - "Trivial to parallelise"
- Structure of the (computational) problem matters
- Taking into account what the computer actually does helps
- Often, only low-level languages allow to exploit this

Thank you!