Counting points on K3 surfaces: some complexity guesstimates

David Harvey

University of New South Wales

20th October 2015
ICERM

Introduction

$X =$ variety over $\mathbf{Z}$

$p =$ prime

$X_p =$ reduction of X modulo p

Zeta function at p:

$$Z_p(T) = \exp\left(\sum_{r \geq 1} \frac{|X_p(\mathbf{F}_{p^r})|}{r} T^r\right) \in \mathbf{Q}(T).$$

Theorem (H., 2007)

Let X be a hyperelliptic curve of genus g. We may compute $Z_p(T)$ in

$$g^{O(1)} p^{1/2+o(1)}$$

bit operations.

Theorem (H., 2012)

Let X be a hyperelliptic curve of genus g. We may compute $Z_p(T)$ for all $p < N$ in

$$g^{O(1)} N(\log N)^{3+o(1)}$$

bit operations.

The latter is "average polynomial time" — $(g \log p)^{O(1)}$ per prime.

Are these algorithms practical? **Yes!**

Square-root time algorithm:

- Implemented in C++ for my thesis (2008)
- For $g = 4$ and $p \approx 10^{14}$, runs in about 21 hours

Average polynomial time algorithm:

- Baby version implemented by H.–Sutherland (2014)
- So far only $g = 2, 3$
- Computing $Z_p(T)$ for all $p < 2^{30} \approx 10^9$:
  - Genus two: 1.3 days. (Previously: 1.4 years with `smalljac`.)
  - Genus three: 4.0 days. (Previously: 3.8 years with `hypellfrob`.)
- $g \geq 4$ under development; toy implementation already exists

Theorem (H., 2014)

Let X be **any variety whatsoever**. *We may compute $Z_p(T)$ in*

$$O(p^{1/2+o(1)})$$

bit operations. We may compute $Z_p(T)$ for all $p < N$ in

$$O(N(\log N)^{3+o(1)})$$

bit operations.

Big-O constants depend on X.

Can this sort of algorithm be made **practical** for K3 surfaces?

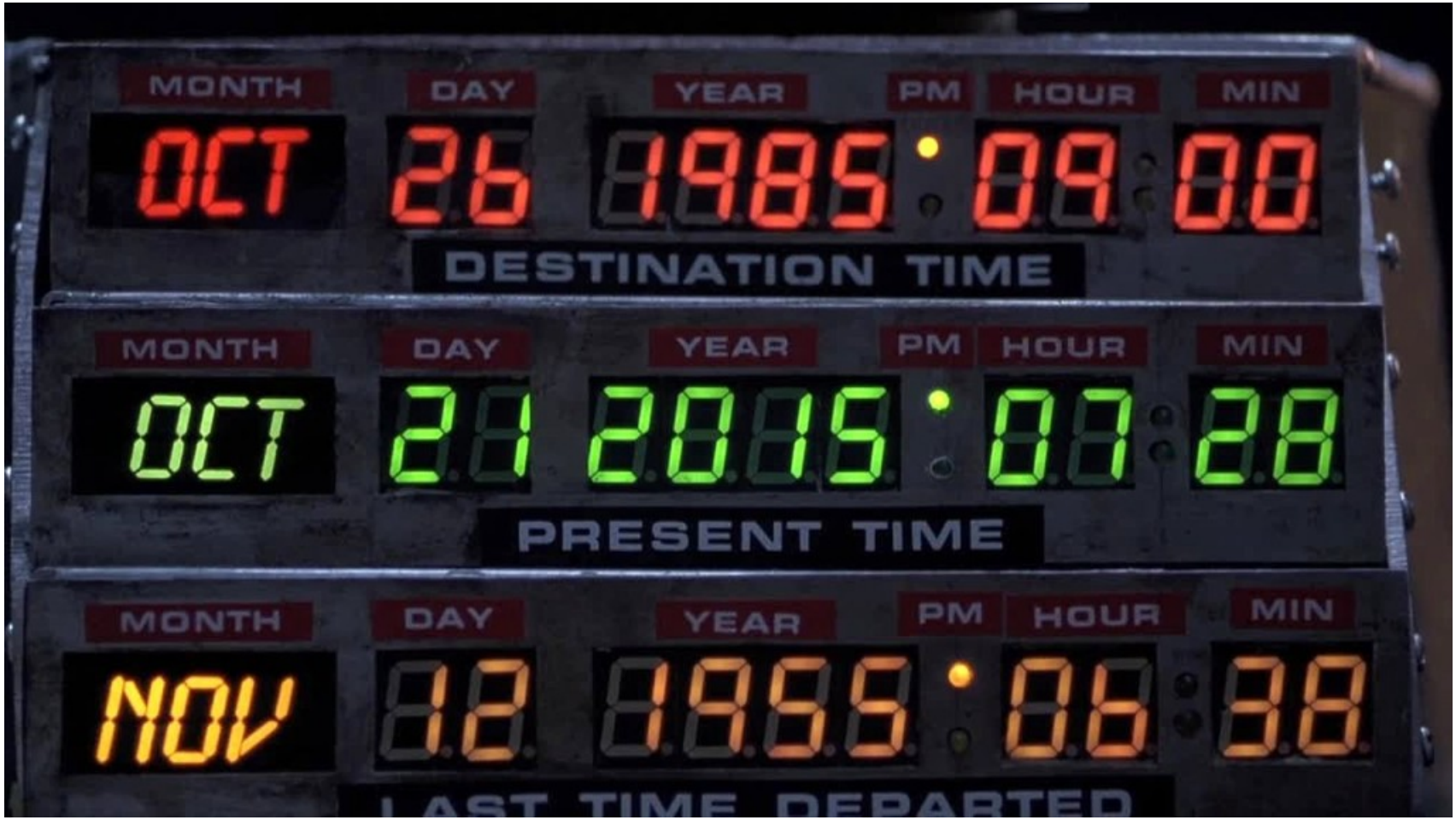
For the rest of this talk, “K3 surface” means a smooth quartic surface in $\mathbf{P}^3$.

Zeta function has the form

$$Z_p(T) = \frac{1}{(1-T)(1-pT)(1-p^2T)C_p(T)}$$

where $C_p \in \mathbf{Z}[T]$, $\deg C_p = 21$.

Roots of $C_p(T)$ in $\mathbf{C}$ all have absolute value $1/p$.



Tomorrow is the big day...

No-one will mind if I slip in a few flux capacitor jokes

What is the state of the art for computing $C_p(T)$ for a **single** prime p?

The fastest implementation I know is Edgar Costa's C++ code:

- Start with **classic AKR** algorithm (Abbott–Kedlaya–Roe, 2006),
- add "sparse Frobenius expansion" (H. 2007),
- add "controlled reduction" (H. 2010, unpublished),
- and some more improvements (Costa's thesis, 2015).

I will call this algorithm **linear AKR**.

Time complexity is essentially $O(p)$.

Space complexity is $O(\log p)$ — in practice this is constant.

Costa can compute:

- $C_p(T)$ for $p \sim 2^{16}$ in about **2.8 CPU hours**.
- $C_p(T)$ for all $p < 2^{16}$ in about **9,000 CPU hours** (about 1 year).
- Memory footprint $\sim$ 300 MB (could surely be improved).

Example from Costa's thesis, for $p = 1{,}048{,}583$:

$$\begin{aligned} C_p(T/p) = \frac{1}{p}\big(&pt^{21} + 160408t^{20} - 363853t^{19} + 94073t^{18} \\ &- 640447t^{17} + 29941t^{16} - 575731t^{15} - 347906t^{14} + 482949t^{13} \\ &- 503777t^{12} + 615760t^{11} + 615760t^{10} - 503777t^{9} + 482949t^{8} \\ &- 347906t^{7} - 575731t^{6} + 29941t^{5} - 640447t^{4} + 94073t^{3} \\ &- 363853t^{2} + 160408t + p\big). \end{aligned}$$

Question: can we beat Costa's code?

For large p, the only currently viable alternative to AKR framework is Lauder's "deformation method" (2004).

Optimised implementation by Pancratz–Tuitman (2014).

Time complexity $p^{1+o(1)}$.

Space complexity **linear**. For $p \sim 2^{16}$, probably about 200 GB.

For $p \sim 2^{11}$, about 5 times slower than Costa (on very sparse input).

Maybe this can be improved — I am not an expert.

For the rest of the talk I will stay in the AKR framework.

A crash course on AKR

Let $F \in \mathbf{Z}[x_0, x_1, x_2, x_3]$, homogeneous, degree 4.

We work with a space of **differentials** of the form

$$\left(\frac{A_1}{F} + \frac{A_2}{F^2} + \cdots \right) \Omega,$$

where

- Ω is a certain fixed 3-form,
- $A_k \in \mathbf{Q}_p[x_0, \ldots, x_3]$ homogeneous of degree $4k - 4$,
- A_k approaches zero p-adically quickly enough as $k \to \infty$.

(really... these are differentials over the weak completion of the coordinate ring of the complement in $\mathbf{P}^3$ of a lift to characteristic zero of the surface of interest which was originally defined over a finite field... but anyway.)

Let V be the quotient of the above differentials by the **relations**

$$\frac{\partial_i G}{F^m}\Omega \sim \frac{1}{m-1}\frac{(\partial_i F)G}{F^{m-1}}\Omega$$

for $G \in \mathbf{Q}_p[x_0, \ldots, x_3]$ and $i = 0, 1, 2, 3$.

(really... this V is a certain Monsky–Washnitzer cohomology space.)

One can explicitly find monomials $x^{u_1}, \ldots, x^{u_{21}}$ so that

$$\frac{x^{u_1}}{F}\Omega, \qquad \frac{x^{u_2}}{F^2}\Omega, \ldots, \frac{x^{u_{20}}}{F^2}\Omega, \qquad \frac{x^{u_{21}}}{F^3}\Omega$$

forms a **basis** for V.

There is a **reduction algorithm**: it takes as input a differential

$$\omega = \left(\frac{A_1}{F} + \cdots + \frac{A_n}{F^n}\right)\Omega,$$

and uses the relations to write ω as a linear combination of basis elements.

Finally, there is also a **Frobenius** map $\sigma : V \to V$.

Classic AKR:

1) For each ω in the basis, compute an approximation

$$\sigma(\omega) \approx \left(\frac{A_1}{F} + \frac{A_2}{F^2} + \cdots + \frac{A_n}{F^n} \right) \Omega.$$

   Must choose n big enough to ensure enough p-adic precision in step (3).
2) Run the reduction algorithm to write each $\sigma(\omega)$ in terms of the basis.
3) This yields a "matrix of Frobenius". Its characteristic polynomial is $C_p(T)$ (up to some scaling).

Can work modulo p^4 throughout.

“It works!! I finally invent something that works!”

Modifications to achieve to linear AKR

Problem #1: the approximations for $\sigma(\omega)$ are horrible dense polynomials!

About p^4 terms altogether.

Solution: use **sparse Frobenius expansion** to get an approximation

$$\sigma(\omega) \approx \left(\frac{A_p}{F^p} + \frac{A_{2p}}{F^{2p}} + \frac{A_{3p}}{F^{3p}} + \frac{A_{4p}}{F^{4p}} \right) \Omega$$

where the A_{kp} are polynomials in $x_0^p, \ldots, x_3^p$ (more or less).

Number of terms does not depend on p.

Problem #2: the standard reduction algorithm does not preserve sparsity!

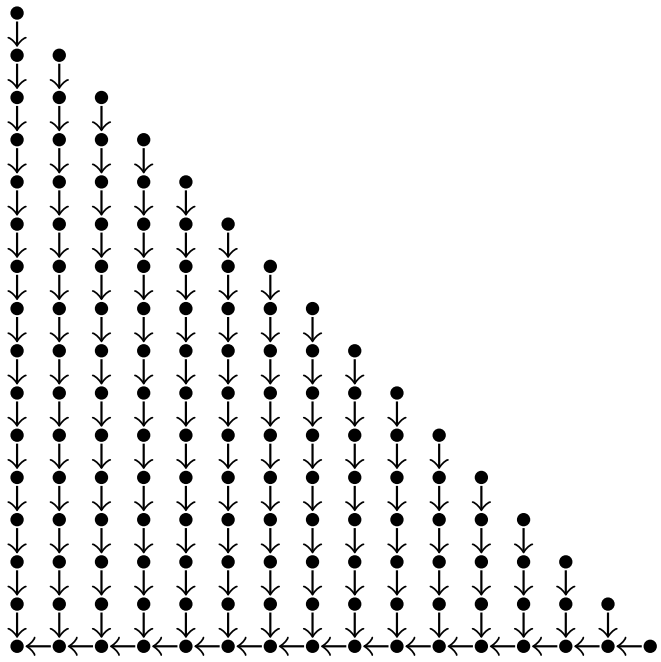
Solution: use **controlled reduction**.

Example: if $\deg G = 9$ one can reduce

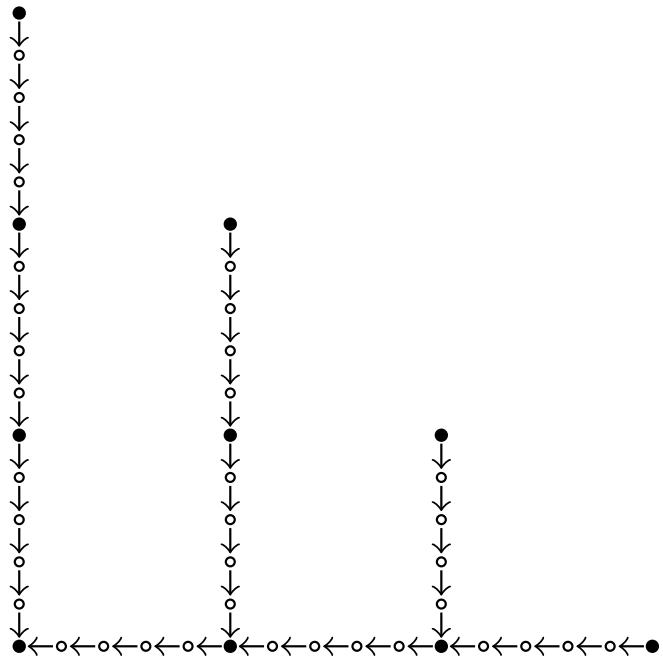
$$\frac{x^\alpha G}{F^m}\Omega \qquad \Longrightarrow \qquad \frac{x^\alpha G'}{(x_0 \cdots x_3)F^{m-1}}\Omega$$

where G' also has degree 9.

This preserves sparsity.



classic AKR



AKR with sparse Frobenius and controlled reduction

Problem #3: there are too many monomials of degree 9!

In fact, 220 of them.

Each reduction step involves a matrix-vector multiplication of size 220.

(Costa's code spends almost all of its time performing these multiplications.)

Solution: use relations to extract some redundancy.

Can reduce 220 to 64.

Yields a speedup by a factor of almost 12.

The resulting algorithm is what I call **linear AKR**.

From this slide onwards, all running times are **guesstimates**, i.e.,

“If my calculations are correct, when this baby hits eighty-eight miles per hour, you’re gonna to see some serious shit”

First improvement — interpolation

Problem: in the sparse expansion

$$\sigma(\omega) \approx \left(\frac{A_p}{F^p} + \frac{A_{2p}}{F^{2p}} + \frac{A_{3p}}{F^{3p}} + \frac{A_{4p}}{F^{4p}} \right) \Omega$$

there are still too many terms — in fact 1624 terms.

Costa needs to perform p reduction steps for each term.

And he needs to do that once for each of the 21 basis differentials.

Consider the composition of p successive reduction steps:

$$\frac{x^{\alpha p} G}{F^m}\Omega \qquad \Longrightarrow \qquad \frac{x^{\alpha p} G'}{(x_0^p \cdots x_3^p) F^{m-p}}\Omega$$

Let

$$T(\alpha_0 p, \alpha_1 p, \alpha_2 p, \alpha_3 p) \in M_{64}(\mathbf{Z}/p^4\mathbf{Z})$$

be the corresponding matrix that sends G to G'.

Observation: T is ***p*-adically analytic** in $\alpha_0 p, \ldots, \alpha_3 p$.

Instead of computing $T(\alpha_0 p, \ldots, \alpha_3 p)$ for 1624 tuples, we could compute it for 69 tuples and interpolate to get the rest.

($69 = 35 + 20 + 10 + 4$*; this includes all reduction "directions"*)

Time estimate?

- We *gain* a factor of $1624/69 \approx 24$.
- We also *gain* a factor of 21, because the hard work can be shared among all basis differentials.
- We *lose* a factor of 64, because we need matrix-matrix products instead of matrix-vector products.

Time estimate?

- We *gain* a factor of $1624/69 \approx 24$.
- We also *gain* a factor of 21, because the hard work can be shared among all basis differentials.
- We *lose* a factor of 64, because we need matrix-matrix products instead of matrix-vector products.

Conclusion: for $p \sim 2^{16}$, reduce 2.8 hours $\Rightarrow$ **22 minutes**.

For all $p < 2^{16}$, reduce from 9000 hours $\Rightarrow$ **1200 hours** (about 7 weeks).

Second improvement — square root trick

Modern version is Bostan–Gaudry–Schost (2004).

Can compute matrix product $M(1) \cdots M(p)$ in $p^{1/2+o(1)}$ time instead of naive $O(p)$ time.

Depends heavily on fast polynomial arithmetic (FFTs).

We apply this to computing the $T(\alpha_0 p, \ldots, \alpha_3 p)$.

So what is my time estimate for a single $p \sim 2^{16}$?

Drum roll....

So what is my time estimate for a single $p \sim 2^{16}$?

Drum roll....

About **2 minutes**.

To obtain this number, I estimated the number of FFTs, matrix multiplies, etc, of various sizes, and timed these building blocks using real life code.

But hang on... in 2010, I gave a few talks about the same sort of algorithm.

I even demonstrated an implementation!

It didn't run nearly that fast!

Does anyone remember the running time?

Let's go back to 2010 and find out...

Computational examples

Random degree 4 in $\mathbf{P}^3$ (K3 surfaces) over a prime field.

$\deg P(T) = 21$

Used $N = 2$ (ok provided that p is not too 'small').

p	cores	wall time
1009	12	3.4h
10007	12	7.7h
100003	12	18.4h
1000003	6	121h

19th October 2010

ECC 2010, Microsoft Research

Computational examples

Random degree 4 in $\mathbf{P}^3$ (K3 surfaces) over a prime field.

$\deg P(T) = 21$

Used $N = 2$ (ok provided that p is not too 'small').

p	cores	wall time
1009	12	3.4h
10007	12	7.7h
100003	12	18.4h
1000003	6	121h

Problem: in 2010 everything was metric.

I want to compare to $p \sim 2^{16}$.

We can fix that...

Computational examples

Random degree 4 in $\mathbf{P}^3$ (K3 surfaces) over a prime field.

$\deg P(T) = 21$

Used $N = 2$ (ok provided that p is not too 'small').

p	cores	wall time
1009	12	3.4h
10007	12	7.7h
65537	12	15.7h
100003	12	18.4h
1000003	6	121h

In 2010 it took **8 days** to handle $p \sim 2^{16}$!

So which is it: 2 minutes or 8 days?

What is the difference between 2010 and 2015?

Two main algorithmic differences:

- Matrix size: 64 vs 220.
- Interpolation trick.

If we scale the performance to account for these, we get about **15 minutes**.

Now the discrepancy is only a factor of 7.

Easily explained by Sage overhead, hardware improvements.

I think 2 minutes is closer to the truth.

For all $p < 2^{16}$ this yields **140 hours** — about 6 days.

Third improvement — average polynomial time

In theory it is now straightforward to convert this to an average polynomial algorithm.

Recall that

$$T(\alpha_0 p, \alpha_1 p, \alpha_2 p, \alpha_3 p) \in M_{64}(\mathbf{Z}/p^4\mathbf{Z})$$

is the matrix that executes p reduction steps, starting at position x^α.

The idea is to work over $\mathbf{Q}$ instead of $\mathbf{Q}_p$, and replace every $\alpha_i p$ with a formal variable P_i:

$$T(P_0, P_1, P_2, P_3) \in M_{64}(\mathbf{Z}[P_0, P_1, P_2, P_3]/(P_0, P_1, P_2, P_3)^4).$$

Then the machinery of the “accumulating remainder tree” does the rest.

Problem: the coefficients of $T(P_0, P_1, P_2, P_3)$ are huge integers!

We didn't see this when working one p at a time, because we always reduced modulo p^4.

But the average polynomial time algorithm has to work "globally".

Why are they so huge? When we reduced

$$\frac{x^\alpha G}{F^m}\Omega \qquad \Longrightarrow \qquad \frac{x^\alpha G'}{(x_0 \cdots x_3)F^{m-1}}\Omega,$$

we had to write G as a linear combination of the $\partial_i F$.

This involves solving a system of equations over $\mathbf{Z}$.

So the problem is coefficient growth in the inverse (or RREF) of a matrix.

I do not know how to ameliorate the coefficient growth completely.

Partial solution: use **sideways reduction**.

Instead of reducing like this:

$$\frac{x^\alpha G}{F^m}\Omega \qquad \Longrightarrow \qquad \frac{x^\alpha G'}{(x_0 \cdots x_3)F^{m-1}}\Omega,$$

do it like this:

$$\frac{x^\alpha G}{F^m}\Omega \qquad \Longrightarrow \qquad \frac{x_0 x^\alpha G'}{x_1 F^m}\Omega.$$

Notice the pole order doesn't change.

Instead we are working on reducing the exponent of x_1.

After x_1 is done, we work on x_2. Then x_3.

At the very end, reduce the pole order.

When is sideways reduction possible?

The algebra is not too hard, but will not fit on this slide.

It requires a **nondegeneracy condition** on some of the faces of F.

We still have to solve a system of equations, but it is much smaller.

Experiments suggest savings in coefficient size by a factor of about 8.

How long does it take?

Assume the coefficients of F randomly distributed in $[-4, 4]$.

(Note: running time is highly sensitive to the size of the coefficients.)

To handle all $p < 2^{16}$, my estimate is...

How long does it take?

Assume the coefficients of F randomly distributed in $[-4, 4]$.

(Note: running time is highly sensitive to the size of the coefficients.)

To handle all $p < 2^{16}$, my estimate is... **90 hours**.

Memory usage is perhaps **60 GB**.

Table 1 : Summary of complexity guesstimates (CPU hours)

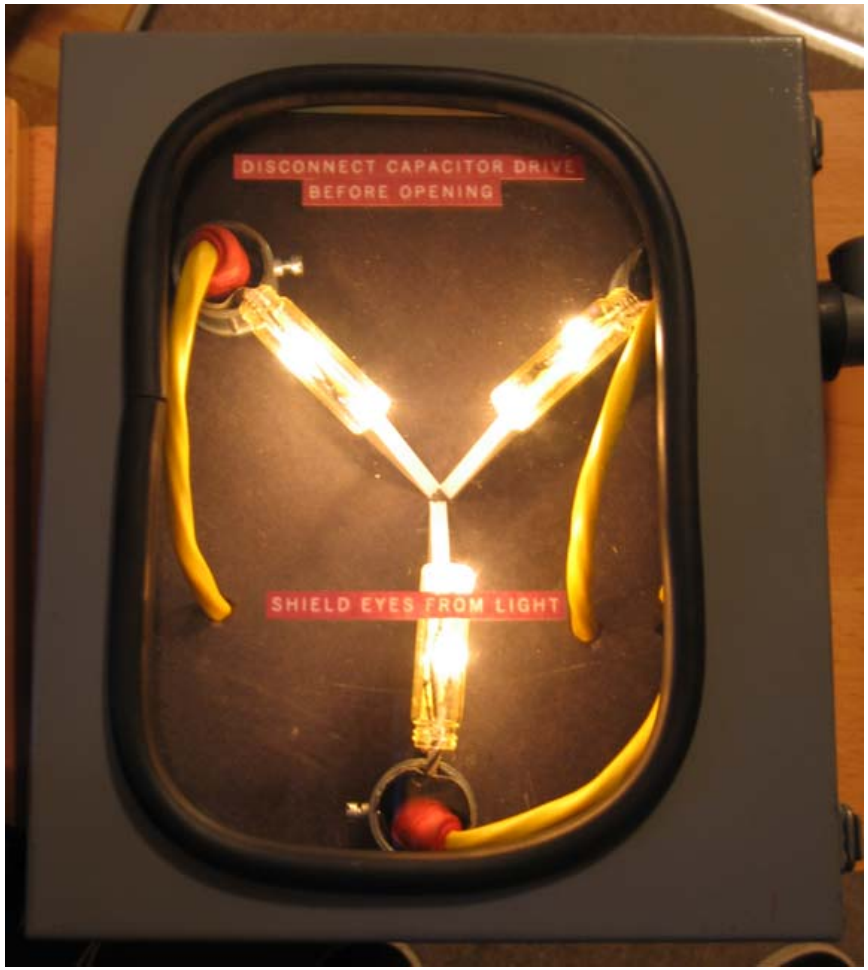
	$p < 2^{16}$	$p < 2^{20}$
Linear AKR (existing code)	9,000	2,300,000
Linear AKR + interpolation	1,200	310,000
Square root	140	9,000
Average polynomial	90†	2,000†
	(60 GB*)	(1 TB*)

†: not to be taken too seriously

*: reasonable time/space tradeoffs available

Conclusions:

- Square root and average polynomial are both highly feasible, and should be serious improvements over existing implementations.
- At this stage, cannot really tell if average polynomial time is worth the trouble.


DISCONNECT CAPACITOR DRIVE
BEFORE OPENING
SHIELD EYES FROM LIGHT

Thank you!