More on Fast Constant-Time GCD Computation and Modular Inversion

Daniel J. Bernstein Han-Ting Chen John R. Harrison
Gregory Maxwell Bow-Yaw Wang Pieter Wuille **Bo-Yin Yang**

Friday, December 27, 2024.

PQC Needing Inversions

NTRU Key generation (where n is prime)

- Find inverse in $\mathbb{Z}_3[X]/(X^n - 1)$
- Find inverse in $\mathbb{Z}_q[X]/(X^n - 1)$, which (for $q = 2^k$) depends on finding inverse in $\mathbb{Z}_2[X]/(X^n - 1)$.

NTRU Prime Key generation (where n is prime)

- Find inverse in $\mathbb{Z}_{4591}[X]/(X^n - X - 1)$ (= a field).
- Find inverse in $\mathbb{Z}_3[X]/(X^n - X - 1)$

Numerical Modular Inversions in CSIDH (similarly, SQIsign)

Needs inverse modulo $p = 4p_1 p_2 p_3 \cdots p_{73} p_{74} - 1$, where $p_1 \cdots p_{73}$ are the smallest 73 odd primes and $p_{74} = 587$.

Then and Now: Fast, Safe GCD and Inversions

Pre-2019: Fermat's Little Theorem: Compute $1/x$ in $\mathbf{F}_p$ as x^{p-2}.

$n^{3+o(1)}$ bit ops	using schoolbook multiplication
$n^{2.58...+o(1)}$ bit ops	using Karatsuba multiplication
$n^{2+o(1)}$ bit ops	using FFT-based multiplication

Post-2019: `safegcd` (or other constant time variations on Euclid's algorithm

$n^{2+o(1)}$ bit ops	using schoolbook multiplication
$n^{1.58...+o(1)}$ bit ops	using Karatsuba multiplication
$n^{1+o(1)}$ bit ops	using FFT-based multiplication

`safegcd`

`safegcd` is constant-time; $n^{1+o(1)}$ bit ops;

simpler than previous variable-time algorithms.

No division subroutine between recursive calls.

Cryptographic Constant-Time Algorithms

What is Constant-Time?

- No Conditional Branches depending on secrets
- No Variable Indices Memory Reads.
- Why? Otherwise cache-timing attacks leaks information.

A Vari-time Euclid-Stevins run in $\mathbb{Z}_7[X]$: see $R_4 \to R_5$, $R_5 \to R_6$

"Ideal" Euclidean step has deg dividend − deg divisor = deg divisor − deg remainder = 1.

$$
\begin{aligned}
R_0 &= 2y^7 + 7y^6 + y^5 + 8y^4 + 2y^3 + 8y^2 + y + 8 \\
R_1 &= 3y^6 + y^5 + 4y^4 + y^3 + 5y^2 + 9y + 2 \\
R_2 &= R_0 - (3y+6)R_1 = 4y^5 + 2y^4 + 2y^3 + 4y + 3 \\
R_3 &= R_1 - (6y+6)R_2 = y^4 + 3y^3 + 2y^2 + 2y + 5 \\
R_4 &= R_2 - (4y+4)R_3 = 3y^3 + 5y^2 + 4y + 4 \\
R_5 &= R_3 - (5y+2)R_4 = 2y + 4 \\
R_6 &= R_4 - (5y^2 + 3y + 3)R_5 = 6 \\
R_7 &= R_5 - (5y+3)R_6 = 0
\end{aligned}
$$

#Subtractions = #Coeffs. – 1 – #Skips

15 coefficients to start, 1 to end = 14 steps?

$$
\begin{aligned}
R_0 &= 2y^7 + 7y^6 + y^5 + 8y^4 + 2y^3 + 8y^2 + y + 8 \\
R_1 &= 3y^6 + y^5 + 4y^4 + y^3 + 5y^2 + 9y + 2 \\
& R_0 - 3yR_1 = 4y^6 + 3y^5 + 5y^4 + y^3 + 2y^2 + 2y + 1 \\
R_2 &= R_0 - (3y+6)R_1 = 4y^5 + 2y^4 + 2y^3 + 4y + 3 \\
& R_1 - 6yR_2 = 3y^5 + 6y^4 + y^3 + 2y^2 + 5y + 2 \\
R_3 &= R_1 - (6y+6)R_2 = y^4 + 3y^3 + 2y^2 + 2y + 5 \\
& R_2 - 4yR_3 = 4y^4 + y^3 + 6y^2 + 5y + 3 \\
R_4 &= R_2 - (4y+4)R_3 = 3y^3 + 5y^2 + 4y + 4 \\
& R_3 - 5yR_4 = 6y^3 + 3y^2 + 3y + 5 \\
R_5 &= R_3 - (5y+2)R_4 = 2y + 4 \\
& R_4 - 5y^2R_5 = 6y^2 + 4y + 4 \\
& R_4 - (5y^2 + 3y)R_5 = 6y + 4 \\
R_6 &= R_4 - (5y^2 + 3y + 3)R_5 = 6 \\
& R_5 - 5yR_6 = 4 \\
R_7 &= R_5 - (5y+3)R_6 = 0
\end{aligned}
$$

The Subtraction Stage in `safegcd`

To Start

- Reverse polynomials, start bigger poly as "Divisor" to ensure lead term $\neq 0$!
- Track the degree difference $\delta = \deg \text{Divisor} - \deg \text{Dividend}$.

Our Subtraction Stage: "divstep"

- Iff δ positive, *and* Dividend lead (constant) term $\neq 0$, then Swap, negate δ.
- Take linear combination of Divisor and Dividend to eliminate lead term.
- Divide by x (shift the array) and increment δ.

From "Divisor" $f = x^d R_0(1/x)$**, "Dividend"** $g = x^{d-1} R_1(1/x)$**, "Degree Diff"** $\delta = 1$
divstep : $\mathbb{Z} \times k[[x]]^* \times k[[x]] \to \mathbb{Z} \times k[[x]]^* \times k[[x]]$, divstep$(\delta, f, g) \mapsto$

$$\begin{cases} (1-\delta, g, (g(0)f - f(0)g)/x) & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ (1+\delta, f, (f(0)g - g(0)f)/x) & \text{otherwise.} \end{cases}$$

n	δ_n	f_n x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	...	g_n x^0	x^1	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	...
0	1	2	0	1	1	2	1	1	1	0	0	...	3	1	4	1	5	2	2	0	0	0	...
1	0	3	1	4	1	5	2	2	0	0	0	...	5	2	1	3	6	6	3	0	0	0	...
2	1	3	1	4	1	5	2	2	0	0	0	...	1	4	4	0	1	6	0	0	0	0	...
3	0	1	4	4	0	1	6	0	0	0	0	...	3	6	1	2	5	2	0	0	0	0	...
4	1	1	4	4	0	1	6	0	0	0	0	...	1	3	2	2	5	0	0	0	0	0	...
5	0	1	3	2	2	5	0	0	0	0	0	...	1	2	5	3	6	0	0	0	0	0	...
6	1	1	3	2	2	5	0	0	0	0	0	...	6	3	1	1	0	0	0	0	0	0	...
7	0	6	3	1	1	0	0	0	0	0	0	...	1	4	4	2	0	0	0	0	0	0	...
8	1	6	3	1	1	0	0	0	0	0	0	...	0	2	4	0	0	0	0	0	0	0	...
9	2	6	3	1	1	0	0	0	0	0	0	...	5	3	0	0	0	0	0	0	0	0	...
10	−1	5	3	0	0	0	0	0	0	0	0	...	4	5	5	0	0	0	0	0	0	0	...
11	0	5	3	0	0	0	0	0	0	0	0	...	6	4	0	0	0	0	0	0	0	0	...
12	1	5	3	0	0	0	0	0	0	0	0	...	2	0	0	0	0	0	0	0	0	0	...
13	0	2	0	0	0	0	0	0	0	0	0	...	6	0	0	0	0	0	0	0	0	0	...
14	1	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
15	2	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
16	3	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
17	4	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
18	5	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
19	6	2	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋱

Table 1: Iterates $(\delta_n, f_n, g_n) = \text{divstep}^n(\delta, f, g)$ for $k = \mathbf{F}_7$, $\delta = 1$, $f = 2 + 7x + 1x^2 + 8x^3 + 2x^4 + 8x^5 + 1x^6 + 8x^7$, and $g = 3 + 1x + 4x^2 + 1x^3 + 5x^4 + 9x^5 + 2x^6$. Line 8 with a leading 0 is the irregular remainder R_5; 9–12 are the irregular division $R_5 \to R_6$; two divsteps with $\delta = 0, 1$ usually represents a regular division.

What is special about divstep?

- divstep : $\binom{f}{g} \mapsto T(\delta, f, g)\binom{f}{g}$, $T = \begin{pmatrix} 1 & 0 \\ \frac{-g(0)}{x} & \frac{f(0)}{x} \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ \frac{g(0)}{x} & \frac{-f(0)}{x} \end{pmatrix}$. if $\delta > 0$, $g(0) \neq 0$.
- Can compute transition matrix of divstep^n from bottom n f, g coefficients.
- n divsteps only takes constant time $O(n \log^{2+O(1)} n)$, and data flow is regular

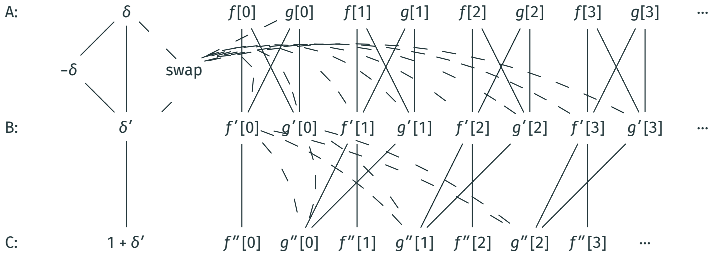


Figure 1: Data flow in an x-adic division step decomposed as conditional swap A to B and elimination B to C. The swap bit is set if $\delta > 0$ and $g[0] \neq 0$. The g outputs are $f'[0]g'[1] - g'[0]f'[1]$, $f'[0]g'[2] - g'[0]f'[2]$, $f'[0]g'[3] - g'[0]f'[3]$, etc.

Time-Constant divstep

- first half $(\delta, f, g) \to (\delta', f', g')$,

$$
\begin{aligned}
swap &= \begin{cases} -1 & \text{if } \delta > 0 \text{ and } g(0) \neq 0, \\ 0 & \text{otherwise.} \end{cases} \\
mask &= (f \text{ xor } g) \text{ and } swap \\
f' &= f \text{ xor } mask \\
g' &= g \text{ xor } mask \\
\delta' &= \delta \text{ xor } ((\delta \text{ xor } -\delta) \text{ and } swap)
\end{aligned}
$$

(equivalent vector instructions are available).

- second half:

$$(\delta, f, g) \to (1 + \delta, f, (f(0)g - g(0)f)/x).$$

Main Theorem (for Polynomials)

Let k be a field. Let d be a positive integer. Let R_0, R_1 be elements of the polynomial ring $k[x]$ with $\deg R_0 = d > \deg R_1$. Define $G = \gcd\{R_0, R_1\}$, and let V be the unique polynomial of degree $< d - \deg G$ such that $VR_1 \equiv G \pmod{R_0}$. Define $f = x^d R_0(1/x)$; $g = x^{d-1} R_1(1/x)$; $(\delta_n, f_n, g_n) = \text{divstep}^n(1, f, g)$; $T_n = T(\delta_n, f_n, g_n)$; and $\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} = T_{n-1} \cdots T_0$. Then

$$\begin{aligned} \deg G &= \delta_{2d-1}/2; \\ G &= x^{\deg G} f_{2d-1}(1/x)/f_{2d-1}(0); \\ V &= x^{-d+1+\deg G} v_{2d-1}(1/x)/f_{2d-1}(0). \end{aligned}$$

Jumping Through divsteps

Sub-Quadratic GCD and Inversions

To compute (δ_n, f_n, g_n) and transition matrix $T_{n-1} \cdots T_0$:

- If $n \le 1$, use the definition of divstep and stop.
- Choose $j \in \{1, 2, \ldots, n-1\}$.
- Jump j steps from δ, f, g to δ_j, f_j, g_j. Specifically, using only the first j coefficients, compute the j-step transition matrix $T_{j-1} \cdots T_0$, and then multiply into $\begin{pmatrix} f \\ g \end{pmatrix}$ to obtain $\begin{pmatrix} f_j \\ g_j \end{pmatrix}$.
- Similarly jump $n-j$ steps from δ_j, f_j, g_j to δ_n, f_n, g_n.

So an (n, t) problem (n steps, t terms) becomes a (j, j) problem plus an $(n-j, n-j)$ problem, plus $O(1)$ polynomial multiplications with $O(t+n)$ coefficients.

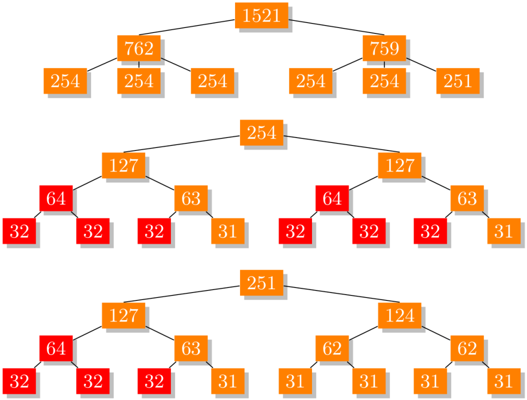
Jumping divsteps for divstep$^n(\delta, f, g)$

```
from divstepsx import divstepsx

def jumpdivstepsx(n,t,delta,f,g):
  assert t >= n and n >= 0
  kx = f.truncate(t).parent()

  if n <= 1: return divstepsx(n,t,delta,f,g)

  j = n//2

  delta,f1,g1,P1 = jumpdivstepsx(j,j,delta,f,g)
  f,g = P1*vector((f,g))
  f,g = kx(f).truncate(t-j),kx(g).truncate(t-j)

  delta,f2,g2,P2 = jumpdivstepsx(n-j,n-j,delta,f,g)
  f,g = P2*vector((f,g))
  f,g = kx(f).truncate(t-n+1),kx(g).truncate(t-n)

  return delta,f,g,P2*P1
```

Figure 2: Algorithm `jumpdivstepsx`, Same inputs and outputs.

How to Invert $R_1(x)$ in $\mathbf{Z}_{4591}[x]/(x^{761} - x - 1)$ today

1. Set $f = 1 - x^{760} - x^{761}$, $g = x^{760} R_1(1/x)$.



1.1 sheared jumpdivsteps track polynomials u_n, v_n, q_n, r_n scaled by $x^{n-1}, x^{n-1}, x^n, x^n$.

1.2 unsaturated jumpsteps has u_n, v_n, q_n, r_n all scaled by x^n

1.3 recursively split divstep^{1521} using *sheared* + *unsaturated*.

1.4 From unsaturated 7 and sheared 8 divsteps use jumps to assemble divstep^{1521} result

2. Then $R_1^{-1} = x^{-760} v(1/x)/f_{1521}(0)$

Why sheared and unsaturated, and multiplications in $\mathbb{Z}_{4591}[x]/(x^{761} - x - 1)$

Computing with sheared $\begin{bmatrix} u'/x & v'/x \\ q' & r' \end{bmatrix}$ (because unsaturated is easy)

1. $\begin{bmatrix} f'/x \\ g' \end{bmatrix} = x^{-n} \times \begin{bmatrix} u/x & v/x \\ q & r \end{bmatrix} \times \begin{bmatrix} f \\ g \end{bmatrix}$.
2. Multiply f'/x by x.

1. $\begin{bmatrix} \frac{u'}{x} & \frac{v'}{x} \\ q' & r' \end{bmatrix} = \begin{bmatrix} \frac{u_2}{x} & \frac{v_2}{x} \\ q_2 & r_2 \end{bmatrix} \times \begin{bmatrix} \frac{u_1}{x} \cdot x = u_1 & \frac{v_1}{x} \cdot x = v_1 \\ q_1 & r_1 \end{bmatrix}$.
2. Multiply $\frac{u'}{x}, \frac{v'}{x}$ by x for unsaturated result.

Small polymals = Karatsuba: 8×8 **(or** 8×7**),** 16×16 **(or** 16×15**),** 32×32 **(or** 32×31**)**

Big polymuls = T(runcated)Rader-17: 64×64, 128×128 **(and slightly smaller)**

Biggest Polymuls = TRader-17 + Good-3: 256×256, 256×512, 768×768 **(!!)**

Table 2: Cycle counts for key generation in **sntrup761**, currently being verified

sntrup761	Cortex-A53	Cortex-A72	Cortex-A76	M1
`ref` from supercop	33,504,035	23,837,956	16,958,229	13,449,469
divstep [eprint:2023/1580]	6,547,768	5,517,692	3,047,956	1,051,392
jump divstep	2,569,555	1,969,656	1,429,813	471,571
jump divstep/`ref`	7.66%	8.26%	8.43%	3.5%
jump divstep	39.24%	**35.69%**	46.91%	**44.85%**

Radix-2 divstep for Integers case (Bernstein-Yang 2019): no top-down version

$$\text{divstep on } \mathbf{Z} \times \mathbf{Z}_2^* \times \mathbf{Z}_2 : (\delta, f, g) \mapsto \begin{cases} (1-\delta, g, (g-f)/2), & \text{if } \delta > 0 \text{ and } g \text{ is odd} \\ (1+\delta, f, (g + (g \bmod 2) f)/2), & \text{otherwise.} \end{cases}$$

$$\text{divstep}\begin{pmatrix} f \\ g \end{pmatrix} = T\begin{pmatrix} f \\ g \end{pmatrix}, T(\delta, f, g) = \begin{pmatrix} 0 & 1 \\ -\frac{1}{2} & \frac{1}{2} \end{pmatrix} \text{if } \delta > 0 \text{ and } g \text{ is odd}, \begin{pmatrix} 1 & 0 \\ \frac{g \bmod 2}{2} & \frac{1}{2} \end{pmatrix} \text{otherwise}$$

2-adic divstep Split in Two Halves

- Conditional Swap: $(\delta, f, g) \to (-\delta, g, -f)$ iff $g \bmod 2 = 1$ and $\delta > 0$
- Eliminate: $\delta \to \delta + 1$, $g = (g + (g \bmod 2) f)/2)$.

Theorem (Thm 11.2, Bernstein-Yang 2019, in part via exhaustive search)

f (odd), g: int., $(\delta_n, f_n, g_n) := \text{divstep}^n(1, f, g)$; $T_n := T(\delta_n, f_n, g_n)$.] If $f^2 + 4g^2 \leq 5 \cdot 2^{2d}$, m: posint.; $m \geq \lfloor (49d + 80)/17 \rfloor$ if $d < 46$, and $m \geq \lfloor (49d + 57)/17 \rfloor$ if $d \geq 46$. Then if $\begin{pmatrix} u_n & v_n \\ q_n & r_n \end{pmatrix} := T_{n-1} \cdots T_0$, we have $g_m = 0$; $f_m = \pm \gcd\{f, g\}$; $2^{m-1} v_m \in \mathbf{Z}$; and $2^{m-1} v_m g \equiv 2^{m-1} f_m \pmod{f}$.

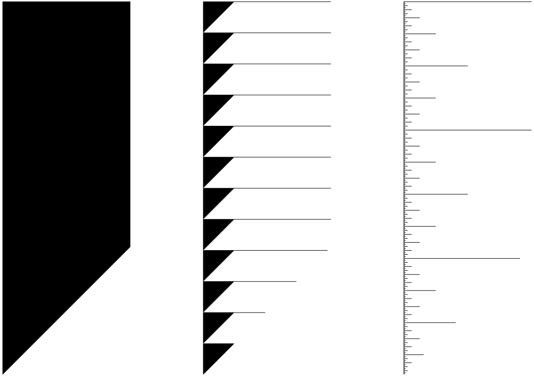
Invert a 255-bit x Modulo $p = 2^{255} - 19$ in 2019 (copying polynomial `safegcd`)

1. Set $f = p, g = x, \delta = 1, i = 1$.
2. Set $f_0 = f \pmod{2^{64}}, g_0 = g \pmod{2^{64}}$.
3. Compute $(\delta', f_1, g_1) = \mathrm{divstep}^{62}(\delta, f_0, g_0)$ and obtain a scaled transition matrix T_i s.t. $\frac{T_i}{2^{62}} \begin{pmatrix} f_0 \\ g_0 \end{pmatrix} = \begin{pmatrix} f_1 \\ g_1 \end{pmatrix}$. 63-bit signed entries of T_i fit into 64-bit registers. (`jump64divsteps2`)
4. Compute $(f, g) \leftarrow T_i(f, g)/2^{62}$. Set $\delta = \delta'$.
5. Set $i \leftarrow i + 1$. Go back to step 3 if $i \le 12$.
6. Compute $v \bmod p$ where v is top-right entry of $T_{12}T_{11} \cdots T_1$:
   - 6.1 Compute (126-bit signed integers) pair-products $T_{2i}T_{2i-1}$
   - 6.2 Compute (252-bit signed) 4-products $T_{4i}T_{4i-1}T_{4i-2}T_{4i-3}$
   - 6.3 Convert 4-products to unsigned ints modulo p (4×64-bit limbs).
   - 6.4 Compute final vector × matrix *times* vector modulo p.
7. Compute $x^{-1} = \mathrm{sgn}(f) \cdot v \cdot 2^{-744} \pmod{p}$ where 2^{-744} is precomputed.

Results on Intel CPUs, $p = 2^{255} - 19$

- 10050, 8778, and 8543 cycles on Haswell, Skylake, and Kaby Lake;
- Nath-Sarkar 2018: 11854, 9301, and 8971 cycles (resp.)

Jump Strategies (picture from 2019)



Left strategy: $j = 1$; i.e., computing each divstep in turn to full precision. Middle strategy: $j = 1$ for ≤62 requested iterations, else $j = 62$; i.e., using 62 bottom bits to compute 62 iterations, jumping 62 iterations, using 62 new bottom bits to compute 62 more iterations, etc. Right strategy: $j = 1$ for 2 requested iterations, $j = 2$ for 3 or 4 requested iterations, $j = 4$ for 5 or 6 or 7 or 8 requested iterations, etc.

Figure 3: Three jump strategies for 744 divstep iterations on 255-bit ints

Vertical axis, 0 on top through 744 on bottom: number of iterations. Horizontal axis (within each strategy), 0 on left through 254 on right: bit positions used in computation.

Questions and More Recent Results

Pornin eprint 2020/972

- report 7490 cycles on Intel Coffee Lake (~ Kaby Lake) via another algorithm.
- reported proof bug in ver.2020.08.23, and updates to 6253 Coffee Lake cycles
- “the algorithm, and the revised proof, are believed correct”

Obvious Questions

- Is [Bernstein-Yang 2019, Theorem 11.2] — which relies on a large exhaustive search computation — correct? Is there a simpler proof?
- how quickly can the resulting modular-inversion software run?
- Can the software, with many speed-induced complications, be correct?
- are divsteps are the best approach in the first place?

Questions and More Recent Results

Pornin eprint 2020/972

- report 7490 cycles on Intel Coffee Lake (~ Kaby Lake) via another algorithm.
- reported proof bug in ver.2020.08.23, and updates to 6253 Coffee Lake cycles
- "the algorithm, and the revised proof, are believed correct"

Obvious Questions

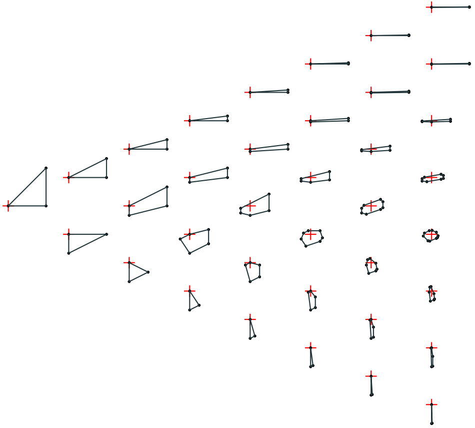
- Is [Bernstein-Yang 2019, Theorem 11.2] — which relies on a large exhaustive search computation — correct? Is there a simpler proof? Yes.
- how quickly can the resulting modular-inversion software run? See below.
- Can the software, with many speed-induced complications, be correct? Yes.
- are divsteps are the best approach in the first place? As far as we know.

Easier (and Better) Proof: Convex Hulls

Reducing 744 divsteps to 720

Consider all real $f > g > 0$

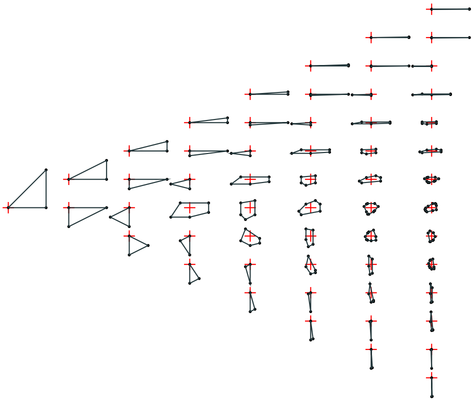
- Start $\{(1,0,0),(1,1,0),(1,1,1)\}$.
- Perforce, take all 3 branches
  1. $(\delta, f, g) \mapsto (1+\delta, f, g/2)$
  2. $(\delta, f, g) \mapsto (1+\delta, f, (f+g)/2)$
  3. $(\delta, f, g) \mapsto (1-\delta, g, (g-f)/2)$
- Brute-force shows that either coordinate goes below 2^{-255} after 720 (later 719) iterations.
- A stable 42-point hull for $\delta = 1$ shrinking by a constant factor every 14 iterations exist.



Even Better: Divsteps starting with $\delta = \frac{1}{2}$ (`hddivsteps`)

A stable 102-point hull for $\delta = \frac{1}{2}$ shrinking by $\frac{1591853137+3\sqrt{273548757304312537}}{2^{55}}$ every 54 iterations exist.

HOL Light Proof exists for 255-bit numbers and 588 `hddivsteps`



Consider all real $f > g > 0$

- Start $\{(\frac{1}{2}, 0, 0), (\frac{1}{2}, 1, 0), (\frac{1}{2}, 1, 1)\}$.
- Perforce, take all 3 branches
  1. $(\delta, f, g) \mapsto (1 + \delta, f, g/2)$
  2. $(\delta, f, g) \mapsto (1 + \delta, f, (f + g)/2)$
  3. $(\delta, f, g) \mapsto (1 - \delta, g, (g - f)/2)$
- Brute-force shows that either coordinate goes below 2^{-255} after 588 iterations.
- $2.304n$ `hddivsteps` suffices for n-bit numbers

Faster Assembly Language Routines for Single-Limb divsteps

Parallel Processing inside a single limb!

- Running divsteps requires us to evaluate two flags and update f, g, u, v, q, r.
- To do n iterations, we need the bottom n bits of f, g; $1 \geq u, v > -1$ and $1 > r, s > -1$ (renaming) are multiples of 2^{-n}, so we scale them by -2^n.
- We can merge (f, u, v) and (g, r, s) each into a 64-bit limb for 30 iterations with `fuv` $= 2^{33}(f \bmod 2^i) - 2^{i+31}u - 2^i v$ and `grs` $= 2^{33}(g \bmod 2^i) - 2^{i+31}r - 2^i s$
- Actually used (`fuv`, `grs`) $= (f, g) - 2^{i+42}(u, r) - 2^{21+i}(v, s)$ for 20 iterations.
- We completely unroll and the code cache gets trampled with > 20 iterations.

One iteration looks like this (inside `qhasm` code verified by Han-Ting Chen):

```
z = -1                                  grs -= fuv
oldg = grs                           (int64) grs >>= 1
h = grs + fuv                        (int64) h >>= 1
            =? grs & 1               m = -m
                                               signed<? z - 0
z = m    if !=                       fuv = oldg if !signed<
h = grs if  =                        grs = h    if  signed<
mnew = m + 1                         m = mnew   if  signed<
```

Parallelized BigInt Update Every 60 divsteps (also Verified by Han-Ting)

Limbs of F, V, G, S in a vector register, signed, radix 2^{30} in 64 bits, lazy reductions

- Use Montgomery modular arithmetic to compute $\begin{bmatrix} u & v \\ r & s \end{bmatrix} \begin{bmatrix} F & V \\ G & S \end{bmatrix}$
  - Input u, v, r, s in two 30-bit limbs.
  - Update F, V, G, S in 9 30-bit limbs.
  - add suitable multiple of the modulus to zero out bottom two limbs.
  - free division by 2^{60} every loop.
- `jumpdivsteps` inner loop strictly uses integer registers only
- bigint update uses vector registers only
- We can interleave the vector and integer arithmetic
  - Use Genetic algorithm to compute best interleaving (speed up: 20%)

Recent Developments

In `lib25519` on x86 after we interleave int and vector loops

- 25519: 5908 cycles (Haswell), 3880 cycles (Skylake)
- 256 bit arbitrary prime invmod: 6418 cycles (Haswell), 4028 cycles (Skylake)
- long arbitrary prime invmod on skylake:
  - 512-bit: 9091 cycles
  - 1024 bit: 21671 cycles
  - 2048 bit: 64053 cycles
- Key components verified in CryptoLine

John Harrison/Amazon`s2n` bignum library. Verified in HOL Light

- provide verified x86 code
- provide verified ARMv8 Neon code

Future Applications? (25519, Pairings, CSIDH, SQISign ...)

Questions?