Speeding-up Newton iteration using variants of polynomial multiplication

Ling Ding, Éric Schost

ORCCA, UWO

Context

Newton iteration

- computing symbolic solutions
- to polynomial / differential equations
- at high precision

Example Consider the equation, with coefficients in $\mathbb{Z}/101\mathbb{Z}$
[Bostan, Morain, Salvy, Schost, 08]

$$(x^6 + x^4 + 1)f'(x)^2 = 1 + 75f(x)^4 + 16f(x)^6, \quad f(0) = 0, \quad f'(0) = 1.$$

We want to find the first few terms of the power series solution

$$f = x + 68x^5 + 66x^7 + 60x^9 + 84x^{11} + \cdots .$$

Context

Newton iteration is fast

- $\mathsf{M}(n)$ denotes the cost of polynomial multiplication in degree n
- then, for most problems, $O(\mathsf{M}(n))$ to get n terms
- compared to (usually) $O(n^2)$

Objective: make it faster

- reducing the constant in the big-Oh
- using tricks such as short product (Mulders) or middle product (Hanrot, Quercia, Zimmermann)
- for moderate degrees

This talk

- first order differential equations

Related work

Newton for ODE's

- [Brent, Kung, 78]
  Focused on first-order equations.
- [Watt, 88]
  Recast differential equations as fixed point problems.
- [Hoeven, 02]
  Used a similar idea + fast "relaxed multiplication".
- [Bostan, Chyzak, Ollivier, Salvy, Schost, 07]
  Focused in particular on higher order equations.

Other contexts

- [Hanrot *et al.*, 04]
  middle product for inverse, square-root
- [Hanrot-Zimmermann, 04], [Bernstein, 04], [Bostan-Schost, 08]
  tricks for the FFT model

Motivation

Previous example from a point-counting algorithm in elliptic cryptology: computing a degree n morphism

$$\Phi : E \to E', \quad (x, y) \mapsto (\varphi(x), y\varphi'(x)).$$

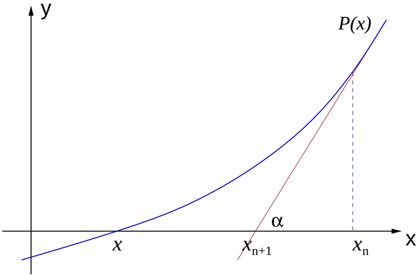
- not-so-naive algorithm $O(n^2)$
- Newton $O(\mathsf{M}(n))$

Newton wins for record-size computations (degree > 1000).

However, *"if we want the cryptologists to buy our stuff, we'd better be competitive in crypto size"*:

- small degree (about **300**).

Newton iteration for numerical root-finding



$$x_{n+1} = x_n - \frac{P(x_n)}{P'(x_n)}.$$

The number of correct digits approximately *doubles* at each iteration.

Newton iteration for ODE's

Given the equation $G(x,f,f') = 0$ and $f \mod x^n$, we want $f \mod x^{2n}$.

- evaluate

$$a = \frac{\partial G}{\partial u}(x,f,f'), \quad b = \frac{\partial G}{\partial t}(x,f,f'), \quad c = -G(x,f,f') \mod x^{2n}$$

- use *inverse* and *exponential* to compute

$$d = \frac{b}{a}, \quad e = \frac{c}{a}, \quad j = \exp(\textstyle\int d) \mod x^{2n},$$

we obtain

$$f = f + \frac{\int ej}{j} \mod x^{2n}.$$

Newton iteration for ODE's

Given the equation $G(x,f,f') = 0$ and $f \mod x^n$, we want $f \mod x^{2n}$.

- evaluate

$$a = \frac{\partial G}{\partial u}(x,f,f'), \quad b = \frac{\partial G}{\partial t}(x,f,f'), \quad c = -G(x,f,f') \mod x^{2n}$$

- use *inverse* and *exponential* to compute

$$d = \frac{b}{a}, \quad e = \frac{c}{a}, \quad j = \exp(\textstyle\int d) \mod x^{2n},$$

we obtain

$$f = f + \frac{\int ej}{j} \mod x^{2n}.$$

Power series inverse

Consider power series

$$f = \sum_{i \geq 0} f_i x^i \quad \text{and} \quad \widetilde{g} = \sum_{i \geq 0} g_i x^i$$

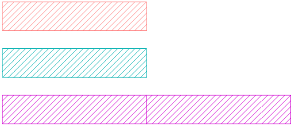
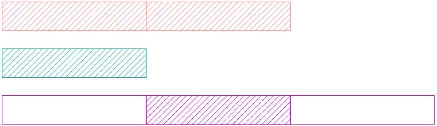
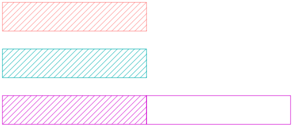
such that $f_0 = 1, \widetilde{g} = \frac{1}{f}$.

Newton iteration:

- suppose that we know $g = \widetilde{g} \bmod x^n$
- then we get $G = \widetilde{g} \bmod x^{2n}$ as

$$g(2 - fg) \mod x^{2n}$$

Various multiplications

Type	Lengths & Graph rep.
plain product $\mathsf{M}(n)$	A: (0,n) B: (0,n) C: (0,2n) 
middle product $\mathsf{M}(n) + O(n)$	A: (0,2n) B: (0,n) C: (n,2n) 
short product $\mathsf{m}(n)$	A: (0,n) B: (0,n) C: (0,n) 
⋮	⋮

Updating inverses

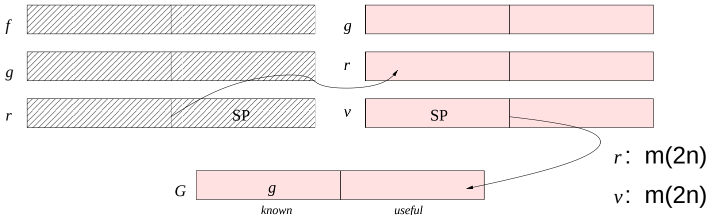


Figure: Naive inverse

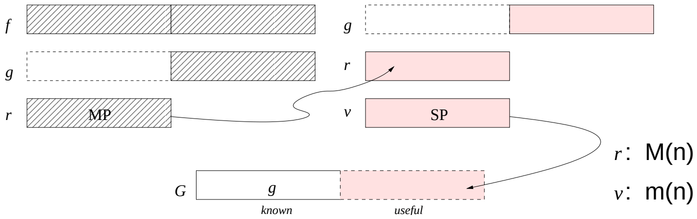
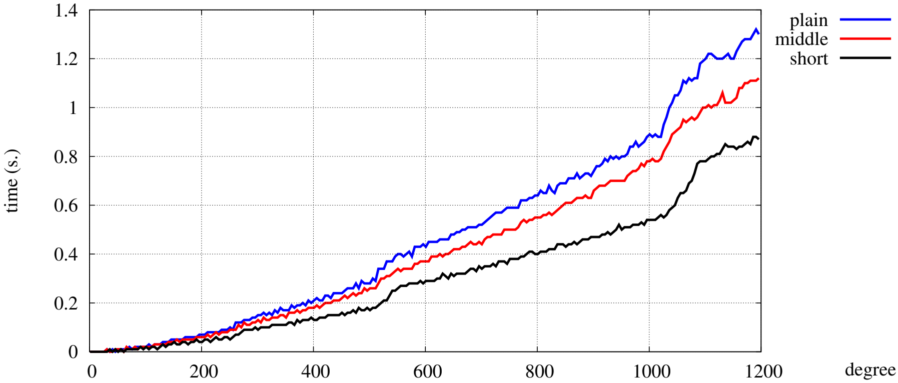
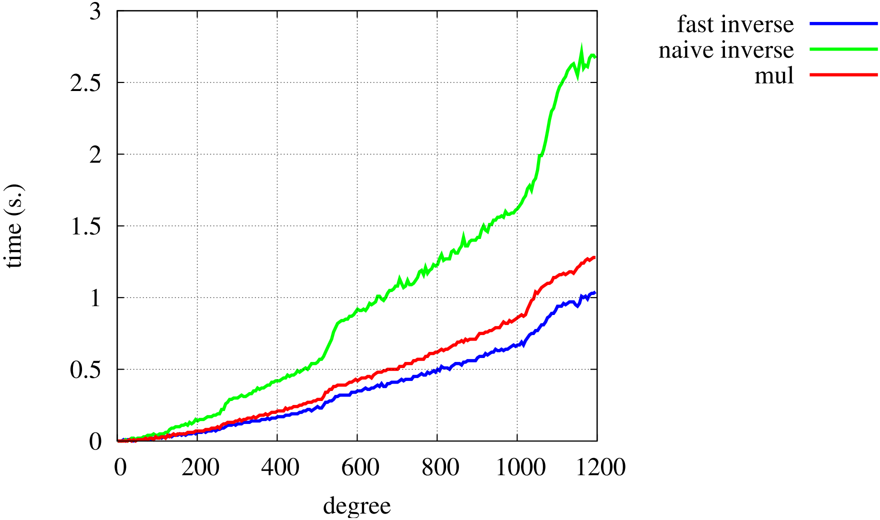


Figure: Updating inverse

Fast multiplication



Comparison between naive and fast inverse



Precision issues for evaluation

Recall: we need

$$a = \frac{\partial G}{\partial u}(x, f, f'), \quad b = \frac{\partial G}{\partial t}(x, f, f'), \quad c = -G(x, f, f') \mod x^{2n}$$

Objective: avoid computing useless quantities, as for the inverse

Starting points

- c starts with n zeros
- a and b needed only modulo x^n

Propagation

- length analysis: high-deg, low-deg.

$$A = a_0 + a_1 x + \cdots + a_n x^n + \cdots + a_{2n} x^{2n} + \cdots + a_i x^i$$

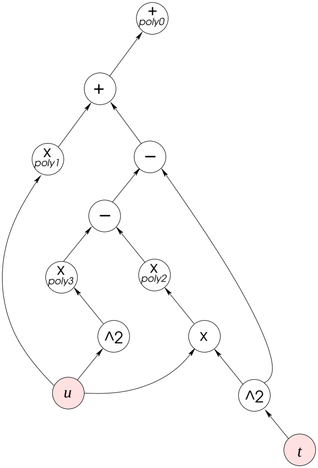
$(\textit{low-deg}, \textit{high-deg}) = (n, 2n)$

- apply variants of multiplications (middle, short product, ...)

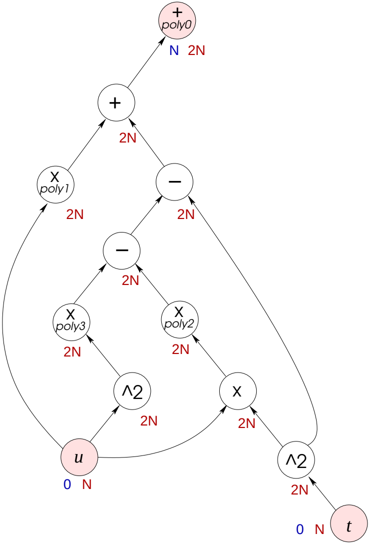
Example

We consider $G(x, f, f') = 0$, with

$$G(x,t,u) = (1+x+x^2)u^2 - (2+x)ut^2 - t^2 + 5u + 3.$$



Assigning high-degrees

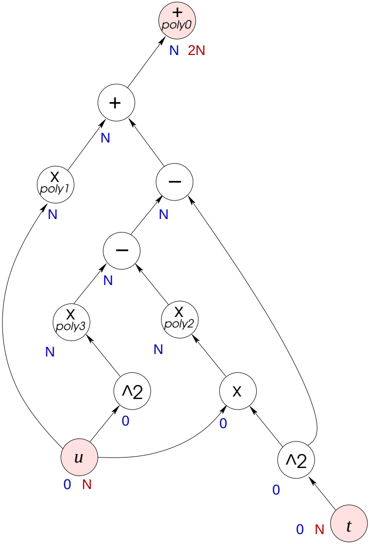


Assign "high−deg"

For "+", "−",
"high−deg" is max of the arguments.

For "x", "^",
"high−deg" is the same as the output.

Assigning low-degrees

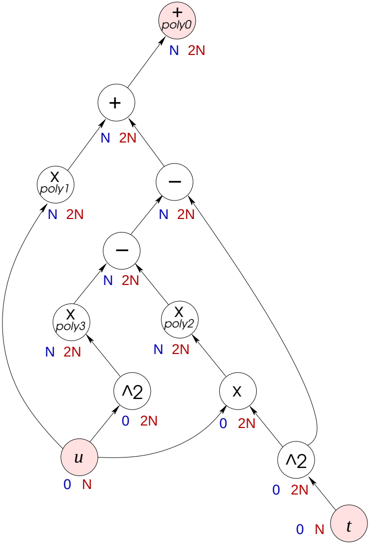


Assign "low−deg"

For "+", "−",
"low−deg"s are the same as the result.

For "x", "^",
"low−deg"s are set to "0"s.

Choosing which multiplication



Assign "high−deg"

For "+", "−",
"high−deg" is max of the arguments.

For "x", "^",
"high−deg" is the same as the output.

Assign "low−deg"

For "+", "−",
"low−deg"s are the same as the result.

For "x", "^",
"low−deg"s are set to "0"s.

Turning graph to code

Java code generator

- input: a DAG for G
- outputs C code

Main steps

- **Workspace allocation.**
  allocate the memory for temporary results.
- **Initialization.**
  initialize constants and polynomials given in the graph $\mathcal{G}$
- **Evaluation.**
  evaluate c by following the graph $\mathcal{G}$
  evaluate a, b by following the graph $\mathcal{G}'$

Output code overview

```
void G_unsigned_long(unsigned long * __restrict__ C,
                     unsigned long * __restrict__ A, unsigned long * __restrict__ B,
               const unsigned long * __restrict__ t, const unsigned long * __restrict__ u,
               const unsigned long p, const unsigned long ip, const unsigned long jp, int N){
/*------------ Workspace allocation ------------*/
unsigned long *wk=(unsigned long *) malloc(30*N*sizeof(unsigned long));

/*--------------- Initialization ---------------*/
unsigned long *poly0=(unsigned long *)malloc(1*sizeof(unsigned long));
unsigned long *poly0_pre=(unsigned long *)malloc(1*sizeof(unsigned long));
poly0[0]=3;
poly0_pre[0]=mulredcred(p, ip, jp, 3);
...
/*---------------- Evaluation C ---------------*/
mul_plain_unsigned_long(wk+N*0, u, u, p, ip,  N);
constant_mul_unsigned_long(wk+N*2, wk+N*0, poly3_pre, 3, 1*N, 2*N, 0*N, 2*N, p, ip);
...
constant_add_unsigned_long(C, wk+N*16, poly0, 1, 1*N, 2, 1*N, 2*N, p, ip);

/*---------------- Evaluation A ---------------*/
zero_unsigned_long(wk+N*18, p, ip, N);
...
sub_unsigned_long(A, wk+N*22, wk+N*23, p, ip, 0*N, 1*N);

/*---------------- Evaluation B ---------------*/
add_unsigned_long(wk+N*24, u, u, p, ip, 0*N, 1*N);
...
/*---------- Workspace, polys free ------------*/
  free(wk);
  free(poly0);
  free(poly0_pre);...}
```

Timings

$G(x, t, u) = (1 + x + x^2)u^2 - (2 + x)ut^2 - t^2 + 5u + 3.$