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

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

- $M(n)$ denotes the cost of polynomial multiplication in degree $n$
- then, for most problems, $O(M(n))$ 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

\[
\Phi : E \rightarrow E', \quad (x, y) \mapsto (\varphi(x), y\varphi'(x)).
\]

- not-so-naive algorithm \( O(n^2) \)
- Newton \( O(M(n)) \)

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

we obtain

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

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

we obtain

$$f = f + \frac{\int e j}{j} \mod x^{2n}.$$
Consider power series

\[ f = \sum_{i \geq 0} f_i x^i \quad \text{and} \quad \tilde{g} = \sum_{i \geq 0} g_i x^i \]

such that \( f_0 = 1, \tilde{g} = \frac{1}{f} \).

**Newton iteration:**

- suppose that we know \( g = \tilde{g} \mod x^n \)
- then we get \( G = \tilde{g} \mod x^{2n} \) as

\[ g(2 - fg) \mod x^{2n} \]
### Various multiplications

<table>
<thead>
<tr>
<th>Type</th>
<th>Lengths &amp; Graph rep.</th>
</tr>
</thead>
<tbody>
<tr>
<td>plain product $M(n)$</td>
<td></td>
</tr>
<tr>
<td>A: (0,n)</td>
<td><img src="image1" alt="Graph" /></td>
</tr>
<tr>
<td>B: (0,n)</td>
<td><img src="image2" alt="Graph" /></td>
</tr>
<tr>
<td>C: (0,2n)</td>
<td><img src="image3" alt="Graph" /></td>
</tr>
<tr>
<td>middle product $M(n) + O(n)$</td>
<td></td>
</tr>
<tr>
<td>A: (0,2n)</td>
<td><img src="image4" alt="Graph" /></td>
</tr>
<tr>
<td>B: (0,n)</td>
<td><img src="image5" alt="Graph" /></td>
</tr>
<tr>
<td>C: (n,2n)</td>
<td><img src="image6" alt="Graph" /></td>
</tr>
<tr>
<td>short product $m(n)$</td>
<td></td>
</tr>
<tr>
<td>A: (0,n)</td>
<td><img src="image7" alt="Graph" /></td>
</tr>
<tr>
<td>B: (0,n)</td>
<td><img src="image8" alt="Graph" /></td>
</tr>
<tr>
<td>C: (0,n)</td>
<td><img src="image9" alt="Graph" /></td>
</tr>
<tr>
<td></td>
<td><img src="image10" alt="Graph" /></td>
</tr>
<tr>
<td></td>
<td><img src="image11" alt="Graph" /></td>
</tr>
</tbody>
</table>

**Note:** The graphs are placeholders and should be replaced with actual graphical representations.
Updating inverses

Figure: Naive inverse

Figure: Updating inverse
Fast multiplication

![Graph showing the comparison of time (s.) for different methods (plain, middle, short) across varying degrees.]
Comparison between naive and fast inverse

degree
fast inverse
naive inverse
mul
0
0.5
1
1.5
2
2.5
3
0 200 400 600 800 1000 1200
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.

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

(low-deg, high-deg) = (n, 2n)

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

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

For "x", "^",
"high-deg" is the same as the output.
Assigning low-degrees

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 $G$
- **Evaluation.**
  evaluate $c$ by following the graph $G$
  evaluate $a, b$ by following the graph $G'$
# Output code overview

```c
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, poly0_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

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