Exact Arithmetic on a Computer

Symbolic Computation and Computer Algebra

William J. Turner

Department of Mathematics & Computer Science
Wabash College
Crawfordsville, IN 47933

Tuesday 21 September 2010

Outline

Symbolic Computation

Superset of computer algebra

- Symbols or exact arithmetic
- Exact finite representation of mathematical structures
- Abstract structures (groups, rings, fields, etc.)
- Polynomials and power series
- Linear algebra
- Number theory
- Algebraic geometry
- Computational group theory
- Differential equations
- Automated theorem proving

Computer Algebra Systems

General Purpose Systems

- AXIOM, MAGMA, Maple, *Mathematica*, REDUCE, SAGE

Special Purpose Systems

- CoCoA (Computations in Commutative Algebra)
- GAP (Groups, Algorithms, and Programming)
- NTL (Number Theory Library)
- SINGULAR (polynomial computations)
- *Theorema* (automated theorem proving)

Long History

Ancient Algorithms

- Euclidean Algorithm
- Chinese Remainder Algorithm

Isaac Newton's *The Universal Arithmetic* (1728)

Systematically discusses rules for manipulating universal mathematical expressions, that is, formulae containing symbolic indeterminates, and algorithms for solving equations built with these expressions.

Symbolic Computation vs. Numerical Analysis

Numerical Analysis

- Floating point numbers (approximate real values)
- Find approximation quickly
- Error propagation important
  - Condition Number
  - Stability

Symbolic Computation

- Find exact solution quickly
- May never approximate
  - Structure may not have a metric

Algorithms may not be compatible

- Numerical algorithms may never find exact solution
- Symbolic algorithms may be ill-conditioned or unstable

Infinite Mathematical Structure

General Approach

- Computer has finite memory
- Cannot compute exactly over reals, rationals, integers, etc.
- Compute bound M on desired solution
- Solve via modular algorithms & reconstruct solution
  - Chinese Remainder Algorithm
  - Hensel Lifting
  - Rational Number Reconstruction

Selecting the Modulus

Big Prime Method: $m = p$

Small Prime Method: $m = \prod_i p_i$

Small Prime Power Method: $m = p^\ell$

Storing Integers and Polynomials

Polynomials

- Polynomials $R[x]$ over ring R (e.g., $\mathbb{Z}_m$)
- $a = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \in R[x]$
- store degree n and coefficients a_i for $i = 0, 1, 2, \ldots, n$

Integers

- Radix $r \in \mathbb{N}_{>1}$
- $a = a_n r^n + a_{n-1} r^{n-1} + \cdots + a_1 r + a_0 \in \mathbb{Z}$
- $0 \leq a_i < r$ for $i = 0, 1, 2, \ldots, n$.
- store size n and digits a_i for $i = 0, 1, 2, \ldots, n$

Classical Addition Algorithm

Polynomials

- $a = \sum_{i=0}^{n} a_i x^i$ and $b = \sum_{i=0}^{m} b_i x^i \in R[x]$
- Assume $n = m$: $c = \sum_{i=0}^{n} c_i = a + b = \sum_{i=0}^{n} (a_i + b_i) x^i$

Algorithm

for $i = 0, 1, 2, \ldots n$ **do**

$c_i \leftarrow a_i + b_i$

end for

Complexity

$O(n)$ ring operations

Classical Addition Algorithm

Integer Algorithm

$c_0 \leftarrow 0$
for $i = 0, 1, 2, \ldots n$ **do**
  $c_i \leftarrow a_i + b_i + c_i$
  if $c_i \geq r$ **then**
    $c_i \leftarrow c_i - r$
    $c_{i+1} \leftarrow 1$
  else
    $c_{i+1} \leftarrow 0$
  end if
end for

Complexity

$O(n)$ word operations

Classical Multiplication Algorithm

Polynomial Algorithm

Require: $a = \sum_{i=0}^{n} a_i x^i$ and $b = \sum_{i=0}^{m} b_i x^i$

for $k = 0, 1, 2, \ldots n + m$ **do**

$c_k \leftarrow 0$

for $i = \max\{0, k - m\}, \ldots, \min\{n, k\}$ **do**

$c_k \leftarrow c_k + a_i b_{k-i}$

end for

end for

Complexity

- $O(mn)$ ring operations
- $n = m \Longrightarrow O(n^2)$ ring operations

Classical Multiplication Algorithm

Another Organization

Require: $a = \sum_{i=0}^{n} a_i x^i$ and $b = \sum_{i=0}^{m} b_i x^i$

for $i = 0, 1, 2, \ldots n$ **do**

$d_i \leftarrow a_i x^i \; b$ { x^i just shifts a_i by i places }

end for

return $c \leftarrow \sum_{i=0}^{n} d_i$

Integer Algorithm

Require: $a = (-1)^s \sum_{i=0}^{n} a_i r^i$ and $b = (-1)^t \sum_{i=0}^{m} b_i r^i$

for $i = 0, 1, 2, \ldots n$ **do**

$d_i \leftarrow a_i r^i \; |b|$ { r^i just shifts a_i by i places }

end for

return $c \leftarrow (-1)^{s+t} \sum_{i=0}^{n} d_i$

Classical Division with Remainder Algorithm

Polynomial Synthetic Division Algorithm

Require: $a = \sum_{i=0}^{n} a_i x^i$ and $b = \sum_{i=0}^{m} b_i x^i$ where b_m is a unit and $n \geq m \geq 0$

Ensure: $a = qb + r$ and $\deg r < m$

$r \leftarrow a$ and $u \leftarrow b_m^{-1}$

for $i = n - m, n - m - 1, \ldots 0$ **do**

if $\deg r = m + i$ **then**

$q_i \leftarrow \mathrm{lc}(r)u$ {Leading coefficient of r}

$r \leftarrow r - q_i x^i b$

else

$q_i \leftarrow 0$

end if

end for

return $q \leftarrow \sum_{i=0}^{n-m} q_i x^i$ and r

Fast Multiplication

Roots of Unity

Let R be a ring, $n \in \mathbb{N}_{>1}$, and $\omega \in R$.

- ω is an *nth root of unity* if $\omega^n = 1$.
- ω is a *primitive nth root of unity* if $1 \leq k < n \implies \omega^k \neq 1$.

Discrete Fourrier Transform

- $\mathrm{DFT}_\omega : R^n \to R^n, f \mapsto \left(f(1), f(\omega), f(\omega^2), \ldots, f(\omega^{n-1})\right)$
  - $\deg(f) < n$
  - ω is primitive nth root of unity

Fast Fourrier Transform (FFT)

- Can compute DFT recursively for $n = 2^k$
- $O(n \log n)$ ring operations

Fast Multiplication

DFT and Multiplication

If $\deg(f) + \deg(g) < n$, then $\mathrm{DFT}_\omega(f\ g) = \mathrm{DFT}_\omega(f) \cdot \mathrm{DFT}_\omega(g)$.

Fast Multiplication Algorithm

Require: $\deg(f), \deg(g) < n$

$k \leftarrow \lceil \log_2(2n) \rceil$

$\omega \leftarrow$ primitive 2^kth root of unity in R

$\alpha \leftarrow \mathrm{DFT}_\omega(f)$ and $\beta \leftarrow \mathrm{DFT}_\omega(g)$ {via FFT}

$\gamma \leftarrow \alpha \cdot \beta$ {pointwise multiplication}

return $\mathrm{DFT}_\omega^{-1}(\gamma) = \frac{1}{n}\mathrm{DFT}_{\omega^{-1}}(\gamma)$

Complexity

- $O(n \log n)$ ring operations
- $O(n \log n \operatorname{loglog} n)$ ring operations if must extend ring

Fast Division

Polynomial Reversal

The *reversal* of a polynomial $a = \sum_{i=0}^{n} a_i x^i$ is

$$\mathrm{rev}_k(a) = x^k a\left(\frac{1}{x}\right)$$

When $k = n$, $\mathrm{rev}(a) = \mathrm{rev}_n(a)$ reverses the coefficients of a.

Reversals and Division

If $\deg(a) = n$, $\deg(b) = m$, and $b(0) = 1$, then $\deg(r) < m$ so $\mathrm{rev}_{m-1}(r)$ is a polynomial and

$$\begin{aligned} \mathrm{rev}_n(a) &= \mathrm{rev}_m(b)\,\mathrm{rev}_{n-m}(q) + x^{n-m+1}\,\mathrm{rev}_{m-1}(r) \\ &\equiv \mathrm{rev}_m(b)\,\mathrm{rev}_{n-m}(q) \pmod{x^{n-m+1}} \\ \mathrm{rev}_{n-m}(q) &= \mathrm{rev}_n(a)\ \mathrm{rev}_m(b)^{-1} \bmod x^{n-m+1} \end{aligned}$$

Newton Iteration

Newton's Iteration from Calculus

Require: $\phi(y)$, initial estimate y_0, and tolerance τ

Ensure: $|\phi(\bar{y})| < \tau$

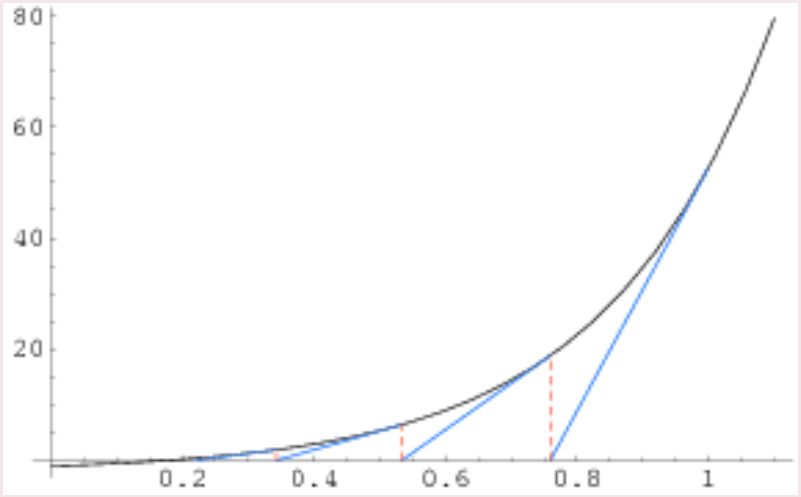
$k \leftarrow 0$

while $|\phi(y_k)| \geq \tau$ **do**

$$y_{k+1} \leftarrow y_k - \frac{\phi(y_k)}{\phi'(y_k)} = y_k - \phi(y_k)\left(\phi'(y_k)\right)^{-1}$$

end while

return $\bar{y} \leftarrow y_k$



Algebra—Not Analysis

Formal Derivative

Let R be a ring (commutative, with 1). For $\phi = \sum_{i=0}^{n} \phi_i y^i \in D[y]$, where D is a ring, we define the *formal derivative* of ϕ by

$$\phi' = \sum_{i=0}^{n} i\phi_i y^{i-1}$$

Approximations

Given a modulus m, measure how well b approximates a by the highest power of m such that $a \equiv b \pmod{m}$.

Example

Let $m = 2$. Then 9 is a better approximation for 17 than 15 because $9 \equiv 17 \pmod{2^3}$ but $15 \not\equiv 17 \pmod{2^3}$.

Inversion Using Newton Iteration

Choosing the Function

- Given $f \in R[x] = D$, want $\phi \in D[y]$ such that $\phi(f^{-1}) = 0$.
- Must be invertible
- Update $\phi\left(\phi'\right)^{-1}$ without division

The Function

- $\phi(y) = \frac{1}{y} - f$
- $\phi'(y) = -\frac{1}{y^2}$
- $\phi(y)\left(\phi'(y)\right)^{-1} = -y + fy^2$
- $y - \phi(y)\left(\phi'(y)\right)^{-1} = 2y - fy^2$

Inversion Using Newton Iteration

Inversion Algorithm

Require: $f \in R[x]$ with $f(0) = 1$ and $\ell \in \mathbb{N}$

Ensure: $g \in R[x]$ with $fg \equiv 1 \pmod{x^\ell}$

$g_0 \leftarrow 1$

$r \leftarrow \lceil \log_2 \ell \rceil$

for $i = 1, \ldots, r$ **do**

$g_i \leftarrow \left(2g_{i-1} - f\ g_{i-1}^2\right) \operatorname{rem} x^{2^i}$ { truncates polynomial }

end for

return g_r

Complexity

- $3M(\ell) + \ell = O(M(\ell))$ ring operations
- $M(\ell)$ is multiplication time

Fast Division with Remainder

Fast Division Algorithm

Require: $a, b \in R[x]$ where $b \neq 0$ is monic.
Ensure: $q, r \in R[x]$ such that $a = qb + r$ and $\deg r < \deg b$
  if $\deg a < \deg b$ **then**
    return $q \leftarrow 0$ and $r \leftarrow a$
  end if
  $m \leftarrow \deg a - \deg b$
  $c \leftarrow (\mathrm{rev}_{\deg b}(b))^{-1} \bmod x^{m+1}$ { Newton Iteration }
  $q^* \leftarrow \mathrm{rev}_{\deg a}(a)\ c \operatorname{rem} x^{m+1}$ { truncates polynomial }
  return $q \leftarrow \mathrm{rev}_m(q^*)$ and $r \leftarrow a - b\ q$

Complexity

- $3M(m) + M(n) + O(n)$ ring operations
- $\deg b = n$ and $\deg a = m + n$

Generalized Newton Iteration

Newton Iteration

Require: $\phi \in D[y]$, $p \in R$, $\ell \in \mathbb{N}_{>0}$, $g_0 \in R$ with $\phi(g_0) \equiv 0 \pmod{p}$ and $\phi'(g_0)$ invertible modulo p, and s_0 such that $s_0\phi'(g_0) \equiv 1 \pmod{p}$

Ensure: $g \in R$ with $\phi(g) \equiv 0 \pmod{p^l}$ and $g \equiv g_0 \pmod{p}$

$r \leftarrow \lceil \log_2 \ell \rceil$

for $i = 1, \ldots, r-1$ **do**

$g_i \leftarrow (g_{i-1} - \phi(g_{i-1})\, s_{i-1}) \bmod p^{2^i}$

$s_i \leftarrow \left(2s_{i-1} - \phi'(g_i)\, s_{i-1}^2\right) \bmod p^{2^i}$

end for

return $g \leftarrow g_{r-1} - \phi(g_{r-1})\, s_{r-1} \bmod p^l$

Complexity

Polynomial Ring

$(3n + 3/2)M(\ell) + O(n\ell)$ when

- $D = R[x]$
- $p = x$
- $\ell = 2^k$
- $\deg_y \phi = n$ and $\deg_x \phi < \ell$

Integers

$O(nM(\ell \log p))$ word operations when

- $R = \mathbb{Z}$
- $0 < g_0 < p$
- $\deg \phi = n$
- $|\phi_i| < p^l$ for $i = 0, 1, \ldots, n$