Generic Modular Computations in Aldor

Jinlong Cai, Steve Wilson & Marc Moreno Maza
(University of Western Ontario)

CATLAN-04, 8-9 July
1. Without modular computations, Computer Algebra would remain theory.

2. The key ideas in modular algorithms are quite simple, but implementing them efficiently is often more tricky than the corresponding non-modular algorithms.

3. The main needs are
   (a) good interface with the machine arithmetic
   (b) good data-structures (primitive arrays, ...)
   (c) memory management care (in-place methods, dispose!, ...),

4. Modular methods are generally specialized to a particular polynomial or matrix ring.

5. But in fact, they are essentially based on two recipes:
   (a) The Chinese Remaindering Theorem
   (b) The Hensel Lemma
Modular computation in $\mathbb{Z}[x]$

The classical modular gcd algorithm

**Input:** $f, g \in \mathbb{Z}[x]$ primitive.

**output:** $\gcd(f, g)$.

\[
\begin{align*}
    b &:= \gcd(\text{lcm}(f), \text{lcm}(g)) \ ;
    d := \min(\deg(f), \deg(g)) \ ;
    (m, g_m) := (1, 0) \\
    \text{repeat} \\
    &\quad \text{choose a prime } p \text{ not dividing } mb \\
    &\quad g_p := b \text{monicGcd}(f \mod p, g \mod p) \text{ in } \mathbb{Z}/\langle p \rangle[x] \\
    &\quad \deg(g_p) = 0 => \text{return } 1 \\
    &\quad \deg(g_p) < d => (m, g_m, d) := (p, g_p, \deg(g_p)) \{ \text{previous unlucky} \} \\
    &\quad \deg(g_p) > d => \text{iterate} \{ \text{unlucky reduction} \} \\
    &\quad w := \text{combine}(p, m)(g_p, g_m) \ ;
    w := \text{symmetricMod}(w, mp) \\
    &\quad \text{if } w = g_m \text{ then} \{ \text{stabilization} \} \\
    &\quad \quad h := \text{pp}(w) \\
    &\quad \quad \text{if } h | f \text{ and } h | g \text{ then return } h \\
    &\quad (m, g_m) := (mp, w)
\end{align*}
\]
Let $E$ be an Euclidean domain with an Euclidean size $\delta$ together with

1. a stream of unassociated primes $p_1, p_2, p_3, \ldots$, such that
   $\delta(p_1) < \delta(p_1 p_2) < \delta(p_1 p_2 p_3) < \cdots$.

2. a mapping $\text{scs}$ from $E \times E \setminus \{0\}$ to $E$ such that

   **Simplification.** For any $a \in E$ and any $m \in E \setminus \{0\}$ we have:
   \[
   a \equiv \text{scs}(a, m) \mod m. \quad (1)
   \]

   **Canonicity.** For any $m \in E \setminus \{0\}$, any two elements $a, b \in E$, we have:
   \[
   (a \equiv b \mod m) \iff (\text{scs}(a, m) = \text{scs}(b, m)). \quad (2)
   \]

   **Recovery = symmetry.** All elements of a bounded degree are recovered by the simplifier if the modulus is sufficiently large.
   That is, for any $B > 0$, there exists $M \in \mathbb{N}$ such that
   \[
   (\forall (a, m) \in E \times E \setminus \{0\}) \left\{ \begin{array}{l}
   \delta(m) \geq M(B) \\
   \delta(a) < B
   \end{array} \right. \Rightarrow \text{scs}(a, m) = a. \quad (3)
   \]
Input: $E$, Euclidean domain and $f, g \in E[x]$ primitive.

output: $\text{gcd}(f, g)$.

\[
\begin{align*}
    b &:= \text{gcd}(\text{lcm}(f), \text{lcm}(g)) \ ;
    d := \min(\deg(f), \deg(g)) \ ;
    (m, g_m) := (1, 0) \\
    \text{repeat} \\
    &\text{choose a prime } p \text{ not dividing } mb \\
    &g_p := b \text{ monicGcd}(f \mod p, g \mod p) \text{ in } E/\langle p \rangle[x] \\
    &\deg(g_p) = 0 \Rightarrow \text{return } 1 \\
    &\deg(g_p) < d \Rightarrow (m, g_m, d) := (p, g_p, \deg(g_p)) \{ \text{previous unlucky} \} \\
    &\deg(g_p) > d \Rightarrow \text{iterate} \{ \text{unlucky reduction} \} \\
    &w := \text{combine}(p, m)(g_p, g_m) \ ;
    w := \text{symmetricMod}(w, mp) \\
    &\text{if } w = g_m \text{ then} \{ \text{stabilization} \} \\
    &\quad h := \text{pp}(w) \\
    &\quad \text{if } h \mid f \text{ and } h \mid g \text{ then return } h \\
    &\quad (m, g_m) := (mp, w)
\end{align*}
\]
1. How to implement these *good* Euclidean domains $E$? We must take into account the fact that the Hensel lifting is *generic* too:

Let $R$ be a commutative ring with identity element. Let $f, g_0, h_0$ be univariate polynomials in $R[x]$ and let $m \in R$. We assume that the following relation holds

$$ f \equiv g_0 h_0 \mod m \quad (4) $$

We assume also that $g_0$ and $h_0$ are relatively prime modulo $m$, that is there exists $s, t \in R$ such that

$$ sg_0 + th_0 \equiv 1 \mod m \quad (5) $$

Then, for every integer $\ell$ there exist $g^{(\ell)}, h^{(\ell)} \in R[x]$ such that we have

(a) $f \equiv g^{(\ell)} h^{(\ell)} \mod m^\ell$,  
(b) $g_0 \equiv g^{(\ell)} \mod m$.

2. How to implement the residue class rings $E/\langle p \rangle$? We want *genericity* but want to preserve *efficiency*.
The CanonicalSimplification category

CanonicalSimplification: Category == CommutativeRing with

  mod: (%, %) -> %
  mod-: (%, %) -> %
  mod+: (%, %, %) -> %
  mod-: (%, %, %) -> %
  mod*: (%, %, %) -> %
  mod^: (%, AldorInteger, %) -> %
  recipMod: (%, %) -> Partial(%)
  invMod: (%, %) -> %

if (% has EuclideanDomain) then symmetricMod: (%, %) -> %

default

  mod-(a: %, p: %): % == ..
  mod-(a: %, b: %, p: %): % == ..
  mod+(a: %, b: %, p: %): % == ..
  mod*(a: %, b: %, p: %): % == ..
  mod^(a: %, n: AldorInteger, p: %): % == ..
  invMod(a: %, b: %): % == ..
SourceOfPrimes: Category == CommutativeRing with
  prime?: % → Partial(Boolean)
  prime?: % → Boolean
  getPrime: () → Partial(%)
  nextPrime: % → Partial(%)

  if (% has EuclideanDomain) then
    getPrimeOfSize: MachineInteger → Partial(%)

  default prime?(x: %): Boolean == ..
ResidueClassRing(R: CommutativeRing, p: R): Category ==
   CommutativeRing with
      modularRep: R -> %
      canonicalPreImage: % -> R

      if (R has EuclideanDomain) then
         symmetricPreImage: % -> R

      if (R has SourceOfPrimes) then
         import from R pretend SourceOfPrimes
         if (prime?(p)) then Field
The ModularComputation Category

ModularComputation: Category == CanonicalSimplification with
residueClassRing: (p: %) -> ResidueClassRing(% , p)

if (% has EuclideanDomain) then
  combine: (% , %) -> (% , %) -> %
  if (% has IntegerCategory) then
    combine: (% , MachineInteger) -> (% , MachineInteger) -> %

default
if (% has EuclideanDomain) then
  combine(M1: % , M2: %): (% , %) -> % == ..

if (% has IntegerCategory) then
  combine(M: % , m: MachineInteger): (% , MachineInteger) -> %

10
The case of univariate polynomials

UnivariatePolynomialCategoryO(R: Join(ArithmeticType,
    ExpressionType)): Category ==

........................................................................................................

if (R has CommutativeRing) then ModularComputation

........................................................................................................

........................................................................................................

UnivariatePolynomialResidueClass(R: CommutativeRing,
    U: UnivariatePolynomialCategoryO(R),
    p: U): ResidueClassRing(U, p)

........................................................................................................

........................................................................................................

UnivariatePolynomialCategoryO(R: Join(ArithmeticType,
    ExpressionType)): Category ==

........................................................................................................

if (R has CommutativeRing) then
    residueClassRing(p: %): ResidueClassRing(% , p) ==
The case of univariate polynomials

UnivariatePolynomialResidueClassRing(R pretend CommutativeRing,\Phi)
A similar treatment has been applied to integers.

Now, one can implement the *Generic Modular Gcd Algorithm* as we saw it before.

But we decided to have fun and implement an optimized one following the implementation of gcd in $\mathbb{Z}[x]$ by Laurent Bernardin and Manuel Bronstein.

- Their package is parametrized as follows
  \[
  \text{ModularUnivariateGcd}(Z:\text{IntegerCategory}, U:\text{UnivariatePolynomialCategory})
  \]
  - It uses a *local* gcd in $\mathbb{Z}/p\mathbb{Z}[x]$ rather than instantiating prime fields.
  - To do so, each polynomial $u$: $U$ modulo a small prime $p$ becomes a *PrimitiveArray MachineInteger*.
  - function signatures look like
    \[
    \]
GenericModularPolynomialGcdPackage(
    R: Join(EuclideanDomain, SourceOfPrimes, ModularComputation),
    U: UnivariatePolynomialCategory(R)): with {
    modularGcd: (U, U) -> Partial(U);
    tryprime: (U, SI, ARR R, U, SI, ARR R, R) -> (R, SI, ARR R);
    remainder!: (SI, ARR R, SI, ARR R, R) -> (SI, ARR R, SI, ARR R);
} == add {
        amodp := arrayMod(a, p); da:= machine degree a;
        bmodp := arrayMod(b, p); db:= machine degree b; local lb, lr:
        repeat {
            (dq, qmodp, dr, rmodp) := remainder!(da, amodp, db, bmodp, p);
            (lr, dr) := leadingCoefficient(rmodp);
            if zero? dr and zero? lr then break;
            amodp := bmodp; da := db; bmodp := rmodp; db := dr; lb := lr
        };
        (lb, db, bmodp); }

14
Timings in ms.

<table>
<thead>
<tr>
<th>$d_x, d_y$</th>
<th>sub-resultants</th>
<th>gen mod gcd</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1600</td>
<td>30</td>
</tr>
<tr>
<td>12</td>
<td>4180</td>
<td>50</td>
</tr>
<tr>
<td>14</td>
<td>9230</td>
<td>60</td>
</tr>
<tr>
<td>16</td>
<td>18570</td>
<td>100</td>
</tr>
<tr>
<td>18</td>
<td>34970</td>
<td>130</td>
</tr>
<tr>
<td>20</td>
<td>59740</td>
<td>160</td>
</tr>
<tr>
<td>30</td>
<td>508440</td>
<td>560</td>
</tr>
</tbody>
</table>

Comparison between subresultant gcd and generic modular gcd for $\mathbb{Z}/p\mathbb{Z}[x][y]$
Timings in ms.

<table>
<thead>
<tr>
<th>$d$</th>
<th>spe mod gcd</th>
<th>gen mod gcd</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>120</td>
<td>240</td>
</tr>
<tr>
<td>250</td>
<td>170</td>
<td>350</td>
</tr>
<tr>
<td>300</td>
<td>250</td>
<td>500</td>
</tr>
<tr>
<td>350</td>
<td>310</td>
<td>640</td>
</tr>
<tr>
<td>400</td>
<td>410</td>
<td>820</td>
</tr>
<tr>
<td>450</td>
<td>530</td>
<td>1050</td>
</tr>
<tr>
<td>500</td>
<td>700</td>
<td>1280</td>
</tr>
</tbody>
</table>

Comparison between the specialized modular gcd and optimized generic modular gcd for $\mathbb{Z}[x]$. 
1. The ratio between the *specialized modular gcd* and *optimized generic modular gcd* is satisfactory. Indeed, the *specialized modular gcd* uses an optimized CRT for integers whereas the *optimized generic modular gcd* uses a generic CRT.

2. What we saw in this talk is part of Aldor 1.0.3 to be downloaded at [www.aldor.org](http://www.aldor.org) soon ...

3. We need to compare *optimized generic modular gcd* and *paper-like generic modular gcd*.

4. We are implementing a *generic multivariate Hensel lifting* (Steve Wilson).