COMPUTER ALGEBRA IN PHYSICS RESEARCH

STANISLAV POSLAVSKY

NRC “KURCHATOV INSTITUTE” — IHEP, PROTVOINO, RUSSIA

ACAT19
SAAS FEE, 2019
OUTLINE

- COMPUTER ALGEBRA
  - KEY MATHEMATICAL CONCEPTS
    - KEY PROGRAMMING ASPECTS
      - BENCHMARKS
cross section \( \sigma = \int d\Phi \left| \sum_i \text{Amplitude}_i \right|^2 \)
Cross section \( \sigma = \int d\Phi \left| \sum_i \text{Amplitude}_i \right|^2 \)

**Simple process** *(textbook example)*:

\[
\begin{align*}
\text{Tr}\{\gamma_\mu (\hat{p} + m) \gamma_\mu (\hat{k} + m) \ldots \} & \quad \frac{1}{(q^2 - m^2)} + \ldots \\
\end{align*}
\]

with paper & pencil

**Real world**: 

thousands of rational expressions, producing millions of terms in the intermediate results
**Computer Algebra & HEP-TH**

Cross section \( \sigma = \int d\Phi \left| \sum_i \text{Amplitude}_i \right|^2 \)

**Simple process (textbook example):**

\[
\begin{align*}
\text{Tr}\{\gamma_\mu (\hat{p} + m)\gamma_\mu (\hat{k} + m) \ldots \} & \quad \frac{(q^2 - m^2)}{}
\end{align*}
\]

with paper & pencil

**Real world:**

thousands of rational expressions, producing millions of terms in the intermediate results

**Key performance bottlenecks:**

**Arithmetic**

**Putting Terms Together**

**Simplification**

**Advanced**

\( \Rightarrow \quad 1. \text{Multiplication, evaluation} \ldots \)

\( \Rightarrow \quad 2. \text{Greatest Common Divisors} \)

\( \Rightarrow \quad 3. \text{Polynomial Factorization} \)

\( \Rightarrow \quad 4. \text{Gröbner bases, elimination} \ldots \)
Yet another program for math?
 Really? What for???

An incomplete list of similar software:

Closed source (proprietary)
- Magma
- Maple
- Mathematica
- Fermat, ...

Open source (free)
- Singular
- Macaulay2
- CoCoA
- Reduce
- Maxima
- Pari/GP, ...
- FLINT
- NTL
- FORM, ...

Rings is aimed to be:

- **Ultrafast**: make it faster than existing tools
- **Lightweight**: portable, extensible and embeddable library (not a CAS)
- **Modern**: API which meets modern best programming practices

Rings:
- is the first such library written in JAVA (90%) & SCALA (10%)
- contains more than 100,000 lines of code
- well, see [https://ringsalgebra.io](https://ringsalgebra.io)
EFFICIENT COMPUTER ALGEBRA: *key concept*

RING HOMOMORPHISM
Euclidean algorithm (GCD):

1 function gcd(a, b)
2     if b = 0
3         return a;
4     else
5         return gcd(b, a mod b);
**Ring Homomorphism: modular methods**

**Euclidean algorithm (GCD):**

```python
1  function gcd(a, b)
2      if b = 0
3          return a;
4      else
5          return gcd(b, a mod b);
```

Applying it to

\[ \gcd\left(1 - x^2 + x^{20} - x^{200}, 1 - x^3 + x^{30} - x^{300}\right) = x - 1 \]

will produce the following 3166 digit number at some intermediate step:

```
21178265677150921740253822599595701172055778537749109160433930907991796863546398308149265416417897047
716799242768103353160906637850318917854160052986686548498598432559533166777746185195074259067328652710
553540538380427535 ...(3166 digits) ...
```
RING HOMOMORPHISM: *modular methods*

Euclidean algorithm (GCD):

1. function gcd(a, b)
2.   if b = 0
3.     return a;
4. else
5.     return gcd(b, a mod b);

Applying it to

\[
\text{gcd}(1 - x^2 + x^{20} - x^{200}, 1 - x^3 + x^{30} - x^{300}) = x - 1
\]

will produce the following 3166 digit number at some intermediate step:

211782656677150921740253822599595701172055778537749109160433930907991796863546398308149265416417897047
716799242768103353160906637850318917854160052986686548498598432559533166777746185195074259067328652710
553540538380427535 \ldots (3166 digits) \ldots

▶ This is *intermediate expression swell*. It occurs always in fact.
▶ Computations become $\infty$ slow due to exponential growth of coefficients
Euclidean algorithm (GCD):

```python
function gcd(a, b)
    if b = 0
        return a;
    else
        return gcd(b, a mod b);
```

Applying it to $\gcd(1 - x^2 + x^{20} - x^{200}, 1 - x^3 + x^{30} - x^{300}) = x - 1$

will produce the following 3166 digit number at some intermediate step:

```
21178265667715092174025382259959570117205577853774910916043930907991796863546398308149265416417897047
716799242768103353160906637850318917854160052986686548498598432559533166777746185195074259067328652710
55805838380257535 ...(3166 digits) ...
```

- This is **intermediate expression swell**. It occurs always in fact.
- Computations become $\infty$ slow due to exponential growth of coefficients

**Observation:**

If we compute modulo 17, we obtain the same result, but all intermediate numbers are bounded by 17
Ring Homomorphism: modular methods

- Idea:
  - compute GCD modulo several different 32-bit primes, then "reconstruct" result

\[
gcd(a, b) \mod 17 = 2 + 4x + 3x^2
\]

\[
gcd(a, b) \mod 19 = 3 + 6x + 2x^2
\]

\[
\Rightarrow gcd(a, b) \mod 17 \times 19 = 155 + 310x + 173x^2
\]

- in practice this is \(\infty\) times faster than direct computation
Ring Homomorphism: modular methods

- **Idea:**
  - compute GCD modulo several different 32-bit primes, than ”reconstruct” result

\[
gcd(a, b) \mod 17 = 2 + 4x + 3x^2
\]
\[
gcd(a, b) \mod 19 = 3 + 6x + 2x^2
\]
\[
\Rightarrow gcd(a, b) \mod 17 \times 19 = 155 + 310x + 173x^2
\]

- in practice this is ∞ times faster than direct computation

- **The same for linear systems:**
  - solving \(Ax = B\):

\[
Ax = B \mod p_1
\]
\[
Ax = B \mod p_2
\]
\[
\Rightarrow Ax = B \mod p_1 \times p_2 \times \ldots
\]
**Ring Homomorphism: Modular Methods**

- **Idea:**
  - Compute GCD modulo several different 32-bit primes, then "reconstruct" result

  \[
gcd(a, b) \mod 17 = 2 + 4x + 3x^2
  \]

  \[
gcd(a, b) \mod 19 = 3 + 6x + 2x^2
  \]

  \[
  \Rightarrow gcd(a, b) \mod 17 \times 19 = 155 + 310x + 173x^2
  \]

  - In practice, this is infinitely times faster than direct computation

- **The same for linear systems:**
  - Solving \(Ax = B\):

  \[
  Ax = B \mod p_1
  \]

  \[
  Ax = B \mod p_2
  \]

  \[
  \Rightarrow Ax = B \mod p_1 \times p_2 \times \ldots
  \]

- **The same everywhere:** factorization, resultant theory, Gröbner bases etc.

  ➤ **Problem** ➜ **Solve mod \(p_1 \ldots p_n\)** ➜ **Reconstruct**

  **harder domain** (\(\mathbb{Z}\)) ➜ **simpler domain** (\(\mathbb{Z}_p\)) ➜ **Chinese Remainders/Hensel lifting**
**Ring Homomorphism: Ideal-adic Methods**

**Problems with integer (rational) coefficients:**

- Problem
- **Solve mod** $p_1 \ldots p_n$
- **Reconstruct**

- **Harder domain** ($\mathbb{Z}$)
- **Simpler domain** ($\mathbb{Z}_p$)

**Chinese Remainders/Hensel Lifting**

Example:

- $\gcd(x^3 y^3, x^4 y^4)$
- Assume: $\gcd(f(x, y); g(x, y)) = x^0 (a_0 + \ldots + a_3 y^3) + x^1 (b_0 + \ldots + b_3 y^3) + \ldots$
- Evaluate: $y = 1$:
  - $\gcd(f(x, 1); g(x, 1)) = x$
- $y = 2$:
  - $\gcd(f(x, 2); g(x, 2)) = x^2$
  - $\vdots$
- $\Rightarrow a_0 = 0, a_1 = 1, \ldots$
**Ring Homomorphism: ideal-adic methods**

- **Problems with integer (rational) coefficients:**
  - **Problem** → **Solve mod** $p_1 \ldots p_n$ → **Reconstruct**
    - *harder domain* ($\mathbb{Z}$)
    - *simpler domain* ($\mathbb{Z}_p$)
    - Chinese Remainders / Hensel lifting

- **Problems with multivariate polynomials:**
  - **Problem** → **Solve at** $\vec{X} = \vec{C}_0, \ldots$ → **Reconstruct**
    - *multivariate* $R[\vec{X}]$
    - *univariate* $R[x_1]$
    - Chinese Remainders / Hensel lifting

Example:

\[
gcd(x^3 y^3; x^4 y^4)\]

Assume:

\[
gcd(f(x, y); g(x, y)) = x^0 (a_0 + \ldots + a_3 y^3) + x^1 (b_0 + \ldots + b_3 y^3) + \ldots
\]

Evaluate:

\[
y = 1: gcd(f(x, 1); g(x, 1)) = x^1
\]

\[
y = 2: gcd(f(x, 2); g(x, 2)) = x^2
\]

\[
\vdots
\]

\[
a_0 = 0; a_1 = 1; \ldots
\]
**Ring Homomorphism: ideal-adic methods**

- **Problems with integer (rational) coefficients:**
  - Problem
  - Solve mod $p_1 \ldots p_n$
  - Lift
  - Reconstruct

  **harder domain** $(\mathbb{Z})$
  **simpler domain** $(\mathbb{Z}_p)$

  **Chinese Remainders/Hensel lifting**

- **Problems with multivariate polynomials:**
  - Problem
  - Solve at $\hat{X} = \hat{C}_0, \ldots$
  - Lift
  - Reconstruct

  **multivariate** $R[\hat{X}]$
  **univariate** $R[x_1]$

  **Chinese Remainders/Hensel lifting**

**Example:** $\gcd(x^3 - y^3, x^4 - y^4)$

Assume:

$$\gcd(f(x, y), g(x, y)) = x^0 (a_0 + \cdots + a_3 y^3) + x^1 (b_0 + \cdots + b_3 y^3) + \cdots$$

Evaluate:

- $y = 1:$ $\gcd(f(x, 1), g(x, 1)) = x - 1$
- $y = 2:$ $\gcd(f(x, 2), g(x, 2)) = x - 2$

$$\implies a_0 = 0, a_1 = 1, \ldots$$
RING HOMOMORPHISM: ideal-adic methods

Problems with integer (rational) coefficients:

- Problem
- mod
- Solve mod $p_1 \ldots p_n$
- lift
- Reconstruct

harder domain ($\mathbb{Z}$)
simpler domain ($\mathbb{Z}_p$)

Problems with multivariate polynomials:

- Problem
- eval
- Solve at $\vec{X} = \vec{C}_0, \ldots$
- lift
- Reconstruct

multivariate $R[\vec{X}]$
univariate $R[x_1]$

Chinese Remainders/Hensel lifting

The math is the same: $R \rightarrow R/\langle m \rangle \rightarrow R$

<table>
<thead>
<tr>
<th>HOMOMORPHISM</th>
<th>$\mathbb{Z} \rightarrow \mathbb{Z}_p$</th>
<th>$R[\vec{X}] \rightarrow R[x_1]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATOR</td>
<td>prime number $p$</td>
<td>prime ideal $I = \langle x_2 - c_2, \ldots \rangle$</td>
</tr>
<tr>
<td>IMAGE FUNCTION</td>
<td>$x \mod p$</td>
<td>$f(x) \mod I = f(x_1, c_2, \ldots)$</td>
</tr>
<tr>
<td>RECONSTRUCTION</td>
<td>Chinese Remainders</td>
<td>Newton’s formula</td>
</tr>
</tbody>
</table>
**Ring Homomorphism: generic view**

- **Problem**: $R$
- **Solve modular problem**: $R/\langle m \rangle$
- **Reconstruct**: \( Chinese \ Remainders/Hensel \ lifting \)

Given problem in $\mathbb{Z}[x_1, x_2, \ldots, x_n]$

\[ \text{mod} \]

Image problem in $\mathbb{Z}_p[x_1, x_2, \ldots, x_n]$

\[ \text{eval} \]

Image problem in $\mathbb{Z}_p[x_1]$

**Solution in** $\mathbb{Z}[x_1, x_2, \ldots, x_n]$

\[ \text{Chinese remainders} \]

Solution in $\mathbb{Z}_p[x_1, x_2, \ldots, x_n]$

\[ \text{Hensel lifting} \]

Solution in $\mathbb{Z}_p[x_1]$

**Solve image problem in** $\mathbb{Z}_p[x_1]$
RING HOMOMORPHISM: generic view

Problem \( R \) \( \longrightarrow \) Solve modular problem \( R/\langle m \rangle \) \( \longrightarrow \) Reconstruct Chinese Remainders/ Hensel lifting

Given problem in \( \mathbb{Q}(\alpha)[x_1, x_2, \ldots, x_n] \)

\( \mod \) Image problem in \( \mathbb{GF}(p, q)[x_1, x_2, \ldots, x_n] \)

\( \mod \) Image problem in \( \mathbb{GF}(p, q)[x_1] \)

Solve image problem in \( \mathbb{GF}(p, q)[x_1] \)

Solution in \( \mathbb{Q}(\alpha)[x_1, x_2, \ldots, x_n] \)

\( \text{eval} \) Solution in \( \mathbb{GF}(p, q)[x_1, x_2, \ldots, x_n] \)

\( \text{Hensel lifting} \) Solution in \( \mathbb{GF}(p, q)[x_1] \)

\( \text{Chinese remainders} \)
SOME PROGRAMMING ASPECTS
PROGRAMMING ASPECTS: general design

- Algebraic concepts are perfect for translating into computer with object oriented programming
- But that’s not easy, only few libraries have e.g. strong typing
- Thanks to Java’s (and Scala’s) perfect OOP model, it became possible in Rings

Generic Euclidean algorithm:

1. `def gcd[E](a: E, b: E)(implicit ring: Ring[E]): E =`
2. `if (b == 0) a else gcd(b, a % b)`
PROGRAMMING ASPECTS: \textit{general design}

- Algebraic concepts are perfect for translating into computer with object oriented programming
- But that’s not easy, only few libraries have e.g. strong typing
- Thanks to Java’s (and Scala’s) perfect OOP model, it became possible in Rings

\textbf{Generic Euclidean algorithm:}

\begin{verbatim}
1 def gcd[E](a: E, b: E)(implicit ring: Ring[E]): E =
2   if (b == 0) a else gcd(b, a % b)
\end{verbatim}

Apply it to polynomials from $\mathbb{Q}[x]$:

\begin{verbatim}
4 implicit val ring1 = UnivariateRing(Q, "x")
5 val p1 = gcd(ring1("x^20 - 1"), ring1("x^30 - 1"))
6 // val p1 : UnivariatePolynomial[Rational[IntZ]] = ...
\end{verbatim}
Algebraic concepts are perfect for translating into computer with object oriented programming

But that’s not easy, only few libraries have e.g. strong typing

Thanks to Java’s (and Scala’s) perfect OOP model, it became possible in Rings

**Generic Euclidean algorithm:**

``` scala
1  def gcd[E](a: E, b: E)(implicit ring: Ring[E]): E =
2   if (b == 0) a else gcd(b, a % b)
```

Apply it to polynomials from \( \mathbb{Q}[x] \):

``` scala
4  implicit val ring1 = UnivariateRing(Q, "x")
5  val p1 = gcd(ring1("x^20 - 1"), ring1("x^30 - 1"))
6  // val p1 : UnivariatePolynomial[Rational[IntZ]] = ...
```

Apply it to polynomials from \( \mathbb{Q}(\pm \sqrt{2})[x] \):

``` scala
7  implicit zRing = Z
8  val num = gcd(zRing("213794398743"), zRing("34345"))
9  // val num : Int~
```
PROGRAMMING ASPECTS: modular arithmetic & CPU

— mod is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more

Real CPU: \( N \mod p \equiv N - \lfloor N/p \rfloor \times p \) — one DIV, one MUL and one SR

▶ DIV has 20-80 times worth throughput than MUL (Intel Skylake)
PROGRAMMING ASPECTS: modular arithmetic & CPU

— mod is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more

Real CPU: \[ N \mod p \equiv N - \lfloor N/p \rfloor \times p \] — one DIV, one MUL and one SR

▶ DIV has 20-80 times worth throughput than MUL (Intel Skylake)

▶ Old hack: use floats!

▶ Compute once the 64-bit float magic = 1.0/p
▶ Then \( \lfloor N/p \rfloor = \lfloor N \times magic \rfloor \) which is one float MUL
▶ In practice 1.5-2 times speed up (Skylake)
▶ It was used in many CASs (NTL, Mathematica, Maple etc.)
PROGRAMMING ASPECTS: modular arithmetic & CPU

— mod is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more

Real CPU: \[ N \mod p \equiv N - \lfloor N/p \rfloor \times p \] — one DIV, one MUL and one SR

▶ DIV has 20-80 times worth throughput than MUL (Intel Skylake)

▶ Old hack: use floats!
  ▶ Compute once the 64-bit float magic = \( 1.0/p \)
  ▶ Then \( \lfloor N/p \rfloor = \lfloor N \times \text{magic} \rfloor \) which is one float MUL
  ▶ In practice 1.5-2 times speed up (Skylake)
  ▶ It was used in many CASs (NTL, Mathematica, Maple etc.)

▶ Current hack:
  ▶ Compute once the magic = \( \lfloor 2^m/p \rfloor \) for sufficiently large \( m \)
  ▶ Then \( \lfloor N/p \rfloor = (N \times \text{magic})/2^m \) which is one MUL and one SHIFT
  ▶ Used in many compilers when divisor is known at compile time:
    - Granlund & Montgomery (1994) — GCC, Go, ...
    - Warren’s Hacker’s delight (2002) — JVM, LLVM, ...
PROGRAMMING ASPECTS: *modular arithmetic & CPU*

— mod is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more

- Fast modulo operation in Rings is approx 2 times faster than built-in %
  - solving linear systems $O(n^3)$ — 8 times faster
  - factoring polynomials $O(n^{1+1\log_2 3})$ — 6 times faster

Can be even faster! New algorithm to compute \textsc{mod} with no \textsc{div} (Lemire, Kaser, Kurz, arXiv:1902.01961 [cs.MS] Feb 2019) up to 25% speed up, really major achievement
PROGRAMMING ASPECTS: modular arithmetic & CPU

- mod is heavily used in cryptographic algorithms, hashing algorithms, distributed systems, low level concurrency and many more

▶ Fast modulo operation in Rings is approx 2 times faster than built-in %
  ▶ solving linear systems $O(n^3)$ — 8 times faster
  ▶ factoring polynomials $O(n^{1+\log_2 3})$ — 6 times faster

▶ Can be even faster!

▶ up to 25% speed up, really major achievement
**PROGRAMMING ASPECTS: polynomial data structures**

- **Univariate polynomial:**
  \[ c_0 + c_1 x + c_2 x^2 + \ldots + c_n x^n \]
  
  **Array:**
  
  \[
  \begin{array}{cccc}
  c_0 & c_1 & c_2 & \ldots & c_n \\
  \end{array}
  \]

- **Fast methods:** Karatsuba, FFT, Newton’s iterations, etc.

- **Multivariate polynomial:**
  \[ 2 x^2 y^3 z^4 + 3 y z^2 + 4 x^2 y + 5 z^3 \]

  **Tree/Hash map:**
  
  \[
  \begin{array}{c}
  x^2 y^1 z^0 \rightarrow 4 \\
  x^0 y^1 z^2 \rightarrow 3 \quad x^2 y^3 z^4 \rightarrow 2 \\
  x^0 y^0 z^3 \rightarrow 5 \quad \text{Null} \\
  \end{array}
  \]

  **Sparse recursive:**
  \[ ((5z^3) + (3z^2) y^1) x^0 + (4y^1 + (2z^4) y^3) x^2 \]

  **Dense recursive:**
  \[ ((0z^0 + 0z^1 + 0z^2 + 5z^3) + (0z^0 + 0z^1 + 3z^2) y^1) x^0 + \ldots \]
**How the data structure affects the performance?**

**Fateman’s benchmark:** multiply \( f(f + 1) \) with

\[
f = (x + y + z + t + 1)^{30}
\]

(there will be 635,376 terms in the result...)

<table>
<thead>
<tr>
<th>System/Library</th>
<th>Time, seconds</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>RINGS (hash map)</td>
<td>15</td>
<td>- not used</td>
</tr>
<tr>
<td>RINGS (dense recursive)</td>
<td>153</td>
<td>- used in Hensel lifting</td>
</tr>
<tr>
<td>RINGS (sparse recursive)</td>
<td>365</td>
<td>- used for evaluation</td>
</tr>
<tr>
<td>RINGS (tree map)</td>
<td>490*</td>
<td>- default</td>
</tr>
<tr>
<td>MAPLE 2018</td>
<td>27</td>
<td>- uses efficient tree map</td>
</tr>
<tr>
<td>MATHEMATICA 11</td>
<td>171</td>
<td>-</td>
</tr>
<tr>
<td>SINGULAR 4.1.1</td>
<td>198</td>
<td>- recursive</td>
</tr>
<tr>
<td>MAGMA V2.23</td>
<td>203</td>
<td>-</td>
</tr>
<tr>
<td>SAGE 8.2</td>
<td>1075</td>
<td>- it’s Python...</td>
</tr>
</tbody>
</table>

* 10s for multiply and 480s to rebalance the tree

https://ulthiel.com/math/other/benchmarks-of-computer-algebra-systems/
Things which programmers pay attention, but scientists often do not:

▶ **Unit & integration tests:**
  - Rings covered with thousands of tests
  - Integration tests run external tools (e.g. SINGULAR CAS) to cross check the correctness

▶ **Randomized testing:**
  - It helped to fix *hundreds* of bugs
  - Several bugs in core routines were reported to MMA, SINGULAR, MAPLE etc.

▶ **Continuous integration (CI):**
  - Rings CI takes several hours to run all tests
  - Each new build may reveal new bugs (thanks to randomized tests!)
Computer algebra: benchmarks

BENCHMARKS
POLYNOMIAL GCD:

- take random polynomials \( a, b, c \) and compute \( \gcd(ac, bc) \)

POLYNOMIAL FACTORIZATION:

- take random polynomials \( a, b, c \) and compute \( \text{factor}(abc) \) and \( \text{factor}(abc + 1) \) (irreducibility test)
**BENCHMARKS: polynomial GCD**

**Params (a,b,g):**

- #terms = 40
- #bits = 32
- $\text{exp}_{\text{min}} = 0$
- $\text{exp}_{\text{max}} = 30$

---

- #terms = 40
- #bits = 32
- $\text{exp}_{\text{tot}} = 50$

---

- #terms = 40
- #bits = 1
- $\text{exp}_{\text{min}} = 0$
- $\text{exp}_{\text{max}} = 30$

---

**uniform exponents (characteristic 0)**

- timeout = 8 hours

---

**sharp exponents (characteristic 0)**

- timeout = 8 hours

---

**uniform exponents (characteristic 2)**

- timeout = 8 hours

---

<table>
<thead>
<tr>
<th>#vars</th>
<th>Rings</th>
<th>Mathematica</th>
<th>FORM</th>
<th>Fermat</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BENCHMARKS: polynomial factorization

Params:

#factors = 3
#terms = 20
exp_{min} = 0
exp_{max} = 30
CONCLUSIONS

► **MODERN hep-th REQUIRES HIGH PERFORMANCE COMPUTER ALGEBRA TOOLS**

► **FASTER ALGORITHMS AND MORE EFFICIENT IMPLEMENTATIONS APPEAR FROM TIME TO TIME**
  
  ▶ **MORE DETAILS ON **R**INGS** **CAN BE FOUND AT**
  
  ● [HTTPS://RINGSALGEBRA.IO](https://ringsalgebra.io)
  ● [HTTPS://GITHUB.COM/POSLOFSKYSV/RINGS](https://github.com/PoslavskySV/rings)

► **THE FURTHER SCALING MAY BE ACHIEVED BY USING DISTRIBUTED COMPUTING**

**THANKS FOR ATTENTION!**
Rings: an overview

- **Computational Number Theory**
  - primes: sieving, testing, factorization
  - univariate polynomials over arbitrary coefficient rings: 
    fast arithmetic, gcd, factorization etc.
  - Galois fields & Algebraic number fields

- **Computational Commutative Algebra**
  - multivariate polynomials over arbitrary coefficient rings: 
    fast arithmetic, gcd, factorization etc.
  - fast rational function arithmetic

- **Computational Algebraic Geometry**
  - Gröbner bases
  - Ideals in multivariate polynomial rings

- **Programming in Scala**
  - object-oriented and functional programming in one concise, 
    high-level and statically typed language
Basic algebraic definitions

• **Ring**: a set of elements with "+" and "×" operations defined.
  
  **Examples**:
  
  • \( \mathbb{Z} \) — ring of integers
  
  • \( \mathbb{Z}[i] \) — Gaussian integers
  
  • \( R[X] \) — polynomials with coefficients from ring \( R \)

• **Field**: a ring with "/" (division) operation.
  
  **Examples**:
  
  • \( \mathbb{Q} \) — field of rational numbers
  
  • \( \mathbb{Z}_p \) — field of integers modulo a prime number
  
  • \( Frac(R[X]) \) — field of rational functions

• **Ideal**: a subset of ring elements closed under multiplication with ring.
  
  **Examples**:
  
  • Given a set of generators \( \{f_i(x, y, ...)\} \in R[x, y, ...] \) ideal is formed by all elements of the form

\[
c_1(x, y, ...) \times f_1(x, y, ...) + ... + c_n(x, y, ...) \times f_n(x, y, ...)
\]
Rings: implementation aspects

- Efficient Z/p rings
- Arbitrary precision integer arithmetic
  - Prime numbers: sieving / testing
  - Fast univariate arithmetic
    - Subresultant sequences
    - Univariate (e)GCD (Half-GCD, Modular, Subresultant)
      - Square-free factorization
    - Univariate factorization in finite fields
    - Univariate factorization in Z[x]
    - Univariate Hensel (p-adic) lifting
  - Univariate (e)GCD (Half-GCD, Modular, Subresultant)
- Multivariate GCD (Brown / Zippel / EEZ / generic)
  - Multivariate interpolation
  - Univariate interpolation
  - Multivariate Hensel (ideal-adic) lifting (dense / sparse)
  - Square-free factorization
  - Multivariate factorization
- Rational function arithmetic
- Groebner bases
- Ideals and quotient rings

Dependency graph
Rings: design by examples

Simple example:

```scala
1  implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
2  val x = ring("x") // parse polynomial from string
3  val poly = x.pow(100) - 1 // construct polynomial programmatically
4  val factors = Factor(poly) // factorize polynomial
5  println(factors)
```
Rings: *design by examples*

Simple example:

```scala
1  implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
2  val x = ring("x") // parse polynomial from string
3  val poly = x.pow(100) - 1 // construct polynomial programmatically
4  val factors = Factor(poly) // factorize polynomial
5  println(factors)
```

- Explicit types are omitted for shortness, though Scala is fully statically typed

```scala
val ring : Ring[UnivariatePolynomial[Rational[IntZ]]] = ...
val poly : UnivariatePolynomial[Rational[IntZ]] = ...
```

*(types are inferred automatically at compile time if not specified explicitly)*
Rings: design by examples

Simple example:

```scala
1  implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]
2  val x = ring("x") // parse polynomial from string
3  val poly = x.pow(100) - 1 // construct polynomial programmatically
4  val factors = Factor(poly) // factorize polynomial
5  println(factors)
```

- Explicit types are omitted for shortness, though Scala is fully statically typed

```scala
defining types:
val ring : Ring[UnivariatePolynomial[Rational[IntZ]]] = ...
val poly : UnivariatePolynomial[Rational[IntZ]] = ...
```

_(types are inferred automatically at compile time if not specified explicitly)_

- Trait Ring[E] implements the concept of mathematical ring and defines all basic algebraic operations over the elements of type E

```scala
println( ring.isField ) // access ring properties
println( ring.characteristic ) // access ring characteristic
println( ring.cardinality ) // access ring cardinality
```
Rings: *design by examples*

**Simple example:**

1. `implicit val ring = UnivariateRing(Q, "x") // base ring Q[x]`
2. `val x = ring("x") // parse polynomial from string`
3. `val poly = x.pow(100) - 1 // construct polynomial programmatically`
4. `val factors = Factor(poly) // factorize polynomial`
5. `println(factors)`

- Explicit types are omitted for shortness, though Scala is fully statically typed
  
  ```scala
  val ring : Ring[UnivariatePolynomial[Rational[IntZ]]] = ...
  val poly : UnivariatePolynomial[Rational[IntZ]] = ...
  ```

  *(types are inferred automatically at compile time if not specified explicitly)*

- Trait `Ring[E]` implements the concept of mathematical ring and defines all basic algebraic operations over the elements of type `E`

  ```scala
  println( ring.isField ) // access ring properties
  println( ring.characteristic ) // access ring characteristic
  println( ring.cardinality ) // access ring cardinality
  ```

- The `implicit` brings operator overloading via type enrichment (continue =>)
Rings: *design by examples*  

**Meaning of implicits:**

```scala
1 // ring of elements of type E
2 implicit val ring : Ring[E] = ...
3 val a : E = ...
4 val b : E = ...

6 val sum = a + b // compiles to ring.add(a, b)
7 val mul = a * b // compiles to ring.multiply(a, b)
8 val div = a / b // compiles to ring.divideExact(a, b)
```

**Example:**

```scala
1 val a : IntZ = Z(12)
2 val b : IntZ = Z(13)
3 assert (a * b == Z(156))  // no any implicit Ring[IntZ]

5 implicit val ring = Zp(17) // implicit Ring[IntZ]
6 assert (a * b == Z(3))  // multiplication modulo 17
```
Multivariate polynomials

1 // base ring Q[x, y, z]
2 implicit val ring = MultivariateRing(Q, Array("x", "y", "z"))
3 val (x, y, z) = ring("x", "y", "z") // parse polynomials from strings

5 val poly1 = (x + y + z).pow(10) - 1 // construct poly
6 val poly2 = ring("(x + y + z)^3 + 1") // or just parse from string

8 println( PolynomialGCD(poly1, poly2) ) // compute GCD
9 println( Factor(poly1) ) // factorize polynomial

11 // construct some non-trivial polynomial ideal
12 implicit val ideal = Ideal(Seq(poly1 - x, poly2 - y), LEX)
13 assert ( ideal.dimension == 1 )

15 // reduce poly modulo ideal
16 assert ( poly1 %% ideal == x )
17 assert ( poly2 %% ideal == y )
Rings: *design by examples*

**Rational function arithmetic:**

```scala
1  // rational functions Frac(Z[x, y, z])
2  implicit val ring = Frac(MultivariateRing(Z, Array("x", "y", "z")))
3  val (x, y, z) = ring("x", "y", "z") // parse elements from strings

5  // construct expression
6  val expr1 = x / y + z.pow(2) / (x + y - 1)

8  // or import from file
9  import scala.io.Source
10 val expr2 = ring(Source.fromFile("myFile.txt").mkString)

12 val expr3 = expr1 * expr2
13 // unique factor decomposition of fraction
14 println ( ring.factor(expr3) )
```

Fractions are always reduced to a common denominator and GCD is cancelled automatically;
## Rings: *design by examples*

<table>
<thead>
<tr>
<th>Built-in ring</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{Z}$</td>
<td>ring of integers</td>
</tr>
<tr>
<td>$\mathbb{Q}$</td>
<td>field of rationals</td>
</tr>
<tr>
<td>GaussianRationals</td>
<td>field of complex rational numbers $\mathbb{Q}(i)$</td>
</tr>
<tr>
<td>$\mathbb{Z}_p(p)$</td>
<td>integers modulo $p$</td>
</tr>
<tr>
<td>$\text{GF}(p,q)$</td>
<td>finite field with cardinality $p^q$</td>
</tr>
<tr>
<td>AlgebraicNumberField(alpha)</td>
<td>algebraic number field $F(\alpha_1, \ldots, \alpha_s)$</td>
</tr>
<tr>
<td>$\text{Frac}(R)$</td>
<td>field of fractions over Euclidean ring $R$</td>
</tr>
<tr>
<td>UnivariateRing(R, x)</td>
<td>univariate ring $R[x]$</td>
</tr>
<tr>
<td>MultivariateRing(R, vars)</td>
<td>multivariate ring $R[x_1, x_2, \ldots]$</td>
</tr>
<tr>
<td>QuotientRing(R, ideal)</td>
<td>multivariate quotient ring $R[x_1, x_2, \ldots]/I$</td>
</tr>
</tbody>
</table>
Rings: *design by examples*

**Diophantine equations:** solve $\sum f_is_i = gcd(f_1, \ldots, f_N)$ for given $f_i$ and unknown $s_i$: 
Rings: *design by examples*

**Diophantine equations:** solve $\sum f_is_i = \gcd(f_1, \ldots, f_N)$ for given $f_i$ and unknown $s_i$:

```scala
def solveDiophantine[E](fi: Seq[E])(implicit ring: Ring[E]) =
  fi.foldLeft((ring(0), Seq.empty[E])) {
    case ((gcd, seq), f) =>
      val xgcd = ring.extendedGCD(gcd, f)
      (xgcd(0), seq.map(_ * xgcd(1)) ++ xgcd(2))
  }
```

Diophantine equations: solve $\sum f_i s_i = \gcd(f_1, \ldots, f_N)$ for given $f_i$ and unknown $s_i$:

```scala
def solveDiophantine[E](fi: Seq[E])(implicit ring: Ring[E]) = 
  fi.foldLeft((ring(0), Seq.empty[E])) { case ((gcd, seq), f) => 
    val xgcd = ring.extendedGCD(gcd, f) 
    (xgcd(0), seq.map(_ * xgcd(1)) ++ xgcd(2)) 
  }
```

Diophantine equations in $\text{Frac}(\text{GF}(17^3)[x, y, z])[W]$:

``` scala
// Galois field GF(17, 3)
implicit val gf = GF(17, 3, "t")
// Rational functions in x, y, z over GF(17, 3)
implicit val fracs = Frac(MultivariateRing(gf, Array("x", "y", "z")))
// univariate ring Frac(GF(17, 3)[x,y,z])[W]
implicit val ring = UnivariateRing(fracs, "W")

val f1 = ring("1 + t^2 + x/y - W^2") // parse elements from strings
val f2 = ring("1 + W + W^3/(t - x)") // parse elements from strings
val f3 = ring("t^2 - x - W^4") // parse elements from strings
// do the job
val solve = solveDiophantine(Seq(f1, f2, f3))
```

▶ this is a piece of one-loop master integral reduction algorithm
Rings: implementation aspects

- Efficient $\mathbb{Z}/p$ rings
  - Arbitrary precision integer arithmetic
  - Prime numbers: sieving / testing
    - Fast univariate arithmetic
      - Subresultant sequences
      - Univariate (e)GCD (Half-GCD, Modular, Subresultant)
      - Square-free factorization
      - Univariate factorization in finite fields
      - Univariate Hensel (p-adic) lifting
    - Univariate factorization in $\mathbb{Z}[x]$
    - Univariate primality test
    - Galois fields
    - Conclusions
  - Univariate factorization in $\mathbb{Z}[x]$
  - Univariate interpolation
  - Multivariate Hensel (ideal-adic) lifting (dense / sparse)
  - CRT / Rational reconstruction
  - Multivariate GCD (Brown / Zippel / EEZ / generic)
  - Groebner bases
  - Ideals and quotient rings
  - Multivariate factorization
  - Rational function arithmetic
- Fast multivariate arithmetic
  - Multivariate GCD (Brown / Zippel / EEZ / generic)
  - Multivariate Hensel (ideal-adic) lifting (dense / sparse)
  - Fast univariate arithmetic
    - Subresultant sequences
    - Univariate interpolation
    - Multivariate interpolation
    - Univariate interpolation
- Square-free factorization
- Dependency graph
Rings: polynomials

- Polynomials over $\mathbb{Z}_p$ with $p < 2^{64}$ (machine numbers) have separate implementations
  - `E[]` data — generic array for univariate polynomials over generic rings (with elements of reference type $E$)
  - `long[]` data — native array for univariate polynomials over $\mathbb{Z}_p$ with $p < 2^{64}$ (machine words)

- Motivation:
  - $\mathbb{Z}_p$ with $p < 2^{64}$ already has separate implementation
  - more specific and optimized algorithms
  - avoid inefficient generics with primitive types in Java (however, e.g. in C/C++ one would have to do the same, like in NTL)
Rings: polynomials

- **Interface**
  - `IPolynomial<PolyType>`

- **Abstract class**
  - `AMultivariatePolynomial<MonomialType, PolyType>`

- **Final class**
  - `IUnivariatePolynomial<PolyType>`
  - `MultivariatePolynomialZp64`
    - Multivariate polynomials over $\mathbb{Z}/p$ with $p < 2^{64}$ (over machine integers)
  - `UnivariatePolynomialZp64`
    - Univariate polynomials over $\mathbb{Z}/p$ with $p < 2^{64}$ (over machine integers)
  - `UnivariatePolynomial<E>`
    - Univariate polynomials over generic coefficient ring $\text{Ring}<E>$
  - `MultivariatePolynomial<E>`
    - Multivariate polynomials over generic coefficient ring $\text{Ring}<E>$
  - `Monomial<E>`
    - Monomials with generic coefficients
# Rings: polynomials

<table>
<thead>
<tr>
<th></th>
<th>Univariate</th>
<th>Multivariate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>((n) is polynomial degree)</td>
<td>((n) is polynomial size)</td>
</tr>
<tr>
<td><strong>Addition/Subtraction</strong></td>
<td>(O(n))</td>
<td>(O(n\log n))</td>
</tr>
<tr>
<td></td>
<td>(O\left( n^{\log_2 3} \right) )</td>
<td>(O\left( nm\log(n)\log(m) \right) )</td>
</tr>
<tr>
<td>Multiplication</td>
<td>via Karatsuba method (with lots of heuristic to reduce the constant)</td>
<td>via plain method (Kronecker trick is used to significantly reduce the constant)</td>
</tr>
<tr>
<td>Division</td>
<td>(O\left( n^{\log_2 3} \right) )</td>
<td>(O\left( nm\log(n)\log(m) \right) )</td>
</tr>
<tr>
<td>Evaluation</td>
<td>(O(n))</td>
<td>(O(n\log(d)))</td>
</tr>
<tr>
<td></td>
<td>via Horner method</td>
<td>via plain method with caching or via recursive Horner scheme</td>
</tr>
</tbody>
</table>
**Rings: polynomial GCD**

- **Univariate (e)GCD:**
  - Rings switches between Euclidean GCD, Half-GCD and Brown’s GCD (for coefficient rings with characteristic zero)

- **Multivariate GCD:**
  - For sparse inputs Rings uses Zippel’s algorithm based on linear algebra
  - For relatively dense polynomials Rings uses Enhanced Extended Zassenhaus (EEZ) approach based on multivariate (ideal-adic) Hensel lifting
  - When the coefficient ring has very small cardinality Rings uses a version of Kaltofen-Monagan generic GCD algorithm
  - For coefficient rings of characteristic zero, modular algorithm (Zippel-like for sparse or Brown-like with EEZ for dense inputs) is used

- All these contain tons of heuristic (code for algorithms spans more than 6,000 l.o.c.)
Rings: polynomial GCD

Benchmarks:

- Generate three polynomials $a$, $b$ and $g$ at random and compute $gcd(a g, b g)$ (non-trivial) and $gcd(a g + 1, b g)$ (trivial)
- Terms of polynomials are generated independently
- Two ways to generate exponents inside terms:
  - **Uniform exponents** (uniform distribution):
    choose each exponent independently in range $\exp_{\min} \leq \exp_i < \exp_{\max}$; the total degree will be $N_{\text{vars}} \exp_{\min} < \exp_{\text{tot}} < N_{\text{vars}} \exp_{\max}$
    **Example** ($\exp_{\min} = 0$, $\exp_{\max} = 10$):
    \[
    \ldots + x^5 y^2 z^8 + x^3 y^8 z^6 + \ldots
    \]
  - **Sharp exponents** (multinomial distribution):
    choose the total degree $\exp_{\text{tot}}$, then for the first variable $0 \leq \exp_1 \leq \exp_{\text{tot}}$, for the second variable $0 \leq \exp_2 \leq (\exp_{\text{tot}} - \exp_1)$ and so on
    **Example** ($\exp_{\text{tot}} = 10$):
    \[
    \ldots + x^7 y^2 z^1 + x^0 y^8 z^2 + \ldots
    \]
Rings: polynomial GCD

“Record” problems:

<table>
<thead>
<tr>
<th>size of input polynomials</th>
<th>time per problem, s</th>
</tr>
</thead>
<tbody>
<tr>
<td>(sparse)</td>
<td>(dense)</td>
</tr>
<tr>
<td>#vars = 3</td>
<td></td>
</tr>
<tr>
<td>#vars = 4</td>
<td></td>
</tr>
</tbody>
</table>

Params (a,b,g):

exp<sub>tot</sub> = 50 / #bits = 128 / #terms = 50, 100, 500, 1000, 5000
Dense input:

\[ a = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1 \]
\[ b = (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1 \]
\[ g = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3 \]
Rings: polynomial GCD

Dense input:

\[ a = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1 \]
\[ b = (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1 \]
\[ g = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3 \]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Mathematica</th>
<th>FORM</th>
<th>Fermat</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{gcd}(ag, bg) )</td>
<td>( \mathbb{Z} )</td>
<td>104s</td>
<td>115s</td>
<td>148s</td>
<td>1759s</td>
<td>141s</td>
</tr>
<tr>
<td>( \text{gcd}(ag, bg + 1) )</td>
<td>( \mathbb{Z} )</td>
<td>0.4s</td>
<td>2s</td>
<td>0.3s</td>
<td>0.1s</td>
<td>0.4s</td>
</tr>
<tr>
<td>( \text{gcd}(ag, bg) )</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>25s</td>
<td>33s</td>
<td>N/A</td>
<td>147s</td>
<td>46s</td>
</tr>
<tr>
<td>( \text{gcd}(ag, bg + 1) )</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>0.5s</td>
<td>2s</td>
<td>N/A</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
</tbody>
</table>

\( \triangleright \) GCD performance on trivial input is very important (since e.g. most part of GCDs computed in rational function arithmetic are trivial)
Rings: polynomial GCD

Dense input:

\begin{align*}
a &= (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^7 - 1 \\
b &= (1 - 3x_1 - 5x_2 - 7x_3 + 9x_4 - 11x_5 - 13x_6 + 15x_7)^7 + 1 \\
g &= (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 - 15x_7)^7 + 3
\end{align*}

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Mathematica</th>
<th>FORM</th>
<th>Fermat</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>( gcd(a, bg) )</td>
<td>( \mathbb{Z} )</td>
<td>104s</td>
<td>115s</td>
<td>148s</td>
<td>1759s</td>
<td>141s</td>
</tr>
<tr>
<td>( gcd(a, bg + 1) )</td>
<td>( \mathbb{Z} )</td>
<td>0.4s</td>
<td>2s</td>
<td>0.3s</td>
<td>0.1s</td>
<td>0.4s</td>
</tr>
<tr>
<td>( gcd(a, bg) )</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>25s</td>
<td>33s</td>
<td>N/A</td>
<td>147s</td>
<td>46s</td>
</tr>
<tr>
<td>( gcd(a, bg + 1) )</td>
<td>( \mathbb{Z}_{524287} )</td>
<td>0.5s</td>
<td>2s</td>
<td>N/A</td>
<td>0.2s</td>
<td>0.2s</td>
</tr>
</tbody>
</table>

- GCD performance on trivial input is very important (since e.g. most part of GCDs computed in rational function arithmetic are trivial)
- one have to make a trade-off between performance on non-trivial and trivial inputs
Rings: polynomial factorization

Univariate factorization:
- \texttt{Rings} switches between Cantor-Zassenhaus and Shoup’s baby-step-giant-step algorithms for polynomials over finite fields
- p-adic Hensel lifting is used to compute factorization over $\mathbb{Z}$ (resp. $\mathbb{Q}$)

Multivariate factorization:
- for bivariate polynomials Bernardin’s algorithm is used
- Kaltofen’s algorithm is used in all other cases
- ideal-adic Hensel lifting switches between sparse (based on linear algebra) and dense (based on Bernardin’s algorithm)
- all these contain tons of heuristic
**Benchmark:** generate three polynomials $a$, $b$ and $c$ at random and compute $\text{factor}(abc)$ (non-trivial) and $\text{factor}(abc + 1)$ (trivial)

**Params:**
- #factors = 3
- #terms = 20
- $\exp_{\text{min}} = 0$
- $\exp_{\text{max}} = 30$
Rings: *polynomial factorization*

Dense input:

\[
p_1 = (1 + 3x_1 + 5x_2 + 7x_3 + 9x_4 + 11x_5 + 13x_6 + 15x_7)^{15} - 1
\]

\[
p_2 = -1 + (1 + 3x_1 x_2 + 5x_2 x_3 + 7x_3 x_4 + 9x_4 x_5 + 11x_5 x_6 + 13x_6 x_7 + 15x_7 x_1)^3
\times(1 + 3x_1 x_3 + 5x_2 x_4 + 7x_3 x_5 + 9x_6 x_5 + 11x_7 x_6 + 13x_6 x_1 + 15x_7 x_2)^3
\times(1 + 3x_1 x_4 + 5x_2 x_5 + 7x_3 x_6 + 9x_6 x_7 + 11x_7 x_1 + 13x_6 x_2 + 15x_7 x_3)^3
\]

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Singular</th>
<th>Mathematica</th>
</tr>
</thead>
<tbody>
<tr>
<td>factor(p1)</td>
<td>(\mathbb{Z})</td>
<td>55s</td>
<td>20s</td>
<td>271s</td>
</tr>
<tr>
<td>factor(p1)</td>
<td>(\mathbb{Z}_2)</td>
<td>0.25s</td>
<td>&gt; 1h</td>
<td>N/A</td>
</tr>
<tr>
<td>factor(p1)</td>
<td>(\mathbb{Z}_{524287})</td>
<td>28s</td>
<td>109s</td>
<td>N/A</td>
</tr>
<tr>
<td>factor(p2)</td>
<td>(\mathbb{Z})</td>
<td>23s</td>
<td>12s</td>
<td>206s</td>
</tr>
<tr>
<td>factor(p2)</td>
<td>(\mathbb{Z}_2)</td>
<td>6s</td>
<td>3s</td>
<td>N/A</td>
</tr>
<tr>
<td>factor(p2)</td>
<td>(\mathbb{Z}_{524287})</td>
<td>26s</td>
<td>9s</td>
<td>N/A</td>
</tr>
</tbody>
</table>
Rings: polynomial factorization

Univariate input:

\[ p_{\deg}[x] = 1 + \sum_{i=1}^{\deg} i \times x^i \]

This benchmark covers almost all aspects of univariate arithmetic in finite fields.
Rings: Gröbner bases

- **Note:** Rings is not optimized for computing Gröbner bases for “challenging” problems yet (like those arise in post-quantum cryptography).
- Gröbner bases for graded orders for polynomials over finite fields computed with Faugere’s F4 algorithm (hardly based on fast sparse linear algebra).
- In other cases Rings may switch between Buchberger algorithm (with different selection strategies), Hilbert-driven methods or modular algorithms.
- Again, many heuristics applied.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Cf. ring</th>
<th>Rings</th>
<th>Mathematica</th>
<th>Singular</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclic-7</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>3s</td>
<td>26s</td>
<td>N/A</td>
</tr>
<tr>
<td>cyclic-8</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>51s</td>
<td>897s</td>
<td>39s</td>
</tr>
<tr>
<td>cyclic-9</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>14603s</td>
<td>$\infty$</td>
<td>8523s</td>
</tr>
<tr>
<td>katsura-7</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>0.5s</td>
<td>2.4s</td>
<td>0.1s</td>
</tr>
<tr>
<td>katsura-8</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>2s</td>
<td>24s</td>
<td>1s</td>
</tr>
<tr>
<td>katsura-9</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>2s</td>
<td>22s</td>
<td>1s</td>
</tr>
<tr>
<td>katsura-10</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>9s</td>
<td>216s</td>
<td>9s</td>
</tr>
<tr>
<td>katsura-11</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>54s</td>
<td>2295s</td>
<td>65s</td>
</tr>
<tr>
<td>katsura-12</td>
<td>$\mathbb{Z}_{1000003}$</td>
<td>363s</td>
<td>28234s</td>
<td>677s</td>
</tr>
<tr>
<td>katsura-7</td>
<td>$\mathbb{Z}$</td>
<td>5s</td>
<td>4s</td>
<td>1.2s</td>
</tr>
<tr>
<td>katsura-8</td>
<td>$\mathbb{Z}$</td>
<td>39s</td>
<td>27s</td>
<td>10s</td>
</tr>
<tr>
<td>katsura-9</td>
<td>$\mathbb{Z}$</td>
<td>40s</td>
<td>29s</td>
<td>10s</td>
</tr>
<tr>
<td>katsura-10</td>
<td>$\mathbb{Z}$</td>
<td>1045s</td>
<td>251s</td>
<td>124s</td>
</tr>
</tbody>
</table>
Rings: note on the programming languages

- The choice of programming language is not so important as e.g. the choice of algorithms and careful design of the API

- Rings is written in Java and also provides extensive Scala API
  - Java: *just the most popular language*
    - extremely fast, very simple, cross-platform, has the largest community, comes with a dependency manager
    - with the same simplicity can be executed on PC, cluster or a wash machine
  - Scala: *object-oriented and functional programming in one concise, high-level and statically typed language*
    - has many recent developments from the theory of programming languages
    - very flexible and expressive: allows to write code very fast
    - also popular: e.g. Twitter and Spark are written in Scala
Rings: note on the programming languages

- The choice of programming language is not so important as e.g. the choice of algorithms and careful design of the API

- **Rings is written in Java and also provides extensive Scala API**
  - **Java**: *just the most popular language*
    - extremely fast, very simple, cross-platform, has the largest community, comes with a dependency manager
    - with the same simplicity can be executed on PC, cluster or a wash machine
  - **Scala**: *object-oriented and functional programming in one concise, high-level and statically typed language*
    - has many recent developments from the theory of programming languages
    - very flexible and expressive: allows to write code very fast
    - also popular: e.g. Twitter and Spark are written in Scala

- If you need to compute something quickly, you will find that it is easy
- If you need to program something, you will find that it is convenient
Example:
Given polynomial fraction

\[
\frac{1}{((s - t)^2 - m_3^2)(s^2 - m_1^2)(t^2 - m_2^2)}
\]

decompose it in a sum of fractions such that denominators in each fraction are algebraically independent in \((s, t)\)

NOTE: denominators are dependent since

\[
(m_1 - m_2 - m_3)(m_1 + m_2 - m_3)(m_1 - m_2 + m_3)(m_1 + m_2 + m_3) \\
+ 2(-m_1^2 - m_2^2 + m_3^2) Y_1 + 2(m_1^2 - m_2^2 - m_3^2) Y_2 + 2(m_1^2 - m_2^2 - m_3^2) Y_3 \\
+ Y_1^2 + Y_2^2 + Y_3^2 - 2 Y_1 Y_2 - 2 Y_1 Y_3 - 2 Y_2 Y_3 \equiv 0
\]

\[
Y_1 = ((s - t)^2 - m_3^2) \quad Y_2 = (s^2 - m_1^2) \quad Y_3 = (t^2 - m_2^2)
\]
Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```scala
1 // field of coefficients Frac(Z[m1, m2, m3])
2 val cfs = Frac(MultivariateRing(Z, Array("m1","m2","m3")))
3 // field of rational functions Frac(Frac(Z[m1, m2, m3])[s, t])
4 implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
5 // parse variables from strings
6 val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

8 val frac = (1 / ((s - t).pow(2) - m3.pow(2))
9    / (s.pow(2) - m1.pow(2))
10   / (t.pow(2) - m2.pow(2)))
11 // or just parse from string
12 // val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")
13```

PoslavskySV
Rings: design by examples

Multivariate polynomials & rational functions & simplifications

1 // field of coefficients Frac(Z[m1, m2, m3])
2 val cfs = Frac(MultivariateRing(Z, Array("m1","m2","m3")))
3 // field of rational functions Frac(Frac(Z[m1, m2, m3])[s, t])
4 implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
5 // parse variables from strings
6 val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")
7
8 val frac = (1 / ((s - t).pow(2) - m3.pow(2))
9     / (s.pow(2) - m1.pow(2))
10     / (t.pow(2) - m2.pow(2)))
11 // or just parse from string
12 // val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")
13
14 // bring in the form with algebraically independent denominators
15 val dec = GroebnerMethods.LeinartDecomposition(frac)
16 // simplify fractions (factorize)
17 val decSimplified = dec.map(f => field.factor(f))
18 // pretty print
19 decSimplified.map(f => field.stringify(f)).foreach(println)
Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```scala
1  // field of coefficients Frac(Z[m1, m2, m3])
2  val cfs = Frac(MultivariateRing(Z, Array("m1","m2","m3")))
3  // field of rational functions  Frac(Frac(Z[m1, m2, m3])[s, t])
4  implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
5  // parse variables from strings
6  val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

8  val frac = (1 / ((s - t).pow(2) - m3.pow(2))
9    / (s.pow(2) - m1.pow(2))
10   / (t.pow(2) - m2.pow(2)))
11 ...
```

▶ Result:

\[
\frac{1}{((s-t)^2 - m_3^2)(s^2 - m_1^2)(t^2 - m_2^2)} = \\
\frac{1}{8m_1m_2m_3(m_1 + m_2 + m_3)} - \frac{1}{(-m_3 - t + s)(t - m_2)} - \frac{1}{8m_1m_2m_3(m_1 + m_2 + m_3)} + \ldots (+22 \text{ other terms})
\]
Rings: design by examples

Multivariate polynomials & rational functions & simplifications

1 // field of coefficients Frac(Z[m1, m2, m3])
2 val cfs = Frac(MultivariateRing(Z, Array("m1","m2","m3")))
3 // field of rational functions Frac(Frac(Z[m1, m2, m3])[s, t])
4 implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
5 // parse variables from strings
6 val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

8 val frac = (1 / ((s - t).pow(2) - m3.pow(2))
9 // (s.pow(2) - m1.pow(2))
10 // (t.pow(2) - m2.pow(2)))
11 // or just parse from string
12 // val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")

14 // bring in the form with algebraically independent denominators
15 val dec = GroebnerMethods.LeinartDecomposition(frac)
16 // simplify fractions (factorize)
17 val decSimplified = dec.map(f => field.factor(f))
18 // pretty print
19 decSimplified.map(f => field.stringify(f)).foreach(println)
Rings: *design by examples*

Multivariate polynomials & rational functions & simplifications

```scala
1 // field of coefficients Frac(GF(2,16)[m1, m2, m3])
2 val cfs = Frac(MultivariateRing(GF(2,16,"e"), Array("m1","m2","m3")))
3 // field of rational functions Frac(Frac(GF(2,16)[m1, m2, m3])[s, t])
4 implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))
5 // parse variables from strings
6 val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

8 val frac = (1 / ((s - t).pow(2) - m3.pow(2))
9     / (s.pow(2) - m1.pow(2))
10    / (t.pow(2) - m2.pow(2)))
11 // or just parse from string
12 // val frac = field("1/(((s - t)^2 - m3^2)*(s^2 - m1^2)*(t^2 - m2^2))")

14 // bring in the form with algebraically independent denominators
15 val dec = GroebnerMethods.LeinartDecomposition(frac)
16 // simplify fractions (factorize)
17 val decSimplified = dec.map(f => field.factor(f))
18 // pretty print
19 decSimplified.map(f => field.stringify(f)).foreach(println)
```
Rings: design by examples

Multivariate polynomials & rational functions & simplifications

// field of coefficients Frac(GF(2,16)[m1, m2, m3])
val cfs = Frac(MultivariateRing(GF(2,16,"e"), Array("m1","m2","m3")))

// field of rational functions Frac(Frac(GF(2,16)[m1, m2, m3])[s, t])
implicit val field = Frac(MultivariateRing(cfs, Array("s", "t")))

// parse variables from strings
val (m1, m2, m3, s, t) = field("m1", "m2", "m3", "s", "t")

val frac = (1 / ((s - t).pow(2) - m3.pow(2))
  / (s.pow(2) - m1.pow(2))
  / (t.pow(2) - m2.pow(2)))

... Result:
\[
\frac{1}{((s-t)^2 - m_3^2)(s^2 - m_1^2)(t^2 - m_2^2)} =
\frac{1}{(m_1 + m_2 + m_3)^2} \frac{1}{(m_3 + t + s)^2(s + m_1)^2}
\frac{1}{(m_1 + m_2 + m_3)^2} \frac{1}{(m_3 + t + s)^2(t + m_2)^2}
\frac{1}{(m_1 + m_2 + m_3)^2} \frac{1}{(t + m_2)^2(s + m_1)^2}
\]
Rings: parametric number fields

```scala
1  // Q[c, d]
2  val params = Frac(MultivariateRing(Q, Array("c", "d")))
3  // A minimal polynomial X^2 + c = 0
4  val generator = UnivariatePolynomial(params("c"), params(0), params(1))
5  // Algebraic number field Q(sqrt(c)), here "s" denotes square root of c
6  implicit val cfRing = AlgebraicNumberField(generator, "s")
7  // ring of polynomials Q(sqrt(c))(x, y, z)
8  implicit val ring = MultivariateRing(cfRing, Array("x", "y", "z"))
9  // bring variables
10 val (x,y,z,s) = ring("x", "y", "z", "s")
11 // some polynomials
12 val poly1 = (x + y + s).pow(3) * (x - y - z).pow(2)
13 val poly2 = (x + y + s).pow(3) * (x + y + z).pow(4)
14
15 // compute gcd
16 val gcd = PolynomialGCD(poly1, poly2)
17 println(ring stringify gcd)
```