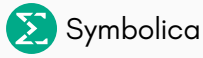
Math for Machines

Ben Ruijl

Nov 04, 2025



Introduction

Computer algebra and particle physics

- One of the first CAS was Schoonschip (1963) by Nobel prize winner Martinus Veltman
- He encountered “a monstrous expression involving 50000 terms in intermediate stages”
- Current cutting edge research in particle physics has expressions with billions of terms that take up terabytes of memory
- The challenge of computer algebra is surviving intermediate expression swell

Modern example of blowup

- Integration-by-parts rules for one-loop vacuum massive bubble:

$$I(n) = \int dk^4 \frac{1}{(k^2 - m^2)^n}$$

$$I(1) = \frac{1}{\varepsilon} + 1$$

$$I(n) = I(n-1)\frac{2 + (4 - 2\varepsilon) - 2n}{(n-1)m^2}$$

- $I(4) = \frac{4}{3} m^{-6}\left(1 + \varepsilon - \varepsilon^2 - \varepsilon^3\right)$
- Multi-loop and multi-mass generalization lead to huge expression blow-up

Four-loop reduction rule to reduce index 14 to 0

```
  id,ifmatch→bubu1,
    Z(n1?pos_,n2?pos_,n3?pos_,n4?pos_,n5?pos_,n6?pos_,n7?pos_,
    n8?pos_,n9?pos_,n10?neg0_,n11?neg0_,n12?neg0_,n13?neg0_,n14?neg_)
       = -rat(1,-2*ep-2*n1-n3-n6-n12-n14+4)*(
+Z(-1+n1,-1+n2,n3,n4,1+n5,n6,n7,n8,n9,n10,n11,n12,n13,1+n14)*rat(-n5,1)
+Z(-1+n1,1+n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,n11,n12,n13,1+n14)*rat(n2,1)
+Z(-1+n1,1+n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,1+n14)*rat(-n2,1)
+Z(-1+n1,n2,1+n3,n4,n5,n6,n7,n8,-1+n9,n10,n11,n12,n13,1+n14)*rat(-n3,1)
+Z(-1+n1,n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,n11,1+n12,n13,1+n14)*rat(-n12,1)
+Z(-1+n1,n2,n3,n4,1+n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,1+n14)*rat(n5,1)
+Z(-1+n1,n2,n3,n4,n5,n6,-1+n7,n8,n9,n10,n11,n12,1+n13,1+n14)*rat(-n13,1)
+Z(-1+n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,1+n12,n13,1+n14)*rat(2*n12,1)
+Z(-1+n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(-1+n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,2+n14)*rat(2*n14+2,1)
+Z(-1+n1,n2,n3,n4,n5,n6,n7,n8,-1+n9,n10,n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(-1+n1,n2,n3,n4,n5,n6,n7,n8,n9,1+n10,-1+n11,n12,n13,1+n14)*rat(-n10,1)
+Z(-1+n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,1+n12,n13,n14)*rat(-n12,1)
+Z(-1+n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,1+n14)*rat(-n2+n5,1)
```

```
+Z(n1,-1+n2,-1+n3,n4,n5,n6,n7,n8,n9,n10,n11,1+n12,n13,1+n14)*rat(n12,1)
+Z(n1,-1+n2,-1+n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,2+n14)*rat(1+n14,1)
+Z(n1,-1+n2,1+n3,n4,n5,n6,n7,n8,-1+n9,n10,n11,n12,n13,1+n14)*rat(n3,1)
+Z(n1,-1+n2,n3,-1+n4,n5,n6,n7,n8,n9,n10,1+n11,n12,n13,1+n14)*rat(n11,1)
+Z(n1,-1+n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,n11,1+n12,n13,1+n14)*rat(n12,1)
+Z(n1,-1+n2,n3,n4,n5,-1+n6,n7,n8,n9,n10,n11,n12,n13,2+n14)*rat(-n14-1,1)
+Z(n1,-1+n2,n3,n4,n5,n6,-1+n7,1+n8,n9,n10,n11,n12,n13,1+n14)*rat(2*n8,1)
+Z(n1,-1+n2,n3,n4,n5,n6,-1+n7,n8,n9,n10,1+n11,n12,n13,1+n14)*rat(-n11,1)
+Z(n1,-1+n2,n3,n4,n5,n6,-1+n7,n8,n9,n10,n11,n12,1+n13,1+n14)*rat(-n13,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,1+n12,n13,1+n14)*rat(-2*n12,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,2+n14)*rat(-2*n14-2,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,1+n8,-1+n9,n10,n11,n12,n13,1+n14)*rat(-n8,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,n8,-1+n9,n10,n11,1+n12,n13,1+n14)*rat(-2*n12,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,n8,n9,1+n10,-1+n11,n12,n13,1+n14)*rat(n10,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,n8,n9,1+n10,n11,n12,n13,1+n14)*rat(-2*n10,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,n8,n9,n10,1+n11,n12,n13,1+n14)*rat(-n11,1)
+Z(n1,-1+n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,1+n12,n13,n14)*rat(-n12,1)
```

```
+Z(n1,1+n2,n3,n4,-1+n5,n6,n7,n8,n9,-1+n10,n11,n12,n13,1+n14)*rat(-n2,1)
+Z(n1,1+n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,n11,n12,n13,1+n14)*rat(-n2,1)
+Z(n1,1+n2,n3,n4,n5,n6,n7,-1+n8,n9,-1+n10,n11,n12,n13,1+n14)*rat(2*n2,1)
+Z(n1,1+n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,1+n14)*rat(n2,1)
+Z(n1,n2,-1+n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,1+n12,n13,1+n14)*rat(-n12,1)
+Z(n1,n2,-1+n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,2+n14)*rat(-n14-1,1)
+Z(n1,n2,1+n3,n4,n5,n6,n7,n8,-1+n9,-1+n10,n11,n12,n13,1+n14)*rat(n3,1)
+Z(n1,n2,n3,n4,-1+n5,n6,-1+n7,n8,n9,n10,n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(n1,n2,n3,n4,-1+n5,n6,n7,n8,-1+n9,n10,n11,n12,1+n13,1+n14)*rat(-n13,1)
+Z(n1,n2,n3,n4,-1+n5,n6,n7,n8,-1+n9,n10,n11,n12,n13,2+n14)*rat(1+n14,1)
+Z(n1,n2,n3,n4,-1+n5,n6,n7,n8,n9,-1+n10,n11,1+n12,n13,1+n14)*rat(n12,1)
+Z(n1,n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,-1+n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(n1,n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,n11,1+n12,n13,1+n14)*rat(n12,1)
+Z(n1,n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(n1,n2,n3,n4,-1+n5,n6,n7,n8,n9,n10,n11,n12,n13,1+n14)*rat(-n2+n8-n13,1)
+Z(n1,n2,n3,n4,n5,-1+n6,n7,-1+n8,n9,n10,n11,n12,n13,2+n14)*rat(1+n14,1)
+Z(n1,n2,n3,n4,n5,n6,-1+n7,-1+n8,n9,n10,1+n11,n12,n13,1+n14)*rat(n11,1)
+Z(n1,n2,n3,n4,n5,n6,-1+n7,-1+n8,n9,n10,n11,n12,1+n13,1+n14)*rat(-2*n13,1)
```

```
+Z(n1,n2,n3,n4,n5,n6,-1+n7,n8,n9,-1+n10,1+n11,n12,n13,1+n14)*rat(n11,1)
+Z(n1,n2,n3,n4,n5,n6,-1+n7,n8,n9,n10,n11,-1+n12,n13,2+n14)*rat(1+n14,1)
+Z(n1,n2,n3,n4,n5,n6,-1+n7,n8,n9,n10,n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(n1,n2,n3,n4,n5,n6,-1+n7,n8,n9,n10,n11,n12,n13,1+n14)*rat(-2*ep-2*n4-1,1)
+Z(n1,n2,n3,n4,n5,n6,1+n7,-1+n8,n9,n10,n11,n12,n13,1+n14)*rat(-n7,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,-1+n9,n10,n11,n12,1+n13,1+n14)*rat(n13,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,-1+n9,n10,n11,n12,n13,2+n14)*rat(-2*n14-2,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,-1+n10,1+n11,n12,n13,1+n14)*rat(-n11,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,-1+n10,n11,1+n12,n13,1+n14)*rat(-2*n12,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,-1+n10,n11,n12,n13,2+n14)*rat(-2*n14-2,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,1+n10,n11,n12,n13,1+n14)*rat(2*n10,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,-1+n11,n12,1+n13,1+n14)*rat(-2*n13,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,1+n11,n12,n13,1+n14)*rat(n11,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,-1+n12,n13,2+n14)*rat(-n14-1,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,1+n12,n13,1+n14)*rat(-2*n12,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,1+n13,1+n14)*rat(-3*n13,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,1+n14)*rat(10*ep+2*n1,1)
+Z(n1,n2,n3,n4,n5,n6,n7,-1+n8,n9,n10,n11,n12,n13,2+n14)*rat(-2*n14-2,1)
```

```
+Z(n1,n2,n3,n4,n5,n6,n7,n8,-1+n9,-1+n10,n11,1+n12,n13,1+n14)*rat(-n12,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,-1+n9,-1+n10,n11,n12,1+n13,1+n14)*rat(-n13,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,-1+n9,n10,-1+n11,n12,n13,2+n14)*rat(-n14-1,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,-1+n9,n10,n11,n12,1+n13,1+n14)*rat(-n13,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,-1+n9,n10,n11,n12,n13,1+n14)*rat(-4*ep-2*n1-n3,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,n9,-1+n10,n11,n12,n13,1+n14)*rat(-n5+1,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,n9,1+n10,-1+n11,n12,n13,1+n14)*rat(n10,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,-1+n11,n12,n13,1+n14)*rat(2*ep+n5+2*n8
    +n9+n10+n11-5,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,1+n12,n13,n14)*rat(n12,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,-1+n13,1+n14)*rat(-2*ep-n5-2*n8
    -n9-n11-n14+3,1)
+Z(n1,n2,n3,n4,n5,n6,n7,n8,n9,n10,n11,n12,n13,1+n14)*rat(2*ep+n2+n7+
    2*n8+n9+n11+n14-4,1)
);
```

Why understand how to compute?

- By understanding **how** computations are performed you can formulate your programs in a way that they are more efficient and more reliable
- Often requires domain knowledge on top
-

Why understand how to compute?

- By understanding **how** computations are performed you can formulate your programs in a way that they are more efficient and more reliable
- Often requires domain knowledge on top
- For IBPs: reduction time is dominated by simplification of rational functions using GCDs
  - Compute knowledge: Avoid GCDs by factoring denominators
  - Domain knowledge: there are factors of space-time dimension d
  - Remove simple common factors early:

$$d + d^2 + x + dx + m^2 + dm^2 + z + dz = (1 + d)\left(d + m^2 + x + z\right)$$

  - Compute knowledge: you can factor with one easy single univariate polynomial GCD

Why understand how to compute?

- Know how computer algebra works means you can tell if following class of questions are easy or hard:
  - Is $\sin(x)^2 + \cos(x)^2 + e^{\pi i}$ zero?
  - What is an antiderivative of $\frac{x^{800}+x^3+4}{x^4+x+1}$?
    - And of $e^x\left(\frac{1}{1+x} + \log(1+x)\right)$?
  - What is the factorization of $1 + 4x + 6x^2 + 4x^3 + x^4 + 4y + 12xy + 12x^2y + 4x^3y + 6y^2 + 12xy^2 + 6x^2y^2 + 4y^3 + 4xy^3 + y^4$?
  - When is $x^2 + y^6 + xy - 3 = 0$ and $x^4 + y^2 + x^2y - 5 = 0$?
    - Are there rational solutions?

Why understand how to compute?

- Bottlenecks are introduced by using the wrong data representation
  - Expanding too early
  - Suboptimal algorithms
- Each data structure has its own normalization and tradeoffs
- A polynomial always expands (in Symbolica), but has faster multiplication and GCD operations
- Prevent many conversions (from general expression to polynomial, etc.)

Goals of computer algebra

- Find reliable methods to perform mathematics
- Create abstractions that prevent double work
- Understand computational bottlenecks

Goals for this course

- An overview of the main algorithms and ideas

Algebra vs analysis

- Analysis is not so useful for exact computations
- $\mathbb{R}$ or $\mathbb{C}$ cannot be described on a computer
  - $1/2 + 1/4 = 3/4$
  - $\sqrt{2} + \sqrt{3} = \mathrm{root}\left(x^4 - 10x^2 + 1, 4\right)$
  - $e + \pi$ =???
- Relations of transcendental numbers not known: is $e + \pi$ rational?

Workaround:

- Assume algebraic independence (addition, multiplication, etc)
- Treat transcendental numbers as independent variables
- Thus: $e + \pi$ stays as-is in our expressions

Why is computer algebra hard?

- Let us look at a simple example: row reduction
- Used for solving linear systems

$$\begin{pmatrix} 6. & 5. \\ 2. & 3. \end{pmatrix} \to \begin{pmatrix} 6. & 5. \\ 0. & 1.3333 \end{pmatrix}$$

- Many linear algebra packages around for this
-

Why is computer algebra hard?

- Let us look at a simple example: row reduction
- Used for solving linear systems

$$\begin{pmatrix} 6. & 5. \\ 2. & 3. \end{pmatrix} \to \begin{pmatrix} 6. & 5. \\ 0. & 1.3333 \end{pmatrix}$$

- Many linear algebra packages around for this
- Let us try to do it exactly:

$$\begin{pmatrix} 6 & 5 \\ 2 & 3 \end{pmatrix} \to \begin{pmatrix} 6 & 5 \\ 0 & ? \end{pmatrix}$$

$$3 - \frac{2}{6}5 = \frac{3}{1} - \frac{10}{6} = \frac{3 \cdot 6 - 10 \cdot 1}{1 \cdot 6} = \frac{8}{6}$$

Euclidean algorithm

- How to get to 4/3?
- We need to find the greatest common divisor of 8 and 6

$$\gcd(a, b) = \begin{cases} a \text{ if } b = 0 \\ \gcd(b, a \bmod b) = 0 \text{ else} \end{cases}$$

$\gcd(8, 6) = \gcd(6, 2) = \gcd(2, 0) = 2$

$$\frac{8/2}{6/2} = \frac{4}{3}$$

- The complexity is rising..

Why is computer algebra hard?

- What if we add a parameter?

$$\begin{pmatrix} t^2+3t+2 & 1+t \\ 2+t & t+4 \end{pmatrix} \to \begin{pmatrix} t^2+3t+2 & 1+t \\ 0 & ? \end{pmatrix}$$

$$t+4-\frac{2+t}{t^2+3t+2}(1+t) = \frac{t^3+6t^2+11t+6}{t^2+3t+2}$$

Why is computer algebra hard?

- Euclidean algorithm also works on univariate polynomials over rationals

$$\gcd\left(t^3 + 6t^2 + 11t + 6, t^2 + 3t + 2\right) = t^2 + 3t + 2$$

- Algorithms stay the same but $+$, $*$ and $/$, etc are different operations now
- First hint of abstraction

Euclidean algorithm

$$a = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$$
$$b = 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

Euclidean algorithm

$$a = x^8 + x^6 - 3x^4 - 3x^3 + 8x^2 + 2x - 5$$
$$b = 3x^6 + 5x^4 - 4x^2 - 9x + 21$$

$$a \bmod b = r_1 = -\frac{5}{9}x^2 + \frac{1}{9}x^2 - \frac{1}{3}$$

$$b \bmod r_1 = r_2 = -\frac{117}{25}x^2 - 9x + \frac{411}{25}$$

$$r_1 \bmod r_2 = r_3 = -\frac{102500}{6591} + \frac{233150}{19773}x$$

$$r_2 \bmod r_3 = r_4 = -\frac{1288744821}{543589225}$$

- Coefficients are rational numbers now...

Euclidean algorithm

An attempt with pseudo-remainders:

$$r_1 = -15x^4 + 3x^2 - 9$$

$$r_2 = 15795x^2 + 30375x - 5953$$

$$r_3 = 1254542875143750x - 1654608338437500$$

$$r_4 = 1259333879550074310093114199218750 0$$

Euclidean algorithm

An attempt with pseudo-remainders:

$$r_1 = -15x^4 + 3x^2 - 9$$

$$r_2 = 15795x^2 + 30375x - 5953$$

$$r_3 = 1254542875143750x - 1654608338437500$$

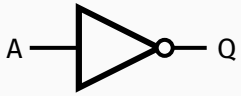
$$r_4 = 12593338795500743100931141992187500$$

- Coefficients still integer, but huge (> `i64`)
- Euclidean algorithm does not apply for multivariate polynomials

We start from scratch

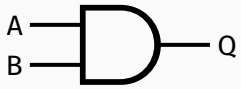
Logical gates

- Our computer has the following logical gates:



NOT: ¬*A*

A	*Q*
0	1
1	0



AND: $A \wedge B$

A	*B*	*Q*
0	0	0
0	1	0
1	0	0
1	1	1

OR: $A \vee B$

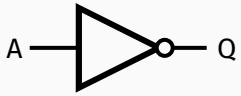
A	*B*	*Q*
0	0	0
0	1	1
1	0	1
1	1	1

XOR: $A \oplus B$

A	*B*	*Q*
0	0	0
0	1	1
1	0	1
1	1	0

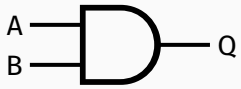
Logical gates

- Our computer has the following logical gates:



NOT: $\neg A$

A	Q
0	1
1	0



AND: $A \wedge B$

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

OR: $A \vee B$

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

XOR: $A \oplus B$

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

- $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$
- How to do integer arithmetic?

Number representations

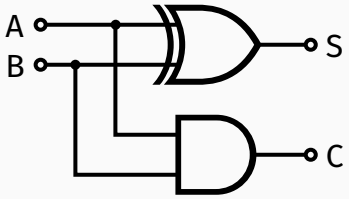
The decimal number 12345679 can be written as:

$$1 \cdot 10^7 + 2 \cdot 10^6 + 3 \cdot 10^5 + 4 \cdot 10^4 + 5 \cdot 10^3 + 6 \cdot 10^2 + 7 \cdot 10 + 9$$

- Polynomial in the base $b = 10$
- every coefficient is in $[0, b)$
- Computers use base 2, binary

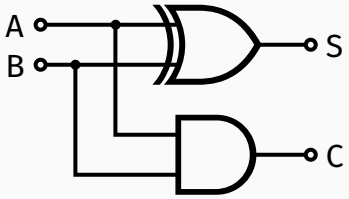
$$12345679_{10} = 101111000110000101000111_2 = \mathrm{BC614F}_{16}$$

Addition

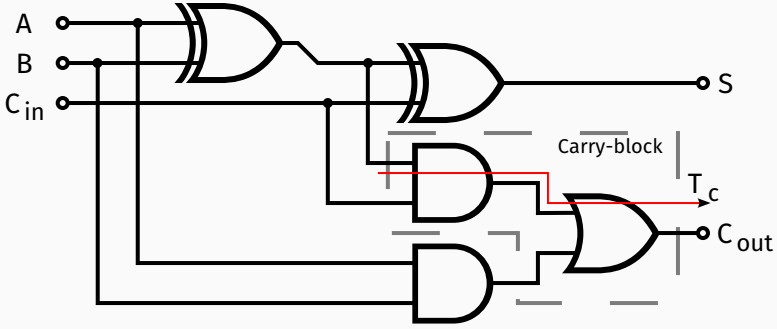


- Add bits A and B and store in S
- Carry bit stored in C

Addition

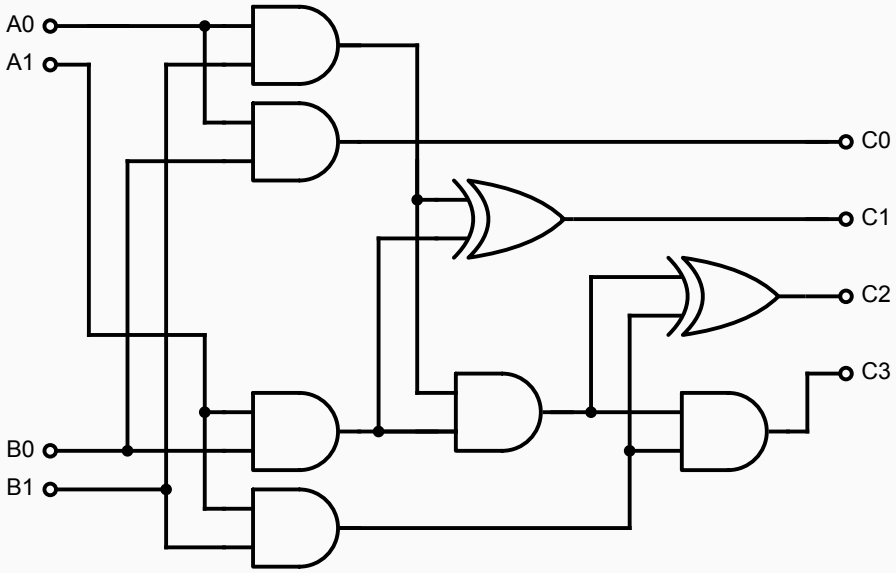


- Add bits A and B and store in S
- Carry bit stored in C



- Add carry bit as input
- Can be chained!
- Modern machines chain 64 adders

Multiplication



- 2-bit by 2-bit binary multiplier
- $A \cdot B = A_0B_0 + (A_0B_1 + A_1B_0)2 + A_1B_1 2^2$
- Contains adder as subcomponent

64-bit base

- 64-bit machines have 64-bit `add` and `mul` operations
- Reinterpret base 2 as 2^{64} by partitioning
  - Example: number in base 2 as base 2^4:

```
base 2: 00010111100011000010100011111
base 4: 0001 0111 1000 1100 0010 1000 1111
```

- How to support larger integers?

Arbitrary precision

- Data structure must be able to grow

```
struct Integer {
    chunks: Vec<u64>,
    sign: bool,
}
```

- The first chunk is the lowest significant
- Overload operators for easy use

```
impl Add for Integer {
    type Output = Integer;
    fn add(self, other: Integer) → Integer {}
}
```

Addition

- Add chunks together and check if carry flag is set
- Sketch:

```
fn add(a: &Integer, b: &Integer) → Integer {
    let mut carry = false;
    let mut chunks = vec![];
    for (c1, c2) in a.chunks.iter().zip(&b.chunks) {
        let (mut res, mut new_carry) = c1.overflowing_add(*c2);
        if carry {
            let (r, c) = res.overflowing_add(1);
            res = r;
            new_carry |= c;
        }
        chunks.push(res);
        carry = new_carry;
    }
    if carry { chunks.push(1); }
}
```

Multiplication

- Multiplication requires double the working precision
- One solution: use base 2^{32} instead of 2^{64}
- Partition number as 2 u32s

$$a = a_0 + a_1 2^{32}$$

$$b = b_0 + b_1 2^{32}$$

- Multiplication:

$$a \cdot b = a_0 b_0 + \left(a_0 b_1 + a_1 b_0\right) 2^{32} + a_1 b_1 2^{64}$$

Multiplication of two 64-bit integers using 32-bit arithmetic

```
fn mul128_on_32(a: u64, b: u64) → (u64, u64) {
  const MASK: u64 = 0×FFFFFFFF; // extract lower 32 bits

  let (a_low, a_high) = (a & MASK, a >> 32); // split in 32-bit parts
  let (b_low, b_high) = (b & MASK, b >> 32);

  let (p_ll, p_lh, p_hl, p_hh) = (a_low * b_low, a_low * b_high, a_high * b_low, a_high * b_high);

  let r0 = p_ll & MASK; // bits 1-32

  let col1 = (p_ll >> 32) + (p_lh & MASK) + (p_hl & MASK);
  let r1 = col1 & MASK; // bits 33-64

  let col2 = (col1 >> 32) + (p_lh >> 32) + (p_hl >> 32) + (p_hh & MASK);
  let r2 = col2 & MASK; // bits 65-96

  let r3 = (col2 >> 32) + (p_hh >> 32); // bits 97-128

  ((r1 << 32) | r0, (r3 << 32) | r2)
}
```

Arbitrary-precision multiplication

```
fn mul(a: &Integer, b: &Integer) → Integer {
    let mut res = Integer::zero();
    for (i, s) in a.chunks.iter().enumerate() {
        for (j, t) in b.chunks.iter().enumerate() {
            let (lo, hi) = mul128_on_32(*s, *t);
            res.shift_add(lo, i + j); // res += lo * 2^64*(i + j)
            res.shift_add(hi, i + j + 1);
        }
    }
    res
}
```

Parsing

Parse a base-10 string representation of an integer:

```
fn parse(input: &str) → Integer {
    let mut out = Integer::zero();
    for c in input.chars() {
        out = out * 10 + (c-'0') as u64
    }
    out
}
```

Floating-point numbers

- An approximation of a real number whose decimal point can move
- Represented as ± mantissa · $\text{base}^{\text{exponent}}$
- Mantissa and exponent have fixed number of digits
-

-

Floating-point numbers

- An approximation of a real number whose decimal point can move
- Represented as ± mantissa · $\text{base}^{\text{exponent}}$
- Mantissa and exponent have fixed number of digits
- `double`: base 2, 1 bit sign, 52 bits mantissa, 11 bits signed exponent
  - All integers up to $\pm 2^{53}$ exactly representable
- Example: base 10, 2-digit mantissa
  - 3.2 + 4.1 = 7.3 (no precision loss)
  - 6.2 + 7.1 = 13.3 = 1.3 · 10 (decimal point move, precision loss of 2.3%)
  - $2.1 \cdot 3.2 = 21 \cdot 32 \cdot 10^{-2} = 672 \cdot 10^{-2} = 6.72 = 6.7$ (small precision loss of 0.3%)
  - Additional precision loss due to conversion to floating point ($\frac{1}{3} = 3.3 \cdot 10^{-1}$)

Properties of integers

- Integers have addition, subtraction, multiplication
- If $x \cdot y = 0$, then $x = 0$ or $y = 0$
- Integers have two numbers with an inverse: 1 and –1
- Integers can be uniquely decomposed in irreducible integers: primes
  - Example: $72364 = 2^2 \cdot 79 \cdot 229$
- Two integers are co-prime when they share no primes
- Division with remainder of x by y: $x = qy + r$ with $|r| < |y|$
  - Example: $23 \div 7 = 3 \cdot 7 + 2$

Generalize properties of integers to other integer-like structures

Domains

Various levels of abstraction:

rings $\supset$ commutative rings $\supset$ integral domains $\supset$ integrally closed domains $\supset$ GCD domains $\supset$ unique factorization domains $\supset$ principal ideal domains $\supset$ Euclidean domains $\supset$ fields $\supset$ algebraically closed fields

Operations needed for row reduction

A reduction step:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \to \begin{pmatrix} A & B \\ 0 & D - \frac{C}{A}B \end{pmatrix}$$

Operations needed for row reduction

A reduction step:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \to \begin{pmatrix} A & B \\ 0 & D - \frac{C}{A}B \end{pmatrix}$$

- Find non-zero pivot: `is_non_zero()` (ring)
- Divide two elements of the same type: `div` (field)
- Multiply two elements of the same type: `mul` (ring)
- Subtract two elements of the same type: `sub` (ring)

Ring

- A ring R is a set with addition, subtraction, multiplication operators
- Example: integers, polynomials over rings, square matrices over rings
- 0 is the neutral element of addition, 1 of multiplication
- A **unit** is an element with an inverse: $a \in R, \exists b \in R : ab = 1$
- For integers, the units are 1 and –1

Rings

Ring

- A ring R is a set with addition, subtraction, multiplication operators
- Example: integers, polynomials over rings, square matrices over rings
- 0 is the neutral element of addition, 1 of multiplication
- A **unit** is an element with an inverse: $a \in R, \exists b \in R : ab = 1$
- For integers, the units are 1 and –1

Commutative ring

- A ring where the multiplication operator is commutative

Integral domain

- A commutative ring where $a \cdot b = 0 \Rightarrow a = 0 \vee b = 0$
- An **irreducible** element cannot be written as a product of two non-units
- A **prime** is a non-unit element with $p \mid ab \Rightarrow p \mid a \vee p \mid b$

Unique factorization domain (UFD)

- An integral domain where every non-zero non-unit element a can factored:

$$a = up_1^{e_1} p_2^{e_2} \ldots p_n^{e_n}$$

where u is a unit, p_i are irreducible and prime

- The factorization is unique up to units
- Prime and irreducible elements are the same
- Examples: integers, Gaussian integers

Greatest common divisors

- A greatest common divisor (GCD) is unique in a UFD
- Simply collect all common irreducible elements
- If R is UFD, then a polynomial with coefficients in R, $R[x]$, is too
- $R[x_1, ..., x_n] = ((R[x_1])[x_2])...[x_n]$ therefore multivariate polynomials over R

are UFDs

Euclidean domain

- A UFD with a Euclidean function $d : R \to \mathbb{N}$
- A division with remainder operation $a, b \in R, b \neq 0$:

$$\exists q, r \in R : a = bq + r, d(r) < d(b)$$

- Examples:
  - Integers: $d(i) = |i|$
  - Polynomials: $d(p) = \deg(p)$
  - Gaussian integers: $d(a + bi) = a^2 + b^2$
-
-
-

Euclidean domain

- A UFD with a Euclidean function $d : R \to \mathbb{N}$
- A division with remainder operation $a, b \in R, b \neq 0$:

$$\exists q, r \in R : a = bq + r, d(r) < d(b)$$

- Examples:
  - Integers: $d(i) = |i|$
  - Polynomials: $d(p) = \deg(p)$
  - Gaussian integers: $d(a + bi) = a^2 + b^2$
- $\gcd(a, b) = \gcd(b, r)$
  - $g = \gcd(a, b), h = \gcd(b, r)$
  - $a = bq + r \Rightarrow h \mid a \Rightarrow h \mid g$, and $r = a - bq \Rightarrow g \mid r \Rightarrow g \mid h \Rightarrow g \sim h$
- $\gcd(a, b) = \gcd(b, a)$
- $\gcd(a, 0) = a$

Euclidean algorithm

- $\text{quot}(a, b) = q, \text{rem}(a, b) = r$
- Repeated division with remainder yields:

$$\gcd(a, b) = \begin{cases} a \text{ if } b = 0 \\ \gcd(b, \text{rem}(a, b)) \end{cases}$$

Euclidean algorithm

```
def euclidean_algorithm(a, b):
    r = [a, b]
    while r[-1] ≠ 0:
        q = r[-2] // r[-1]
        r.append(r[-2] - q * r[-1])
    return r[-2]
```

Euclidean algorithm

```python
def euclidean_algorithm(a, b):
    r = [a, b]
    while r[-1] ≠ 0:
        q = r[-2] // r[-1]
        r.append(r[-2] - q * r[-1])
    return r[-2]
```

- Recurrence:

  $r_0 = a$

  $r_1 = b$

  $r_2 = r_0 - q_2 r_1 = a - q_2 b$

  $r_3 = r_1 - q_3 r_2 = b - q_3(a - q_2 b) = (-q_3)a + (1 + q_2 q_3)b$

- We iterate until we get $\gcd(a, b) = r_i = s_i \cdot a + t_i \cdot b$
- The GCD is a linear combination of the inputs!

Extended Euclidean algorithm

```python
def extended_euclidean_algorithm(a, b):
    r = [a, b]
    s = [1, 0]
    t = [0, 1]
    while r[-1] != 0:
        q = r[-2] // r[-1]
        r.append(r[-2] - q * r[-1])
        s.append(s[-2] - q * s[-1])
        t.append(t[-2] - q * t[-1])
    return r[-2], s[-2], t[-2]
```

- With $r_i = s_i \cdot a + t_i \cdot b$
- Arrays are added for clarity, in practice we only need to store the last two values

Fields

- Every non-zero element is a unit
- Division operator for all non-zero elements
- Specifying the domain is important!
  - 2 is prime in $\mathbb{Z}$, a unit in $\mathbb{Q}$ and factors as $(-i)(1+i)^2$ in $\mathbb{Z}(i)$

Algebraically closed fields

- A field $\mathbb{F}$ where all roots of all polynomials in $\mathbb{F}$ lie in $\mathbb{F}$
- In $\mathbb{F}$ the polynomials split:

$$p(x) = c(x - r_1)(x - r_2)...(x - r_n)$$

where $c, r_i \in \mathbb{F}$

- Every field has a larger algebraically closed field (every $\mathbb{R}[x]$ splits in $\mathbb{C}$)
- All solutions $p(r_i) = 0$ with $r_i \in \mathbb{F}$ are linear factors, as we can write

$$p(x) = r(x) \prod_i (x - r_i)$$

- Example: $x^3 + x^2 + x + 1$
  - Factors in $\mathbb{Q}[x]$ as $(x + 1)(x^2 + 1)$
  - Only rational solution: $x = -1$
  - Factors in $Q(i)[x]$ as $(x + 1)(x - i)(x + i)$

Normal form

- Every element in the ring can be written as $x = \text{unit}(x) \cdot \text{normal}(x)$
- The normal form is a convention
- For integers, $\text{normal}(b) = |b|$: $\text{normal}(-6) = 6$
- For polynomials $p \in R[x]$, $\text{normal}(p) = \frac{p}{\text{unit}(\text{lc}_x(p))}$
  - $\text{normal}\left(1 + 2x - 3x^2\right) = -1 - 2x + 3x^2$ in $\mathbb{Z}[x]$
  - $\text{normal}\left(1 + 2x - 3x^2\right) = -\frac{1}{3} - \frac{2}{3}x + x^2$ in $\mathbb{Q}[x]$
- Unique GCD: $\gcd(a, 0) = \text{normal}(a)$

Let’s create new rings

Ring of fractions

- The ring of fractions K_R is a field if R is an integral domain
- Addition:

$$\frac{a}{s} + \frac{b}{t} = \frac{at + bs}{st}$$

- Multiplication:

$$\frac{a}{s} \cdot \frac{b}{t} = \frac{ab}{st}$$

- For a Euclidean domain R, we can simplify the fraction
- Unique representation of $\frac{n}{d}$ by forcing $\gcd(n, d) = 1$ and d normalized
- Product and sum of two normalized fractions is normalized!
- Inverse needs normalization, e.g. $\left(\frac{-2}{5}\right)^{-1} = \left(\frac{5}{-2}\right)$

Ring of fractions II

- Rational numbers: $\mathbb{Q} = K_{\mathbb{Z}}$
- Convention for K_R:

$$\gcd\left(\frac{a}{b}, \frac{c}{d}\right) = \frac{\gcd(a,c)}{\operatorname{lcm}(b,d)} = g$$

- As a result $(a \div b) \div g$ and $(c \div d) \div g$ yield elements in R
- The fields $K_{R[x]}$ and $K_{K_R[x]}$ are isomorphic: $\frac{5+10x}{1+6x} = \frac{\frac{5}{6}+\frac{5}{3}x}{\frac{1}{6}+x}$

Efficient fraction arithmetic

- Multiplying $\frac{a}{s} \cdot \frac{b}{t} = \frac{ab}{st}$ and dividing by $g = \gcd(ab, st)$ yields large intermediate expressions:

$$\frac{(ab) \div g}{(st) \div g}$$

- Faster: $g_1 = \gcd(a, t)$, $g_2 = \gcd(b, s)$

$$\frac{a}{s} \cdot \frac{b}{t} = \frac{\left(\frac{a}{g_1}\right)\left(\frac{b}{g_2}\right)}{\left(\frac{s}{g_2}\right)\left(\frac{t}{g_1}\right)}$$

Efficient fraction addition

```
def add(a: Fraction, b: Fraction) → Fraction:
  denom_gcd = a.den.gcd(b.den)
  a_den_red = a.den / denom_gcd
  b_den_red = b.den / denom_gcd
  num = a.num * b_den_red + b.num * a_den_red
  den = b_den_red * a.den
  g = num.gcd(den)
  return Fraction(num / g, den / g)
```

Finite fields

- Finite fields are fields with a finite number of elements
- All finite fields with n elements are isomorphic
- For what n can we construct a finite field?
  - For what n can we invert all elements?
  - Consider ring $\mathbb{Z}_p$ with $+ : a + b \bmod p$ and $* : ab \bmod p$

Modular inversion

- We want to find s such that $sx \equiv 1 \bmod p$

$$sx \equiv 1 \bmod p \Rightarrow sx + tp = 1$$

- Extended Euclidean algorithm gives the solution if $\gcd(x, p) = 1$
- $s \equiv x^{-1} \bmod p$ and $t \equiv p^{-1} \bmod x$
- If p is prime, $\gcd(x, p) = 1$ holds for all $0 < x < p$
- Thus, $\mathbb{Z}_p = \{0, \ldots, p - 1\}$ is a field with p elements
- Symmetric representation: $\mathbb{Z}_p = \left\{-\frac{p-1}{2}, \ldots, 0, \ldots, \frac{p-1}{2}\right\}$
- Example: $2 \cdot 4 = 3 = -2$ in $\mathbb{Z}_5$

Galois fields

- EEA holds for polynomials with coefficients in a field as well
- Choose irreducible polynomial $f(x) \in \mathbb{Z}_p[x]$ with degree n
- Then $\mathbb{Z}_{p,f}[x]$ with $+ : a + b \bmod f(x), * : a \cdot b \bmod f(x)$ is a field with p^n elements
- Also written as $\mathbb{F}_{p^n}$ or $\mathrm{GF}(p, n)$
-

Galois fields

- EEA holds for polynomials with coefficients in a field as well
- Choose irreducible polynomial $f(x) \in \mathbb{Z}_p[x]$ with degree n
- Then $\mathbb{Z}_{p,f}[x]$ with $+ : a + b \bmod f(x), * : a \cdot b \bmod f(x)$ is a field with p^n elements
- Also written as $\mathbb{F}_{p^n}$ or $\mathrm{GF}(p, n)$
- Example: $\mathbb{F}_8 = \mathrm{GF}(2, 3)$ is a field with $2^3 = 8$ elements
  - Represented as polynomials over $\mathbb{Z}_2$ with degree less than 3.
  - $x^3 + x + 1$ is an irreducible polynomial in $\mathbb{Z}_2[x]$
  - $\mathrm{GF}(2, 3) = \left\{0, 1, x, x + 1, x^2, x^2 + 1, x^2 + x, x^2 + x + 1\right\}$

Fast finite field arithmetic

- In practice we can use half machine-word sized primes ($\approx 2^{31}$)
- Convert from integer: $x \to x \bmod p$
- Addition: $x + y = (x + y) \bmod p$
- Multiplication: $x \cdot y = (x \cdot y) \bmod p$
- Inverse of x: obtain from $\gcd(x, p)$
- Mod is a slow operation, so we try to avoid it

```
def add(x, y, p):
  r = x + y # fits in machine word
  return r if r < p else r - p
```

Fast finite field arithmetic

- Multiplication is harder and we need a mod, naively:

```
fn reduce(a: u32, p: u32) → u32 { // yields a % p
  let q = a / p;
  a - q * p
}
```

- Try to exchange division by p by bitshift (div 2)
- Precompute $m = \left\lfloor \frac{2^{32}}{p} \right\rfloor$
- Then $q = a \cdot \frac{m}{2^{32}}$ is either the correct divisor or one too small

```
fn reduce(a: u64, m: u32, p: u32) → u32 {
  let q = (a * m as u64) >> 32;
  let r = a - q as u32 * p;
  if r ⩾ p {  r - p } else { r }
}
```

Miller-Rabin prime test

- Fermat: if $a^{n-1} \equiv 1 \bmod n$ for all $a \Leftrightarrow n$ is prime
- Additionally: the only square roots of 1 are 1 and –1
- Let odd $n > 2$ with $n - 1 = 2^s d$, sample a in $2 < a < n - 2$
- Then $a^{n-1} = a^{2^s d} = \left(a^d\right)^{2^s} \equiv 1 \bmod n$
- If $\left(a^d\right)^{2^r} \equiv 1 \bmod n$ for $r > 0$, then $\left(a^d\right)^{2^{r-1}}$ must be $\pm 1 \bmod n$
- If it holds for every a, n is prime
  - Chance is $\frac{1}{4}$ that composite n passes the test for a random a
  - Try several a

Miller-Rabin primality test

```
def prime_test(n, k):
  s, d = 0, n - 1
  while d % 2 == 0:
      s += 1
      d //= 2

  for _ in range(k):
      a = random(2, n - 2)
      x = pow(a, d, n) # fast binary powering for a^d mod n
      for _ in range(s):
          y = pow(x, 2, n)
          if y == 1 and x ≠ 1 and x ≠ n - 1:
              return False # square root of 1 different from ±1
          x = y
      if x ≠ 1:
          return False # a^(n-1) ≠ 1 mod n
  return True
```

Miller-Rabin primality test

- For small n, the test can be made deterministic by choosing a particular set of a
- The set $\{2, 325, 9375, 28178, 450775, 9780504, 1795265022\}$ is sufficient for $n < 2^{64}$
- Essentially a compression of $425\,656\,284\,035\,217\,743 \left(\approx 2^{59}\right)$ primes below 2^{64}
- Gives no information about the factors of composite n

Coding mathematical abstractions

Translating the abstractions into code

Abstract over types that implement the required operations

- Rust: traits
- Java: interfaces
- Python: duck typing
- C++: concepts
- Haskell: type classes

Rings

```
pub trait Ring {
    type Element: Clone + PartialEq + Eq;

    fn add(&self, a: &Self::Element, b: &Self::Element) → Self::Element;
    fn sub(&self, a: &Self::Element, b: &Self::Element) → Self::Element;
    fn mul(&self, a: &Self::Element, b: &Self::Element) → Self::Element;
    fn neg(&self, a: &Self::Element) → Self::Element;
    fn zero(&self) → Self::Element;
    fn one(&self) → Self::Element;
    fn nth(&self, n: u64) → Self::Element;
    fn pow(&self, b: &Self::Element, e: u64) → Self::Element;
    fn is_zero(a: &Self::Element) → bool;
    fn is_one(&self, a: &Self::Element) → bool;
}
```

- Every ring has an associated type `Element` that represents the elements of the ring

Euclidean domain

```
pub trait EuclideanDomain: Ring {
    fn rem(&self, a: &Self::Element, b: &Self::Element) → Self::Element;
    fn quot_rem(&self, a: &Self::Element, b: &Self::Element) →
                                            (Self::Element, Self::Element);
    fn gcd(&self, a: &Self::Element, b: &Self::Element) → Self::Element;
}
```

Field

```rust
pub trait Field: EuclideanDomain {
    fn div(&self, a: &Self::Element, b: &Self::Element) -> Self::Element;
    fn div_assign(&self, a: &mut Self::Element, b: &Self::Element);
    fn inv(&self, a: &Self::Element) -> Self::Element;
}
```

Integers

```
pub struct Z {} // empty structure representing the ring of integers

impl Ring for Z {
    type Element = Integer; // our arbitary-precision integer
    fn add(&self, a: &Integer, b: &Integer) → Integer {
        a + b
    }
    fn sub(&self, a: &Integer, b: &Integer) → Integer {
        a - b
    }
    fn mul(&self, a: &Integer, b: &Integer) → Integer {
        a * b
    }
}
```

Finite fields

```
pub struct Zp {
  p: u32, // prime
  m: u32, // precomputed value
}

impl Ring for Zp {
  type Element = u32;
  fn mul(&self, a: u32, b: u32) → u32  {
      self.reduce(a as u64 * b as u64)
  }
}

impl EuclideanDomain for Zp { ... }
impl Field for Zp { ... }
```

Abstraction example: polynomials

```
struct Polynomial<T: Ring> {
    data: Vec<T::Element>,
    ring: T,
}

impl <T: Ring> Polynomial<T> {
    fn add(&self, other: &Polynomial<T>) → Polynomial<T> {
        // assume same degree for this example
        Polynomial {
            data: self.data.iter().zip(other.data.iter())
                .map(|(a, b)| self.ring.add(a, b)).collect(),
            ring: self.ring.clone(),
        }
    }
}
```

Abstraction example: polynomials

- Polynomials themselves are elements of a polynomial ring!

```
struct PolynomialRing<T: Ring> {
    coeff_ring: T
}
impl Ring for PolynomialRing<T: Ring> {
    type Element = Polynomial<T>;
    fn add(&self, a: &Polynomial<T>, b: &Polynomial<T>) → Polynomial<T> {
        a.add(b)
    }
}
```

- Polynomials over fields are Euclidean domains:

```
impl<T: Field> EuclideanDomain for PolynomialRing<T> { ... }
```

Abstraction example: polynomials

- We can compose types without writing any specific code!
- `PolynomialRing<Z>`, e.g. $2x^2 + 5$
- `PolynomialRing<FractionField<Z>>`, e.g. $\frac{2}{3}x + \frac{3}{2}$
- `PolynomialRing<PolynomialRing<Zp>>`, e.g. $(2 + y)x^2 + 3x + y$
- `FractionField<PolynomialRing<Zp>>`, e.g. $\frac{2+x}{1+x}$

Multivariate polynomial representation

- Multivariate polynomials are rarely dense in all variables
- Use a sparse representation that stores the non-zero coefficients and the associated exponent

```
struct MultivariatePolynomial<R: Ring> {
  coefficients: Vec<R::Element>,
  exponents: Vec<u32>, // flattened
  n_vars: usize,
  ring: R,
}

impl<R: Ring> MultivariatePolynomial<R> {
  pub fn get_monomial(&self, i: usize) → (&R::Element, &[u32]) {
    (&self.coefficients[i],
     &self.exponents[i * self.n_vars..(i+1) * self.n_vars])
  }
}
```

Applications of Extended Euclidean algorithm

Partial fraction decomposition

$$\frac{1}{D_1 D_2} = \frac{A_1}{D_1} + \frac{A_2}{D_2}$$

Rewrite:

$$1 = A_1 D_2 + A_2 D_1$$

Partial fraction decomposition

$$\frac{1}{D_1 D_2} = \frac{A_1}{D_1} + \frac{A_2}{D_2}$$

Rewrite:

$$1 = A_1 D_2 + A_2 D_1$$

- If D_1 and D_2 are coprime, we can find A_1 and A_2 uniquely
- Apply Extended Euclidean algorithm to find A_1 and A_2!

Examples:

$$\frac{1}{35} = \frac{1}{5 \cdot 7} = \frac{3}{7} - \frac{2}{5}$$

$$\frac{1}{(x^2 + x + 1)(x + 2)} = \frac{1}{3(x + 2)} + \frac{1 + x}{3(x^2 + x + 1)}$$

Rational function reconstruction

- Suppose we have unknown $f(x)$, N sample point (u_i, v_i) with $f(u_i) = v_i$
- Use polynomial interpolation to find $g(x)$ with $g(u_i) = v_i$
- We want to find $a(x), b(x)$ with $g(u_i) = \frac{a(u_i)}{b(u_i)}$

$$g(x)b(x) \equiv a(x) \bmod x - u_i \forall i \Rightarrow g(x)b(x) \equiv a(x) \bmod (x - u_1)...(x - u_N)$$

- $m(x) = (x - u_1)...(x - u_N)$
- $g(x)b(x) \equiv a(x) \bmod m(x)$
- $g(x)b(x) + m(x)s(x) = a(x)$
- Apply EEA$(m(x), g(x))$ to find options for $a(x)$ and $b(x)$!

Rational function reconstruction

- For $g(x) = 3x^2 + 3x + 1$ and $m(x) = x(x-1)(x-2) \in \mathbb{Z}_5$

r	q	s	t
$m(x)$		1	0
$g(x)$		0	1
$4x + 3$	$2x + 2$	1	$3x + 3$
4	$4x + 3$	$x + 4$	$3x^2 + 4$
0		$x^2 + x + 2$	$3x^2 + x^2 + x$

$$\frac{r_i}{t_i} = \left\{ g(x), \frac{4x+3}{3x+3}, \frac{4}{3x^2+4} \right\}$$

- If $\gcd(r, t) \neq 1 \Rightarrow t(u_i) = 0$ and there is no solution

Multivariate rational function reconstruction

- Evaluate the black box $f(x_1, x_2, \ldots)$ at $f(tx_1, tx_2, \ldots)$
- Perform reconstruction in t for fixed $\vec{s} = (x_1, x_2, \ldots)$
- Find stable solution of the form

$$r(\vec{s}, t) = \frac{n_0 + n_1 t + \ldots + n_a t^a}{d_0 + d_1 t + \ldots + d_b t^b}$$

Multivariate rational function reconstruction

- Evaluate the black box $f(x_1, x_2, \ldots)$ at $f(tx_1, tx_2, \ldots)$
- Perform reconstruction in t for fixed $\vec{s} = (x_1, x_2, \ldots)$
- Find stable solution of the form

$$r(\vec{s}, t) = \frac{n_0 + n_1 t + \ldots + n_a t^a}{d_0 + d_1 t + \ldots + d_b t^b}$$

- Take random sample $\vec{s}$ of $(x_1, \ldots)$ and $a + b + 2$ samples in t
- Solve linear system in unknowns n_i, d_i:

$$r(\vec{s}, t)\left(d_0 + d_1 t + \ldots + d_b t^b\right) = \left(n_0 + n_1 t + \ldots + n_a t^a\right)$$

- Perform polynomial interpolation per n_i and d_i

Square-free factorization

We can always write a polynomial as:

$$p(x) = a_1 a_2^2 a_3^3 ... a_n^n$$

where a_i, a_j are coprime

- Note that $\frac{d}{dx}p(x) = \frac{da_1}{dx}a_2 a_3^2 ... a_n^n + 2\frac{da_2}{dx}a_1 a_2 ... a_n^n + ...$ keeps all a_i except a_1
- $r(x) = \gcd\left(p(x), \frac{d}{dx}p(x)\right)$ gives us $a_2 a_3^2 ... a_n^{n-1}$
- $k(x) = \frac{p(x)}{r(x)} = a_1 a_2 a_3 ... a_n$

$g = \gcd(p(x), r(x)) = a_2 ... a_n$, so:

$$a_1 = \frac{k(x)}{g(x)}$$

- Repeat process on $\frac{p(x)}{k(x)} = a_2 a_3^2 ... a_n^{n-1}$
- Modification needed for finite fields, also works for multivariate polynomials

Real root isolation

- Roots of $z(x) = a_1 \cdots a_n$ are the same as those as $p(x) = a_1 \cdots a_n^n$
- Every root r of $z(x)$ has $\frac{d}{dx} z(x)|_{x=r} \neq 0$ as each root only occurs once
- We can bisect every segment $l < r < u$ with only one root of z as $\text{sign}(z(l)) \neq \text{sign}(z(u))$
- We can get arbitrarily close!
-
-

Real root isolation

- Roots of $z(x) = a_1 \cdots a_n$ are the same as those as $p(x) = a_1 \cdots a_n^n$
- Every root r of $z(x)$ has $\frac{d}{dx} z(x)|_{x=r} \neq 0$ as each root only occurs once
- We can bisect every segment $l < r < u$ with only one root of z as $\text{sign}(z(l)) \neq \text{sign}(z(u))$
- We can get arbitrarily close!
- Find initial root splitting partition using Descartes's rule of signs
- Count sign changes in coefficients of polynomial, ignoring zeros:

$$z(x) = 2x^4 - 3x^3 + 4x + 5 \Rightarrow +-++ \Rightarrow 2 \text{ sign changes}$$

  - Number of positive real roots is at most the number of sign changes, and differs from it by an even number
    - 0 sign changes -> 0 positive roots
    - 1 sign change -> 1 positive root
    - 2 sign changes -> 0 or 2 positive roots

Simple bisection strategy

- Find a bound on the largest root, e.g. Cauchy bound $b = 1 + \max\left(\left|\frac{c_{n-1}}{c_n}\right|, \ldots, \left|\frac{c_0}{c_n}\right|\right)$
- Divide interval $(-b, b)$ into equal segments $(-b, 0)$ and $(0, b)$
- Map interval (a, b) to $(0, \infty)$ and use the Descartes rule of signs to count roots
- Bisect until each segment has 0 or 1 root

Complex root approximation

- Aberth's method: cubically approximate all roots of $p(x)$ simultaneously
- Start from random z_i, update:

$$z_k = z_k - \frac{1}{\frac{p'(z_k)}{p(z_k)} - \prod_{j \neq k} \frac{1}{z_k - z_j}}$$

- Based on electrostatics: root approximations z_i are negative point charges
- Actual root locations are positive point charges
- Repulsive effect: when $p(z_a) = 0$, $z_{k \neq a}$ will not be attracted to z_a due to $\frac{1}{z_k - z_a}$

System solving

Univariate system solving

- How to solve

$$
\begin{aligned}
x + 4x^2 + x^3 &= 6 \\
-3x + 13x^2 + 7x^3 + x^4 &= 18 \\
2x + 3x^2 + 2x^3 + x^4 &= 8
\end{aligned}
$$

Univariate system solving

- How to solve

$$x + 4x^2 + x^3 = 6$$
$$-3x + 13x^2 + 7x^3 + x^4 = 18$$
$$2x + 3x^2 + 2x^3 + x^4 = 8$$

- Simply compute the gcd
- System simplifies to resulting common factor: $x^2 + x - 2 = 0$

Multivariate system solving

- If any polynomial factors, take the union of systems

$$f_1(x_1, \ldots) = a(x_1, \ldots)b(x_1, \ldots)$$
$$f_2(x_1, \ldots) = c(x_1, \ldots)d(x_1, \ldots)$$

Solve:

$$\{a(x_1, \ldots), c(x_1, \ldots)\} = 0$$
$$\{a(x_1, \ldots), d(x_1, \ldots)\} = 0$$
$$\{b(x_1, \ldots), c(x_1, \ldots)\} = 0$$
$$\{b(x_1, \ldots), d(x_1, \ldots)\} = 0$$

Resultants

- Given

$$a = a_0 + a_1 x + \ldots + a_d x^d$$

$$b = b_0 + b_1 x + \ldots + b_e x^e$$

$$\operatorname{res}_x(a, b) = \begin{vmatrix} a_d & 0 & \cdots & 0 & b_e & 0 & \cdots & 0 \\ a_{d-1} & a_d & \cdots & 0 & b_{e-1} & b_e & \cdots & 0 \\ a_{d-2} & a_{d-1} & \ddots & 0 & b_{e-2} & b_{e-1} & \ddots & 0 \\ \vdots & \vdots & \ddots & a_d & \vdots & \vdots & \ddots & b_e \\ a_0 & a_1 & \cdots & \vdots & b_0 & b_1 & \cdots & \vdots \\ 0 & a_0 & \ddots & \vdots & 0 & b_0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & a_1 & \vdots & \vdots & \ddots & b_1 \\ 0 & 0 & \cdots & a_0 & 0 & 0 & \cdots & b_0 \end{vmatrix}$$

- e columns of shifted a and d columns of shifted b

Properties of resultants

- $\operatorname{res}(ab, c) = \operatorname{res}(a, c) \operatorname{res}(b, c)$
- $\operatorname{res}\left(a_0, b\right) = \det \operatorname{diag}_e\left(a_0\right) = a_0^e$
- $\operatorname{res}(a, b) = (-1)^{de} \operatorname{res}(b, a)$
- $\operatorname{res}\left(x + a_0, x + b_0\right) = b_0 - a_0$
-

Properties of resultants

- $\operatorname{res}(ab, c) = \operatorname{res}(a, c) \operatorname{res}(b, c)$
- $\operatorname{res}\left(a_0, b\right) = \det \operatorname{diag}_e\left(a_0\right) = a_0^e$
- $\operatorname{res}(a, b) = (-1)^{de} \operatorname{res}(b, a)$
- $\operatorname{res}\left(x + a_0, x + b_0\right) = b_0 - a_0$
- Thus, if

$$a = a_0(x - r_1)(x - r_2)\cdots(x - r_d)$$

$$b = b_0(x - q_1)(x - q_2)\cdots(x - q_e)$$

$$\operatorname{res}(a, b) = a_0^e b_0^d (r_1 - q_1)(r_1 - q_2)\cdots(r_d - q_e)$$

- The resultant is only 0 when there is a common factor between a and b

Variable elimination

$$a = x^2 + 2xy - 1$$

$$b = x^2 + y^2 - 1$$

- System only has a solution when $\text{res}_y(a, b) = 1 - 6x^2 + 5x^4 = 0$
- Variable y eliminated!
-
-
-
-
-
-
-

Variable elimination

$$a = x^2 + 2xy - 1$$

$$b = x^2 + y^2 - 1$$

- System only has a solution when $\text{res}_y(a, b) = 1 - 6x^2 + 5x^4 = 0$
- Variable y eliminated!
- $x = \pm 1 \mid x = \pm\sqrt{\frac{1}{5}}$
- Alternatively: $\text{res}_x(a, b) = -4y^2 + 5y^4 = y^2\left(-4 + 5y^2\right)$
- $y = 0 \mid y = \pm\sqrt{\frac{4}{5}}$
- Fill in $y = 0$: $\gcd(a(x, 0), b(x, 0)) = x^2 - 1 \Rightarrow x = \pm 1$
- Fill in $y = \pm\sqrt{\frac{4}{5}}$: $\gcd\left(x^2 + 2x\left(\pm\sqrt{\frac{4}{5}}\right) - 1, x^2 - \frac{4}{5} - 1\right) \Rightarrow x = \pm\sqrt{\frac{1}{5}}$
- Requires algebraic numbers
- Some spurious roots are generated as the degree of the polynomial in x is increased

System solving

$$f_1(x, y) = x^2 + y^2 - 2x + 2y - 2 = 0$$
$$f_2(x, y) = xy + x - y - 1 = 0$$

- $\mathrm{Res}_y(f_1, f_2) = -3 + 4x + 2x^2 - 4x^3 + x^4$
- Rational root theorem: solutions of $c_0 + \ldots + c_d x^d$ are of the form

$$x = \frac{p}{q} \text{ with } p \mid c_0 \text{ and } q \mid c_d$$

System solving

$$f_1(x, y) = x^2 + y^2 - 2x + 2y - 2 = 0$$
$$f_2(x, y) = xy + x - y - 1 = 0$$

- $\text{Res}_y(f_1, f_2) = -3 + 4x + 2x^2 - 4x^3 + x^4$
- Rational root theorem: solutions of $c_0 + \ldots + c_d x^d$ are of the form

$$x = \frac{p}{q} \text{ with } p \mid c_0 \text{ and } q \mid c_d$$

- Try $p = \{-1, 3, -3\}, q = \{1, -1\}$
- Trial by evaluating gives: $x = 3, x = -1$
- $\gcd\big(f_1(3, y), f_2(3, y)\big) = 1 + y \Rightarrow y = -1$
- $\gcd\big(f_1(-1, y), f_2(-1, y)\big) = 1 + y \Rightarrow y = -1$

System solving

$$a_1 = 8x^3 + 26y + 14z - xyz - 54$$

$$a_2 = 8y^3 + 26x + 14z - xyz - 48$$

$$a_3 = 4z^2 - 8yz - 24xz + 4y^2 + 24xy + 36x^2 - 1$$

- $r_{12} = \mathrm{res}_z(a_1, a_2) = 84 + 364x - 112x^3 - 364y - 6xy - 26x^2y + 8x^4y + 26xy^2 + 112y^3 - 8xy^4$
- $r_{13} = \mathrm{res}_z(a_1, a_3) = 11468 - 18144x + 7056x^2 - 3456x^3 + 2688x^4 + 256x^6 - 17280y + 13468xy + 1296x^2y + 1552x^3y - 192x^5y + 6400y^2 + 432xy^2 - 1297x^2y^2 - 28x^4y^2 - 320xy^3 + 24x^3y^3 + 4x^2y^4$

System solving 2

- $\operatorname{res}_y(r_{12}, r_{13}) = 5430173751721831956 48x^2 - 7299897246395400192 00x^3 - 5909010340709899698176x^4 + 1233972870115377964646 4x^5 - 7933599054828011520 0x^6 - 1413783394509829491916 8x^7 + 1792538827849204432896 0x^8 - 1242557788848624146841 6x^9 + 6839963428994500608000x^{10} - 4129382068764911665152x^{11} + 2408867363337109536768x^{12} - 1192892323804862545920x^{13} + 4818265757515740119 04x^{14} - 2075200671274956226 56x^{15} + 9314579458407379763 2x^{16} - 3450943338826629120 0x^{17} + 1117597300552551628 8x^{18} - 3802984778282041344x^{19} + 1451320125577410252 8x^{20} - 3499887372659589 12x^{21} + 1092298010095779 84x^{22} - 255498215321763 84x^{23} + 85432676709826 56x^{24} - 6960460238684 16x^{25} + 5204045274808 32x^{26} + 1139132701081 6x^{28}$
- Rational roots: $x = 0, x = \frac{1}{2}$

System solving 3

- $r_{12}(x = 0) = 84 - 364y + 112y^3$ => no rational solution
- $r_{12}\left(x = \frac{1}{2}\right) = 252 - 373y + 13y^2 + 112y^3 - 4y^4 \Rightarrow y = 1, y = 28$
- $r_{13}\left(x = \frac{1}{2}\right) = 3900 - 10034y + 6290y^2 - 157y^3 + y^4 \Rightarrow y = 1, y = 78$
- $\gcd\left(a_1\left(\frac{1}{2}, 1\right), a_2\left(\frac{1}{2}, 1\right), a_3\left(\frac{1}{2}, 1\right)\right) = -1 + \frac{z}{2} \Rightarrow z = 2$
- Only solution: $(x, y, z) = \left(\frac{1}{2}, 1, 2\right)$

Multivariate system solving

System extension

Given the system f:

$$x + 4x^2 + x^3 - 6 = 0$$
$$-3x + 13x^2 + 7x^3 + x^4 - 18 = 0$$
$$2x + 3x^2 + 2x^3 + x^4 - 8 = 0$$

- We can always add $\sum c_i f_i = 0$ to the system without changing the solution
- Mod operations are linear combinations of the input
  - $f_i \bmod f_j = f_i - q f_j$
  - $f_4 \stackrel{\text{def}}{=} f_3 \bmod f_1 = -20 + 10x + 10x^2$
- We can remove f_3 from the system as $\{f_1, f_4\}$ has the same solutions as $\{f_1, f_3\}$
-
-

System extension

Given the system f:

$$x + 4x^2 + x^3 - 6 = 0$$
$$-3x + 13x^2 + 7x^3 + x^4 - 18 = 0$$
$$2x + 3x^2 + 2x^3 + x^4 - 8 = 0$$

- We can always add $\sum c_i f_i = 0$ to the system without changing the solution
- Mod operations are linear combinations of the input
  - $f_i \bmod f_j = f_i - q f_j$
  - $f_4 \stackrel{\text{def}}{=} f_3 \bmod f_1 = -20 + 10x + 10x^2$
- We can remove f_3 from the system as $\{f_1, f_4\}$ has the same solutions as $\{f_1, f_3\}$
- Iterating leaves a single polynomial which is $\gcd(f_1, \ldots, f_n)$
- Can we do the same for multivariate polynomials?
  - Can we produce a simplified polynomial that is univariate?

Leading term division

- Given a set of multivariate polynomials b_i
  - Also called a **basis** of the solutions of $\sum_i c_i b_i = 0$
- We cannot simply do $\tilde{b} = b + \left\{ b_i \bmod b_j \right\}$
  - Should $x \bmod (x + z)$ yield x or $-z$?
- Define an ordering on monomials, and write the leading term $\mathrm{lt}(f) = \mathrm{lc}(f)\,\mathrm{lm}(f)$
- Leading term division of f by g wrt ordering if $\mathrm{lc}(g) \mid \mathrm{lc}(f)$:

$$q = \frac{\mathrm{lt}(f)}{\mathrm{lt}(g)}, r = f - qg$$

Monomial orderings

- Consider the exponent list $(e_1, ..., e_n) \to x_1^{e_1} \cdots x_n^{e_n}$ and ordering $\underset{T}{\leq}$
- Total ordering:
  - $(0, ..., 0) \underset{T}{\leqq} t; \forall t \in \mathbb{N}^n$
  - $s \underset{T}{\leq} t \Rightarrow s \cdot u \underset{T}{\leq} t \cdot u; \forall s, u, t \in \mathbb{N}^n$
- Lexicographical ordering: $(i_1, ..., i_n) \underset{L}{<} (j_1, ..., j_n) \Leftrightarrow \exists l i_l < j_l \wedge i_k = j_k, k < l$
- Graded reverse lexicographical ordering: $(i_1, ..., i_n) \underset{G}{<} (j_1, ..., j_n) \Leftrightarrow \sum i_k < \sum j_k \vee \left(\sum i_k = \sum j_k \wedge (i_n, ..., i_1) \underset{L}{>} (j_n, ..., j_1)\right)$

Reduction

Reduction of a polynomial by an ordered basis B ($\mathrm{red}(f, B)$):

```
def reduce(f, B):
  r = 0
  while f ≠ 0:
    b = next((h for h in B if f.lm() ⩾ b.lm()), None)
    if b is None:
      r += f.lt()
      f -= f.lt()
    else:
      f -= f.lt() / b.lt() * b
  return r
```

Non-unique reduction

- Reduction is not unique
- May not discover a linear combination of the basis elements
- Reduction of $f = xy^2 - x$ by $f_1 = xy + 1, f_2 = y^2 - 1$:

$$xy^2 - x = yf_1 + 0f_2 - x - y$$

- There is a remainder $-x - y$, however we could have also done:

$$xy^2 - x = 0f_1 + xf_2 + 0$$

Gröbner bases and critical pairs

- A basis $B = \{b_1, ...\}$ is a Gröbner basis if:

$$f = \sum c_i b_i \forall c_i \Rightarrow \text{red}(f, B) = 0$$

- Alternatively, given:

$$h = \text{gcd}(\text{lm}(f), \text{lm}(g))$$

$$S(f, g) = \frac{\text{lt}(g)}{h} f - \frac{\text{lt}(f)}{h} g$$

A basis B is a Gröbner basis if for all $f, g \in B$, if:

$$\tilde{S}(f, g) = \text{red}(S(f, g), B) = 0$$

- Idea: if $\tilde{S}(f, g) \neq 0; f, g \in B$, add $\tilde{S}(f, g)$ to the basis and repeat

Buchberger algorithm

```
def groebner_basis(B):
    t = [(i, j) for i in range(len(B)) for j in range(i + 1, len(B))]
    while len(t) > 0:
        i, j = t.pop()
        f, g = B[i], B[j]
        h = gcd(f.lm(), g.lm())
        s = reduce(g.lt() / h * f - f.lt() / h * g, B)
        if s ≠ 0:
            B.append(s)
            t += [(k, len(B) - 1) for k in range(len(B) - 1)]
    return B
```

- When applied to linear polynomials, this is Gaussian elimination

Examples

- System from before in lex order: $f = \{xy + 1, y^2 - 1\}$

$$h = \gcd(\mathrm{lm}(f_1), \mathrm{lm}(f_2)) = y$$

$$S(f_1, f_2) = \frac{y^2}{h} f_1 - \frac{xy}{h} f_2 = y f_1 - x f_2 = xy^2 + y - xy^2 + x = x + y$$

Examples

- System from before in lex order: $f = \{xy + 1, y^2 - 1\}$

$$h = \gcd(\mathrm{lm}(f_1), \mathrm{lm}(f_2)) = y$$

$$S(f_1, f_2) = \frac{y^2}{h} f_1 - \frac{xy}{h} f_2 = y f_1 - x f_2 = xy^2 + y - xy^2 + x = x + y$$

- Extend basis: $\tilde{f} = \{xy + 1, y^2 - 1, x + y\}$
- Try new combination:

$$h = x$$

$$S(f_1, f_3) = \frac{x}{h} f_1 - \frac{xy}{h} f_3 = xy + 1 - xy - y^2 = -y^2 + 1$$

- $\mathrm{red}(S(f_1, f_3), f_2) = 0 \Rightarrow$ Gröbner basis $\tilde{f}$ found
- $\mathrm{red}(xy^2 - x, \tilde{f}) = y f_1 + 0 f_2 - x - y = y f_1 + 0 f_2 - f_3$

Basis reduction

- Redundant elements can be removed and basis elements can be simplified

```
def simplify_basis(basis):
  new_basis = set()
  while len(basis) > 0:
      f = reduce(basis.pop(), new_basis)
      if f ≠ 0:
        for q in new_basis: # remove redundant elements
            if f.lm() ⩽ q.lm():
                basis.append(q)
                new_basis -= q
          new_basis += f

  return [reduce(h, new_basis - h) for h in new_basis]
```

- For $\{xy + 1, y^2 - 1, x + y\}$ is redundant as $xy > x$
- Unique reduced basis: $\{y^2 - 1, x + y\}$

Gröbner bases optimizations

- If the system has no solution, the reduced basis is simply {1}
- Many S-polynomials will reduce to 0, filtering algorithms can be used
- Gröbner bases can also be computed using matrix row reduction where each monomial is a column
- Gröbner basis finding is EXPSPACE-complete, sometimes double-exponential basis growth, double exponential degree growth, exponential space usage 😔

System solving

- Assume we have a Gröbner basis B for any order in $\mathbb{F}[x_1, ...]$
- Assume solutions are 0-dimensional (finite number of solutions)
- We can find:

$$f(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + ... + a_n x_i^{n_i} = 0 \text{ with } \text{red}(f, B) = 0$$

for some $n_i, a_i \in \mathbb{F}$ whose solutions in x_i are solutions of the system

- Solve the linear system: $\sum_j^k a_j \text{ red}\left(x_i^j, B\right) = 0$
- Start at $k = 0$ and keep going until the linear system has non-trivial solution
- Advantage over resultant method: no spurious roots
- Roots can be filled in in B and further reduction can take place

System solving

- In lex order, we expect a triangular system, as z in $x < y < z$ must be eliminated first

$$B = \left\{-2 + x^2 + yz, -3 + y^2 + xz, -5 + xy + z^2\right\}$$

Gröbner basis:

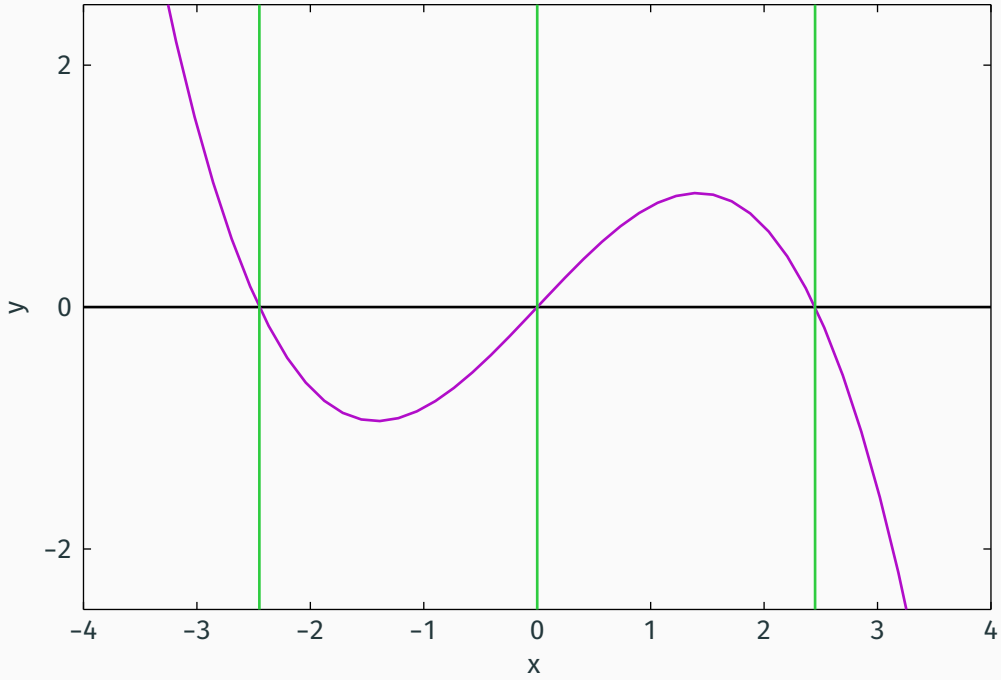
$$\tilde{B} = \left\{x - \frac{88}{361}z^7 + \frac{872}{361}z^5 - \frac{2690}{361}z^3 + 125z,\right.$$
$$y + \frac{8}{361}z^7 + \frac{52}{361}z^5 - \frac{740}{361}z^3 + \frac{75}{19}z,$$
$$\left.z^8 - \frac{25}{2}z^6 + \frac{219}{4}z^4 - 95z^2 + \frac{361}{8}\right\}$$

- Solve z, fill in roots in polynomials in $\mathbb{Q}[y, z]$, take gcd and solve for y

Solving systems of inequalities

Solving inequalities

- How to solve $x - \frac{x^3}{6} < 0$?
- Solve $x - \frac{x^3}{6} = 0$ and create cells: $\left\{\left(-\infty, -\sqrt{6}\right), -\sqrt{6}, \left(-\sqrt{6}, 0\right), 0, \left(0, \sqrt{6}\right), \sqrt{6}, \left(\sqrt{6}, \infty\right)\right\}$
- Sign invariance for every point in a cell
- Test sign by sampling any point in a cell



Solving systems of inequalities

How to solve

$$x - \frac{x^3}{6} > 0$$

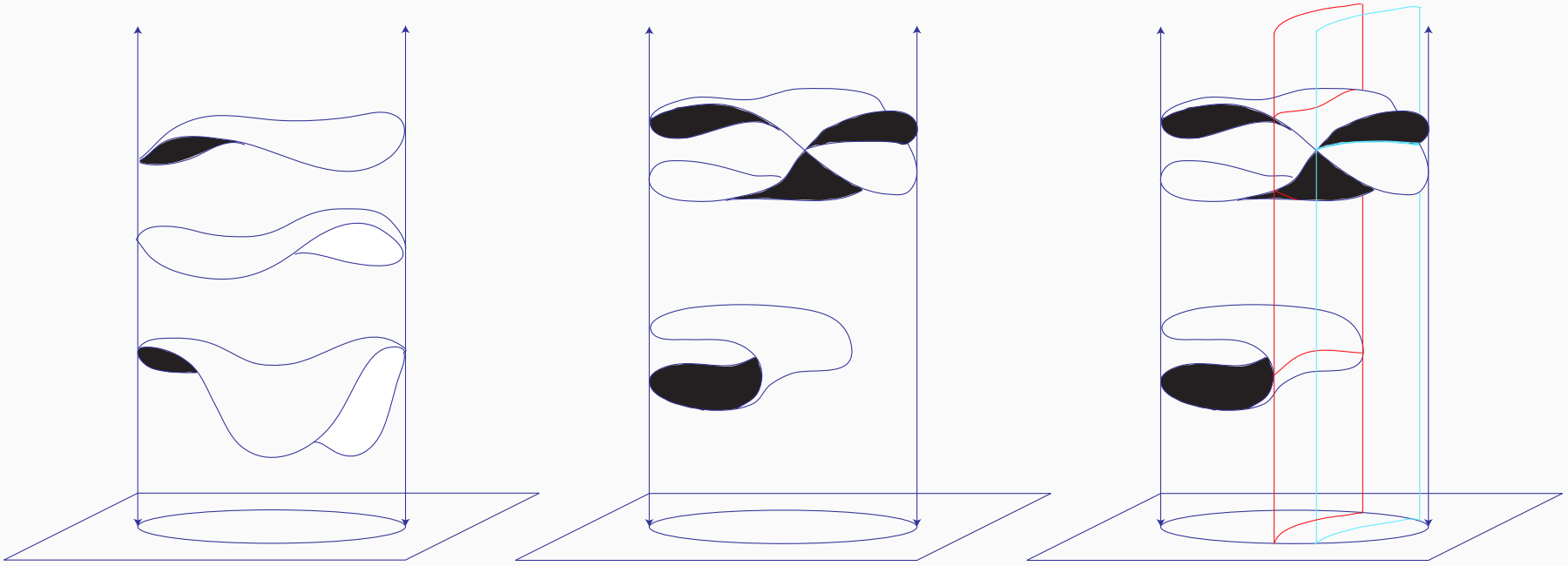
$$x^2 + 2x - 2 \leq 0$$

- Isolate roots from both polynomials and sort them
  - New roots: $\left\{-1 - \sqrt{3}, -1 + \sqrt{3}\right\}$
  - Cells:

$$\left\{\left(-\infty, -1 - \sqrt{3}\right), -1 - \sqrt{3}, \left(-1 - \sqrt{3}, -\sqrt{6}\right),\right.$$
$$-\sqrt{6}, \left(-\sqrt{6}, 0\right), 0, \left(0, -1 + \sqrt{3}\right), -1 + \sqrt{3},$$
$$\left.\left(-1 + \sqrt{3}, \sqrt{6}\right), \sqrt{6}, \left(\sqrt{6}, \infty\right)\right\}$$

Higher dimensions

- Idea: project from 2d to 1d in such a way that every cell in 1d corresponds can be extended to cells in 2d that have the same root ordering

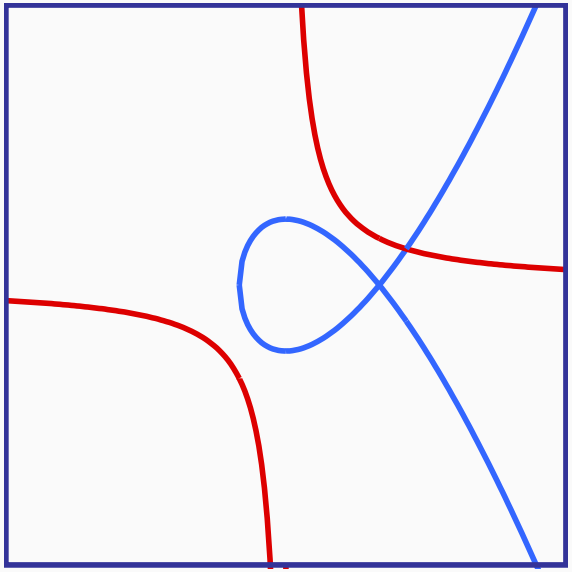


Images on this and next slides by Brown

- Called Cylindrical Algebraic Decomposition (CAD)

Delineability

- Look at $A = \{p = 2y^2 - x^2(2x + 3), q = 2(x + 1)y - 1\}$



- Features that can cause sign changes:
  - Tangent points wrt x axis
  - Curve intersections
  - Intersections between curves

Features

- Intersections between curves: $\text{res}_y(p, q) = 0$
- The discriminant of a curve wrt y finds double poles:

$$\text{disc}_y(p) = \text{res}_y\left(p, \frac{dp}{dy}\right) = 0$$

- This is where a curve intersects itself and where there are tangent points

Projection

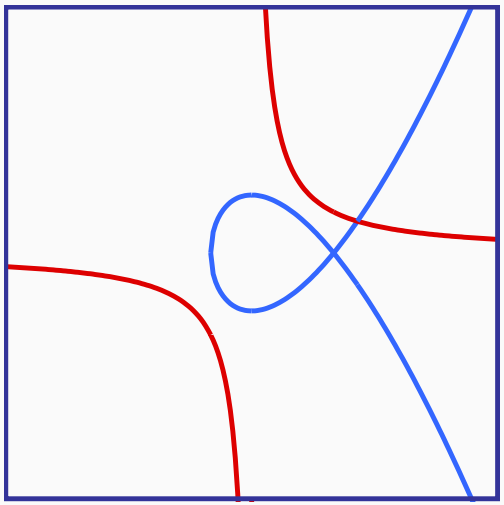
- Many projection operators possible
- The one with the least number of splits is the best
- Brown-McCallum projection operator:

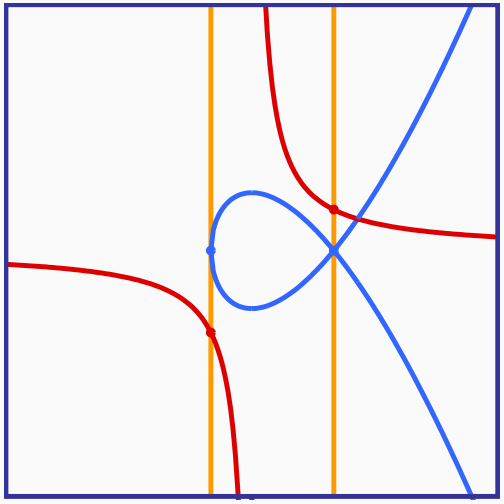
$$P(A_n) = \left\{ \text{res}_{x_n}\left(p, \frac{dp}{dx_n}\right), \text{lc}_{x_n}(p) \mid p \in A_n \right\} \cup \left\{ \text{res}_{x_n}(p, q) \mid p, q \in A_n, p \neq q \right\}$$

- Project all the way down to 1d and perform real root isolation to construct cells

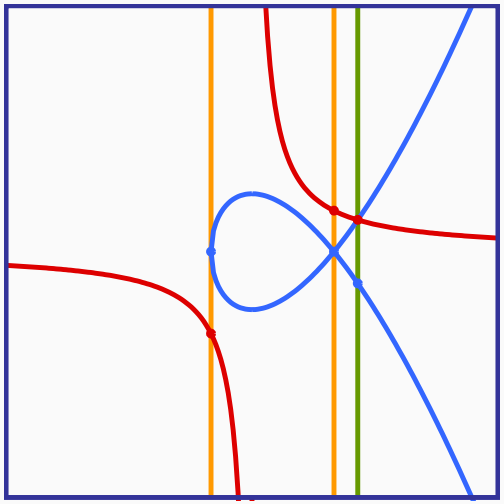
Projection

$$A = \{p = 2y^2 - x^2(2x + 3), q = 2(x + 1)y - 1\}$$

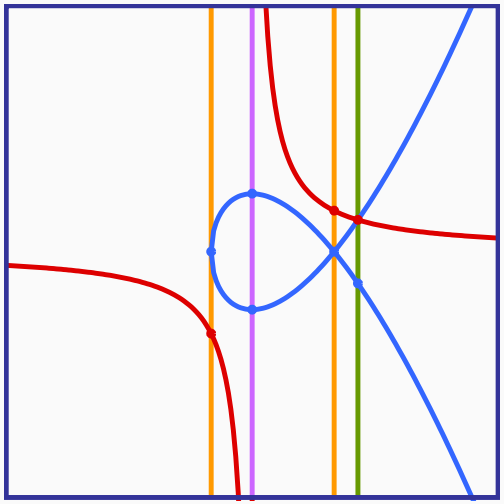




$\text{disc}_y(p) \to \{-\frac{3}{2}, 0\}$



$\text{res}_y(p, q) \to \{0.289\}$



$\text{lc}_y(q) \to \{-1\}$

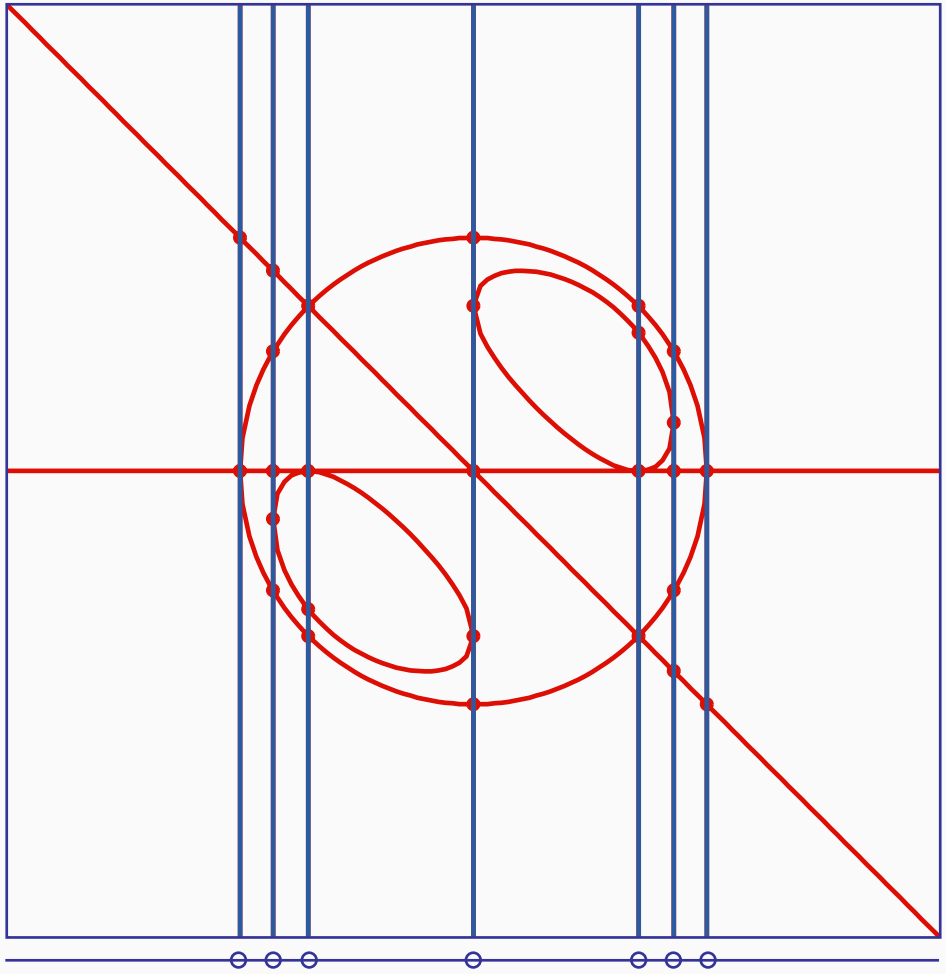
- Cells: $\{(-\infty, -\frac{3}{2}), -\frac{3}{2}, (-\frac{3}{2}, -1), -1, (-1, 0), 0, (0, 0.289), 0.289, (0.289, \infty)\}$

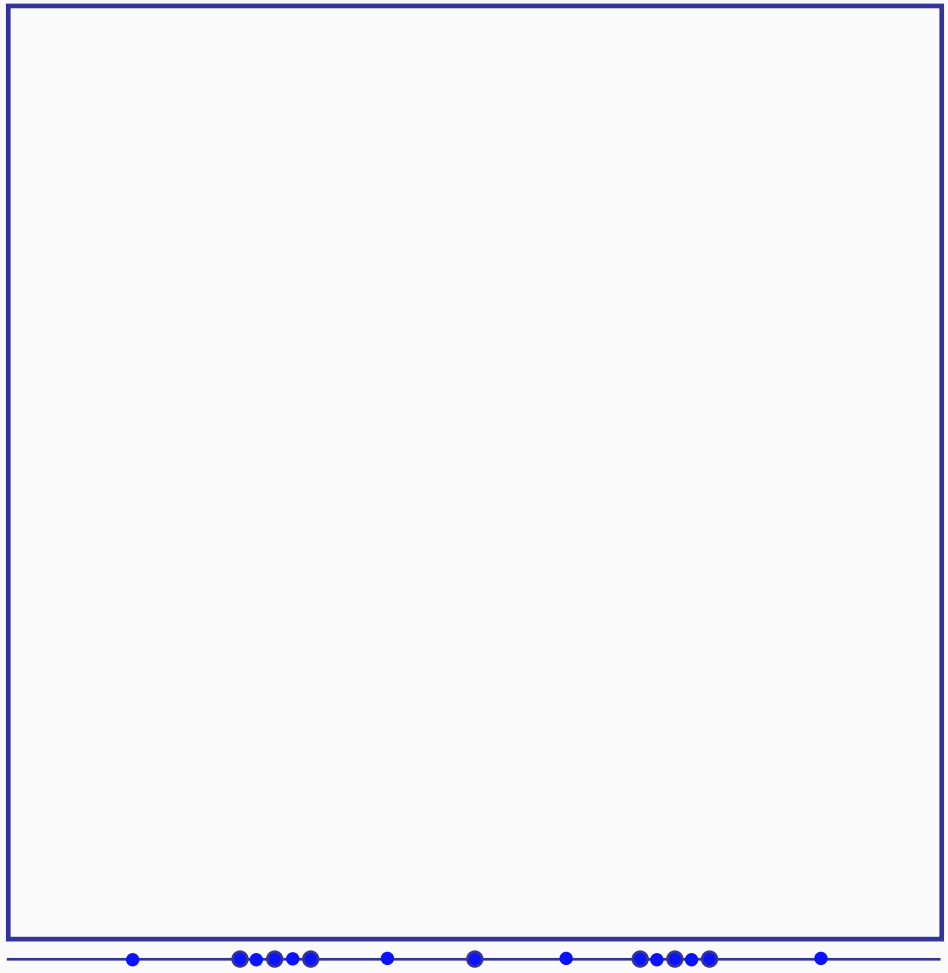
Lifting

- For every cell: sample a (rational if possible) point and fill it in to the polynomial
- Pick $x = -\frac{1}{2}$ for the cell $(-1, 0)$
- We now have $p(x, y) \to p\left(-\frac{1}{2}, y\right), A\left(-\frac{1}{2}, y\right) = \left\{2y^2 - \frac{1}{2}, y - 1\right\}$
- Sort and tag roots: $\left\{ \underbrace{-\frac{1}{2}}_{\text{first root of p}}, \underbrace{\frac{1}{2}}_{\text{second root of p}}, \underbrace{1}_{\text{first root of q}} \right\}$
- Let $k_i(x) = \text{root}_y(k(x, y), i)$
- Cell lifting:

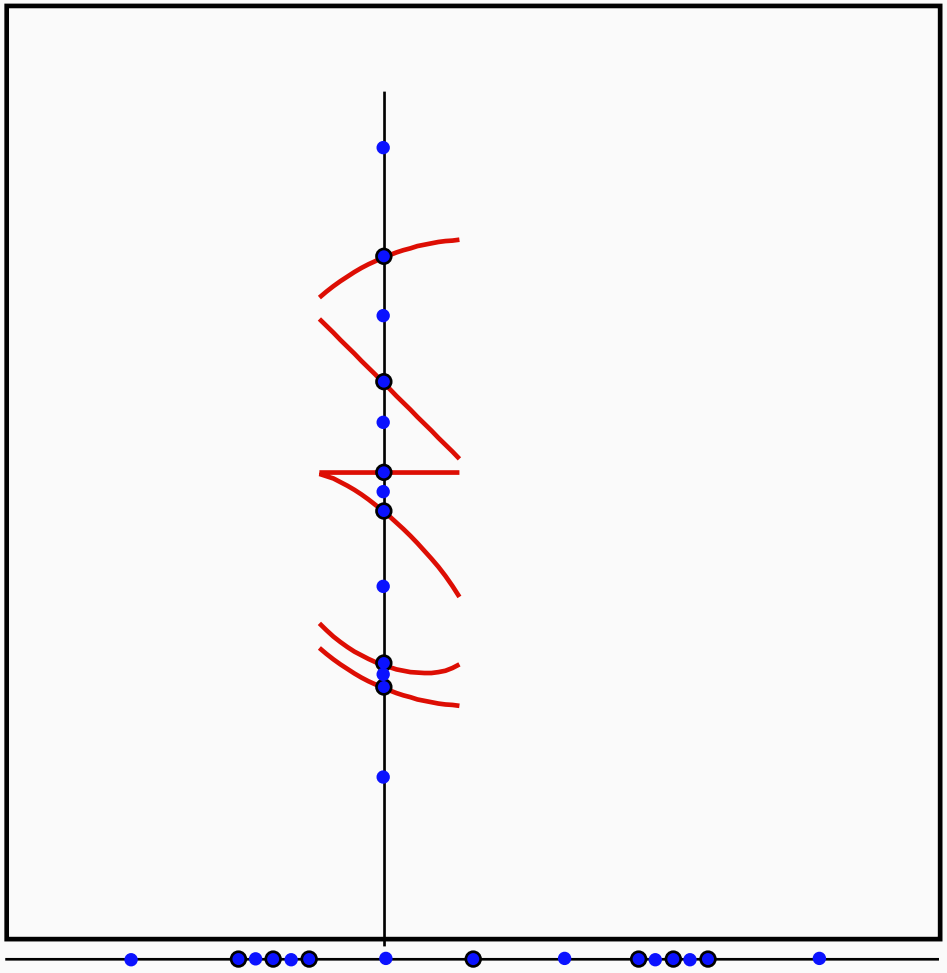
$$-1 < x < 0 \wedge \{-\infty < y < p_1(x), y = p_1(x), p_1(x) < y < p_2(x),$$
$$y = p_2(x), p_2(x) < y < q_1(x), y = q_1(x), q_1(x) < y < \infty\}$$

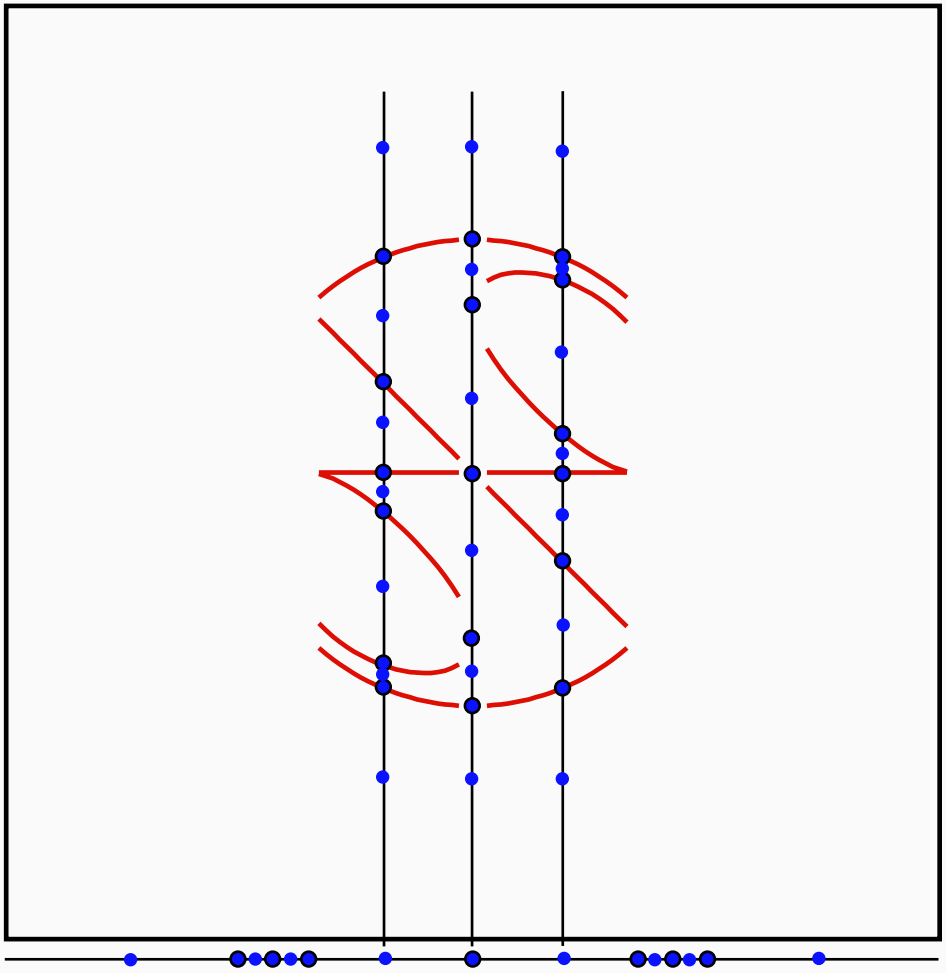
Lifting example





Lifting example





Polynomial GCDs

Polynomial GCDs

- Just factor the polynomials?
  - Factoring is hard and has many GCD sub-problems
- Euclidean algorithm can yield huge coefficients
- Combat expression swell by computing in finite fields and then reconstruct the result?
  - The mapping $\varphi_p : \mathbb{Z}[x] \to \mathbb{Z}_p[x]$ is a ring homomorphism
  - $f = f_1 \cdot f_2 \in \mathbb{Z}[x] \to \varphi_p(f) = \varphi_p(f_1 \cdot f_2) = \varphi_p(f_1) \cdot \varphi_p(f_2)$
  - Factorization is preserved → GCD is preserved

Chinese Remainder Theorem

- Given an unknown $n \in \mathbb{Z}$ and $m_1 \in \mathbb{Z}$, $m_2 \in \mathbb{Z}$, $\gcd(m_1, m_2) = 1$ and

$$n \equiv a_1 \bmod m_1$$

$$n \equiv b_1 \bmod m_2$$

- Can we reconstruct n?

Chinese Remainder Theorem

- Given an unknown $n \in \mathbb{Z}$ and $m_1 \in \mathbb{Z}$, $m_2 \in \mathbb{Z}$, $\gcd(m_1, m_2) = 1$ and

$$n \equiv a_1 \bmod m_1$$

$$n \equiv b_1 \bmod m_2$$

- Can we reconstruct n?

$$n = a_1 t m_2 + a_2 s m_1 \bmod m_1 m_2$$

Chinese Remainder Theorem

- Given an unknown $n \in \mathbb{Z}$ and $m_1 \in \mathbb{Z}$, $m_2 \in \mathbb{Z}$, $\gcd(m_1, m_2) = 1$ and

$$n \equiv a_1 \bmod m_1$$

$$n \equiv b_1 \bmod m_2$$

- Can we reconstruct n?

$$n = a_1 t m_2 + a_2 s m_1 \bmod m_1 m_2$$

$$n = a_1 (m_2^{-1} \bmod m_1) m_2 + a_2 (m_1^{-1} \bmod m_2) m_1 \bmod m_1 m_2$$

Chinese Remainder Theorem

- Given an unknown $n \in \mathbb{Z}$ and $m_1 \in \mathbb{Z}$, $m_2 \in \mathbb{Z}$, $\gcd(m_1, m_2) = 1$ and

$$n \equiv a_1 \bmod m_1$$

$$n \equiv b_1 \bmod m_2$$

- Can we reconstruct n?

$$n = a_1 t m_2 + a_2 s m_1 \bmod m_1 m_2$$

$$n = a_1 (m_2^{-1} \bmod m_1) m_2 + a_2 (m_1^{-1} \bmod m_2) m_1 \bmod m_1 m_2$$

- s and t are the factors from the extended Euclidean algorithm $\operatorname{egcd}(m_1, m_2)$!
- Inverse exists as m_1 and m_2 are coprime!
- Can be iterated, as $m_3 \in \mathbb{Z}, \gcd(m_1, m_3) = 1 \land \gcd(m_2, m_3) = 1 \Rightarrow \gcd(m_1 m_2, m_3) = 1$
- n is unique, so integers smaller than $m_1 m_2 \div 2$ are stable

Modular determinant

- Compute $\det\begin{pmatrix} 37 & 5 \\ 20 & 3 \end{pmatrix} = 11$
- Use prime 3: $\det\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix} = 2$
- Use prime 5: $\det\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix} = 1$
- Use prime 7: $\det\begin{pmatrix} 2 & 5 \\ 6 & 3 \end{pmatrix} = 4$
- $\text{egcd}(3, 5) \Rightarrow s = \underline{2}, t = \underline{-1}$
- CRT: $2 \cdot \underline{-1} \cdot 5 + 1 \cdot \underline{2} \cdot 3 = 11 \bmod 15$
- $\text{egcd}(15, 7) \Rightarrow s = \underline{1}, t = \underline{-2}$
- CRT: $11 \cdot \underline{-2} \cdot 7 + 4 \cdot \underline{1} \cdot 15 = 11 \bmod 105$
- Stable: result is 11 over the integers or larger than $\frac{105}{2}$

Rational number/function reconstruction

- We want to reconstruct $\frac{a}{b} = u_i \bmod m_i \forall i$
- u_i, m_i are elements of a Euclidean domain ($\mathbb{F}$ or $\mathbb{F}[x]$)
- CRT: reconstruct $u \equiv u_i \bmod M$, where $M = \prod m_i$
- $a \equiv bu \bmod M \Rightarrow a = bu + tM$
- As we saw earlier, $\gcd(u, M)$ options for a and b
- u will never stabilize but a and b will

Rational number/function reconstruction

- Given polynomial $n(x)$ that interpolates points $\left(x_i, y_i\right)$
- $M(x) = \prod\left(x - x_i\right)$
- $n\left(x_i\right) = \frac{a\left(x_i\right)}{b\left(x_i\right)}$
- Reconstruct $\frac{a(x)}{b(x)}$ using $a(x) = b(x)n(x) \bmod M(x)$
- Use $\text{egcd}(n(x), M(x))$ to find $a(x)$ and $b(x)$
- Use CRT on coefficients to go from finite field $\mathbb{Z}_p$ to $\mathbb{Z}$
- Use egcd to lift coefficients from $\mathbb{Z}$ to $\mathbb{Q}$

Modular GCD

Compute GCD of

$$a = 24 + 3x + 40x^2 + 5x^3 = \left(5x^2 + 3\right)(x + 8)$$

$$b = 6 + 6x + 10x^2 + 10x^3 = 2\left(5x^2 + 3\right)(x + 1)$$

- What primes could cause problems?
-
-

Modular GCD

Compute GCD of

$$a = 24 + 3x + 40x^2 + 5x^3 = \left(5x^2 + 3\right)(x + 8)$$

$$b = 6 + 6x + 10x^2 + 10x^3 = 2\left(5x^2 + 3\right)(x + 1)$$

- What primes could cause problems?
- $b \bmod 2 = 0$, so GCD is too large: remove integer content first
- $b \bmod 5 = 0$, knocks out leading term so GCD is too small: $\gcd(a \bmod 5, b \bmod 5) = 1$

Accidental lower degree GCD

$$a = (10x + 1)(3x + 1)(x + 1)$$
$$b = (10x + 1)(3x + 1)(x + 2)$$

- $\gcd(a, b) \bmod 2 = x + 1$
- $\gcd(a, b) \bmod 3 = x + 1$ (stable GCD but not coming from the same factor!)
- $\gcd(a, b) \bmod 5 = 3x + 1$
- CRT: $13x + 1 \bmod 30$
- Leads to a bad reconstruction that is only detected after many primes
  - Cannot early return when the CRT stabilizes as we may have reconstructed a subfactor

Preventing GCD construction of lower degree

$$a = 1 + 106x + 3x^2 + 318x^3 = \left(3x^2 + 1\right)(106x + 1)$$

$$b = 1 + 10x + 3x^2 + 30x^3 = \left(3x^2 + 1\right)(10x + 2)$$

- Ensure that $\deg_x(a \bmod p) = \deg_x(a)$, $\deg_x(b \bmod p) = \deg_x(b)$
- Better: $l = \gcd(\mathrm{lc}(a), \mathrm{lc}(b))$, $\deg(l \bmod p) = \deg(l)$, where “lc” is the leading coefficient
- $l = \gcd(318, 30) = 6$
- Primes 2 and 3 are rejected

Scaling

$$a = 1 + 106x + 3x^2 + 318x^3 = \left(3x^2 + 1\right)(106x + 1)$$

$$b = 1 + 10x + 3x^2 + 30x^3 = \left(3x^2 + 1\right)(10x + 2)$$

- We get modular GCDs with lc of 1:
  - $\gcd(a, b) \bmod 5 = 2 + x^2$
  - $\gcd(a, b) \bmod 7 = 5 + x^2$
- To apply Chinese Remainder Theorem we need to scale the images
-

Scaling

$$a = 1 + 106x + 3x^2 + 318x^3 = \left(3x^2 + 1\right)(106x + 1)$$

$$b = 1 + 10x + 3x^2 + 30x^3 = \left(3x^2 + 1\right)(10x + 2)$$

- We get modular GCDs with lc of 1:
  - $\gcd(a, b) \bmod 5 = 2 + x^2$
  - $\gcd(a, b) \bmod 7 = 5 + x^2$
- To apply Chinese Remainder Theorem we need to scale the images
- Scale every image with $l \bmod p$, an overestimate of the lc of the GCD
  - $\gcd(a, b) \bmod 5 = 2 + x^2$
  - $\gcd(a, b) \bmod 7 = 2 + 6x^2$
  - Reconstruct gcd $2 + 6x^2$
  - Strip integer content: $1 + 3x^2$

Can we get a GCD that is too large?

$a = \left(5x^2 + 3\right)(x + 8), b = \left(5x^2 + 3\right)(x + 1)$

Can we get a GCD that is too large?

$a = \left(5x^2 + 3\right)(x + 8), b = \left(5x^2 + 3\right)(x + 1)$

- If $p = 7$, we get gcd $\left(5x^2 + 3\right)(x + 1)$!
- This is rare and cannot be detected in advance
- Sampling with enough different primes will expose a degree mismatch
- A division test in the end will catch this and we can try again with a stricter bound on the GCD degree

Example

$a = 24 + 3x + 40x^2 + 5x^3 = \left(5x^2 + 3\right)(x + 8)$ and $b = 6 + 6x + 10x^2 + 10x^3 = 2\left(5x^2 + 3\right)(x + 1)$

- Remove content: $b' = \frac{b}{2}$
- $\gcd(\mathrm{lc}(a), \mathrm{lc}(b')) = 5$
- $\gcd(a, b') \bmod 3 = \gcd\left(x^2 + 2x^3, 2x^2 + 2x^3\right) = x^2 \Rightarrow 3x^2$
-
-
-
-
-

Example

$a = 24 + 3x + 40x^2 + 5x^3 = \left(5x^2 + 3\right)(x + 8)$ and $b = 6 + 6x + 10x^2 + 10x^3 = 2\left(5x^2 + 3\right)(x + 1)$

- Remove content: $b' = \frac{b}{2}$
- $\gcd(\mathrm{lc}(a), \mathrm{lc}(b')) = 5$
- $\gcd(a, b') \bmod 3 = \gcd\left(x^2 + 2x^3, 2x^2 + 2x^3\right) = x^2 \Rightarrow 3x^2$
- $\gcd(a, b') \bmod 11 = \gcd\left(2 + 3x + 7x^2 + 5x^3, 3 + 3x + 5x^2 + 5x^3\right) = 5 + x^2 \Rightarrow 3 + 5x^2$
- $\gcd(a, b') \bmod 13 = 11 + x^2 \Rightarrow 3 + 5x^2$
- Last two samples are the same, so we may have the correct GCD
- Verify that the first sample is consistent: $\mathrm{egcd}(3, 11) \Rightarrow s = 4, t = -1$
- $0 + 0x + 3x^2 \bmod 3 \leftrightarrow 3 + 0x + 5x^2 \bmod 11 \Rightarrow (3 \cdot 4 \cdot 3) + 0x + (3 \cdot -1 \cdot 11 + 5 \cdot 4 \cdot 3)x^2 \bmod 33 = 3 + 5x^2$

Example II

- $a = \left(5x^2 + 40x + 21\right)(x + 8), b = \left(5x^2 + 40x + 21\right)(x + 1)$
- Sample $p = 3, 11, 13$
- $\text{egcd}(3, 11) \Rightarrow s = 4, t = -1$, $\text{egcd}(33, 13) \Rightarrow s = 2, t = -5$
- CRT: $n = n_1 t p_2 + n_2 s p_1$

	1	x	x^2
3	0	1	2
11	10	7	5
13	8	1	5
33	21	7	5
429	21	40	5

Multivariate polynomial GCDs

- Consider $\mathbb{Z}_p[x_1, x_2, x_3, \ldots]$
- Can be viewed as $\mathbb{Z}_p[x_2, x_3, \ldots][x_1]$
- However, $\mathbb{Z}_p[x_2, x_3, \ldots]$ is not a Euclidean domain so we cannot use the Euclidean algorithm
- We could upgrade to $F_{\mathbb{Z}_p[x_2, x_3, \ldots]}$ but this makes the coefficient arithmetic very slow
- We go for another modular approach: sampling and interpolation

Polynomial sampling

- Sampling a polynomial $f(x) \in \mathbb{R}[x]$ at $x = u$ is the same as $f(x) \bmod x - u$:
  - Shift: $f(x + u) \bmod x$, we get $c_0 + c_1(x + u) + c_2(x + u)^2 + \ldots \bmod x$
  - Evaluates to $f(u)$
- A polynomial of degree n is uniquely determined by $n + 1$ sample points
- What are the degrees of a variable in the GCD?

GCD degree estimate

Upper bound for GCD degree of variable x_i:

- Sample in prime field $\mathbb{Z}_p$ with $\deg_{x_i}(f \bmod p) = \deg_{x_i} f$
- Sample all other variables $x_{j \neq i}$ and do similar degree check
- Compute univariate GCD and get its degree
-
-

GCD degree estimate

Upper bound for GCD degree of variable x_i:

- Sample in prime field $\mathbb{Z}_p$ with $\deg_{x_i}(f \bmod p) = \deg_{x_i} f$
- Sample all other variables $x_{j \neq i}$ and do similar degree check
- Compute univariate GCD and get its degree
- $f(x, y) \in \mathbb{Z}_{11}[x, y] = x^2 + xy + x^2y + xy^2 = x(x + y)(y + 1)$
- $g(x, y) \in \mathbb{Z}_{11}[x, y] = 2x^2 + 2xy + x^2y + xy^2 = x(x + y)(y + 2)$

For $x = 3$ we get $\gcd\left(9 + y + 3y^2, 7 + 4y + 3y^2\right) = 3 + y$

- Degree of y in GCD is ≤ 1

Newton interpolation

$$f(x, y) = x(x + y)(y + 1)$$
$$g(x, y) = x(x + y)(y + 2)$$

- Sample at $n + 1$ points and compute the GCD over $\mathbb{Z}_{11}[x]$
- Assume for now that the leading coefficient in x is a constant
- Sample $y = 2$: $\gcd\left(6x + 3x^2, 8x + 4x^2\right) = 2x + x^2$
- Sample $y = 3$: $\gcd\left(12x + 4x^2, 15x + 5x^2\right) = 3x + x^2$
- Sample $y = 4$: $\gcd\left(20x + 5^2, 24 + 6x^2\right) = 4x + x^2$ (as a check)

Newton interpolation

- Interpolating polynomial of samples $(x_0, y_0), \ldots, (x_n, y_n)$:

$$N_n(x) = [y_0] + [y_0, y_1](x - x_0) + \ldots + [y_0, y_1, \ldots, y_n](x - x_0) \cdots (x - x_{n-1})$$

where $[a, b, \ldots]$ is the divided difference:

$$[y_k] = y_k$$

$$[y_k, \ldots, y_{k+r}] = \frac{[y_{k+1}, \ldots, y_{k+r}] - [y_k, \ldots, y_{k+r-1}]}{x_{k+r} - x_k}$$

Newton interpolation

- Interpolating polynomial of samples $(x_0, y_0), \ldots, (x_n, y_n)$:

$$N_n(x) = [y_0] + [y_0, y_1](x - x_0) + \ldots + [y_0, y_1, \ldots, y_n](x - x_0) \cdots (x - x_{n-1})$$

where $[a, b, \ldots]$ is the divided difference:

$$[y_k] = y_k$$

$$[y_k, \ldots, y_{k+r}] = \frac{[y_{k+1}, \ldots, y_{k+r}] - [y_k, \ldots, y_{k+r-1}]}{x_{k+r} - x_k}$$

- Adding new points will only affect the last term in this basis
- Alternative update rule:

$$N_{n+1}(x) = N_n(x) + (y_{n+1} - N_n(x_{n+1})) \prod_i^n \frac{x - x_i}{x_{n+1} - x_i}$$

Application

- Samples: $(2, 2x + x^2), (3, 3x + x^2), (4, 4x + x^2)$
- $[y_0] = 2x + x^2, [y_1] = 3x + x^2, [y_2] = 4x + x^2$
- $[y_0, y_1] = \frac{[y_1] - [y_0]}{3 - 2} = x$
- $[y_1, y_2] = \frac{[y_2] - [y_1]}{4 - 3} = x$
- $[y_0, y_1, y_2] = \frac{[y_1, y_2] - [y_0, y_1]}{4 - 2} = 0$
- $N(x) = 2x + x^2 + x(y - 2) = x^2 + xy$

Newton interpolation problems

$$f(x, y) \in \mathbb{Z}_{11}[x, y] = (yx - 3x + 1)(x + y)$$
$$g(x, y) \in \mathbb{Z}_{11}[x, y] = (yx - 3x + 1)(2x + y)$$

- Sample $y = 0$: $\gcd((1 - 3x)x, (1 - 3x)(2x)) = x^2 - x$, unlucky sample!
- Sample $y = 2$: $\gcd((1 - x)(2 + x), (1 - x)(2 + 2x)) = x - 1$
- Sample $y = 3$: $\gcd(3 + x, 3 + 2x) = 1$, bad sample as $\mathrm{lc}_x \bmod y - 3 = 0$!
- Sample $y = 4$: $\gcd((1 + x)(4 + x), (1 + x)(4 + 2x)) = x + 1$
- Problems are analogous to the prime sampling case
- How to scale the GCDs?

Scaling

- Compute the gcd of leading coefficient in $R\left[x_n\right]\left[x_1, \ldots x_{n-1}\right]$
- In our example: l(y) = $\gcd\left(\mathrm{lc}_x(f), \mathrm{lc}_x(g)\right) = \gcd(y - 3, 2y - 6) = y - 3$
- Rescale all GCD images: $g_i \rightarrow g_i \mathrm{l}\left(x_n\right)$
- This can be expensive if l is much larger than the actual GCD
- After interpolation, remove the content in $R\left[x_n\right]$

Multivariate GCD algorithm sketch

```
def modular_gcd(a, b, gcd_degrees, variables):
    if len(variables) == 1:
        return univariate_gcd(a, b)
    lc_gcd = univariate_gcd(lc(a), lc(b))
    v = variables[-1]
    samples = []
    for _ in range(gcd_degrees[v] + 1):
        s = random_sample(lc_gcd) # correct random sample
        g = modular_gcd(a.subs(v, s), b.subs(v, s), gcd_degrees, variables[:-1])
        g *= lc_gcd.subs(v, s) # scale image
        samples.append((s, g))
    r = newton_interpolation(samples, v)
    return remove_content(r, v)
```

Multivariate GCD algorithm sketch

```
def gcd_z(a, b, gcd_degrees, variables):
    lc_gcd = integer_gcd(lc(a), lc(b))
    g_m, m = 0, 0
    while True:
        p = random_prime(lc_gcd)
        g_p = modular_gcd(a.mod(p), b.mod(p), gcd_degrees, variables)
        if g_m == 0:
            g_m, m = g_p, p
        else:
            g_mp, m = chinese_remainder(g_m, m, g_p, p), m * p
            if g_m == g_mp:
                return remove_content(g)
            g_m = g_mp
```

Sparse interpolation

- Dense interpolation is very costly
- The two-term polynomial $x^2 + y^{10}z^{30}$ needs $11 \cdot 31$ samples
- Idea: do one dense interpolation and guess the shape of the GCD

Sparse interpolation

- Let $g = 20 + 8x^7 + 3x^7y + x^8 + y^{20} \in \mathbb{Z}[x, y]$, $a = g(1 + x + y), b = g(2 + x + y)$
- We sample 21 points in $\mathbb{Z}_7$ and get: $g_7 = -1 + x^7 + 3x^7y + x^8 + y^{20}$
- We assume no terms are missing in g_7 wrt g, so that we have the shape:

$$g = \left(\alpha_1 + \alpha_2 y^{20}\right) + \left(\beta_1 + \beta_2 y\right)x^7 + \gamma_1 x^8$$

Sparse interpolation

- Let $g = 20 + 8x^7 + 3x^7y + x^8 + y^{20} \in \mathbb{Z}[x, y]$, $a = g(1 + x + y), b = g(2 + x + y)$
- We sample 21 points in $\mathbb{Z}_7$ and get: $g_7 = -1 + x^7 + 3x^7y + x^8 + y^{20}$
- We assume no terms are missing in g_7 wrt g, so that we have the shape:

$$g = \left(\alpha_1 + \alpha_2 y^{20}\right) + \left(\beta_1 + \beta_2 y\right)x^7 + \gamma_1 x^8$$

- The number of samples needed to solve for the coefficients is the largest group length (2)
- Sample in different prime $\mathbb{Z}_{13}$:
  - $y = 1 \Rightarrow 8 + 11x^7 + x^8 \Rightarrow \alpha_1 + \alpha_2 = 8, \beta_1 + \beta_2 = 11, \gamma_1 = 1$
  - $y = 2 \Rightarrow 3 + x^7 + x^8 \Rightarrow \alpha_1 + \alpha_2 9 = 3, \beta_1 + \beta_2 2 = 1, \gamma_1 = 1$
- Solution: $g_{13} = \left(7 + y^{20}\right) + (8 + 3y)x^7 + x^8$

Sparse interpolation

- Let $g = (1 + y)x^2 + x + 1$ be the GCD
- GCD images have the form $1 \cdot x^2 + \alpha x + \beta$
- We have to scale the images before solving the system but for that we need to know the coefficient of x^2
- If there is a coefficient in $R[y][x]$ that is a single term, we can use that term to scale
- If none are present, solve a large system where the scaling parameters are additional variables

Multivariate GCD algorithm sketch

```
def modular_gcd_sparse(a, b, gcd_degrees, variables, shape):
    if len(variables) == 1:
        return univariate_gcd(a, b)
    lc_gcd = univariate_gcd(lc(a), lc(b))
    v = variables[-1]
    samples = []

    if shape is not None:
       system = []
       for _ in range(group_len(shape)):
            samples = [(v, random_sample(lc_gcd)) for v in variables[1:]]
            image = univariate_gcd(a.subs(samples), b.subs(samples))
            system.append((samples, images))
       return solve_system(system, shape)
```

Multivariate GCD algorithm sketch

```
def modular_gcd_sparse(a, b, gcd_degrees, variables, shape):
    if len(variables) == 1:
        return univariate_gcd(a, b)
    lc_gcd = univariate_gcd(lc(a), lc(b))
    v = variables[-1]
    samples = []

    if shape is not None:
       system = []
       for _ in range(group_len(shape)):
            samples = [(v, random_sample(lc_gcd)) for v in variables[1:]]
            image = univariate_gcd(a.subs(samples), b.subs(samples))
            system.append((samples, images))
       return solve_system(system, shape)
    # dense interpolation
    for _ in range(gcd_degrees[v] + 1):
        s = random_sample(lc_gcd) # correct random sample
        g = modular_gcd_sparse(a.subs(v, s), b.subs(v, s), gcd_degrees, variables[:-1], shape)
        g *= lc_gcd.subs(v, s)
        samples.append((s, g))
    r = remove_content(newton_interpolation(samples, v), v)
    shape = guess_shape(r)
    return r
```

Speed tricks

- $\gcd(p_1, \ldots, p_n)$ is with great likelihood $\gcd(p_1, \alpha_2 p_2 + \ldots + \alpha_n p_n)$ for random $\alpha_2, \ldots, \alpha_n \in R$
- If $\deg_x(a) = 0$, and $b = b_0 + b_1 x + b_2 x^2 + \ldots$, then $\gcd(a, b) = \gcd(a, b_0, \ldots)$

Factorization

Factorization

- Step 1: Filter cases where a variable does not appear in every factor, e.g.

$$f(x,y,z) = (x+y+z)\left(x^2+2y\right) = x^3+2xy+x^2y+2y^2+x^2z+2yz$$

- Compute the gcd g of all coefficients of f viewed as $\left(R\left[v_1,\ldots,v_{i-1},v_{i+1},\ldots,v_n\right]\right)\left[v_i\right]$
- If not one, we found a factor, repeat process on $\frac{f(x,y,z)}{g}$

$$\mathrm{cont}_z(f) = \gcd\left(x^3+2xy+x^2y+2y^2, x^2+2y\right) = x^2+2y$$

Factorization

- Step 1: Filter cases where a variable does not appear in every factor, e.g.

$$f(x,y,z) = (x+y+z)\left(x^2+2y\right) = x^3+2xy+x^2y+2y^2+x^2z+2yz$$

- Compute the gcd g of all coefficients of f viewed as $\left(R\left[v_1,\ldots,v_{i-1},v_{i+1},\ldots,v_n\right]\right)\left[v_i\right]$
- If not one, we found a factor, repeat process on $\frac{f(x,y,z)}{g}$

$$\mathrm{cont}_z(f) = \gcd\left(x^3+2xy+x^2y+2y^2, x^2+2y\right) = x^2+2y$$

- Step 2: compute square free factorization
-

Factorization

- Step 1: Filter cases where a variable does not appear in every factor, e.g.

$$f(x, y, z) = (x + y + z)\left(x^2 + 2y\right) = x^3 + 2xy + x^2y + 2y^2 + x^2z + 2yz$$

- Compute the gcd g of all coefficients of f viewed as $\left(R\left[v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n\right]\right)\left[v_i\right]$
- If not one, we found a factor, repeat process on $\frac{f(x,y,z)}{g}$

$$\text{cont}_z(f) = \gcd\left(x^3 + 2xy + x^2y + 2y^2, x^2 + 2y\right) = x^2 + 2y$$

- Step 2: compute square free factorization
- Step 3: Perform additional factorization per square-free factor
  - How do we do $x^2 + 3x + 2 \to (x + 1)(x + 2)$?
  - We first solve the problem in $\mathbb{Z}_p[x]$

Distinct degree factorization in finite fields

- Let $f(x) \in \mathbb{Z}_q[x]$
- Fermat theorem for finite field of size q:

$$x^q - x = \prod_a (x - a)$$

- Thus: $\gcd(f(x), x^q - x)$ will yield (the product of) all linear factors
-
-

Distinct degree factorization in finite fields

- Let $f(x) \in \mathbb{Z}_q[x]$
- Fermat theorem for finite field of size q:

$$x^q - x = \prod_a (x - a)$$

- Thus: $\gcd(f(x), x^q - x)$ will yield (the product of) all linear factors
- Holds for $q = p^d$ as well: $x^{p^d} - x = \prod_a (x - a)$ where $a \in \mathbb{Z}_{p^d}$
  - Every irreducible d-degree polynomial $m_d(x) \in \mathbb{Z}_p[x]$ has all roots in $\mathbb{Z}_{p^d}$
  - Thus: $m_d(x) \mid x^{p^d} - x$ and $\gcd\left(f(x), x^{p^d} - x\right)$ finds all degree d factors
-

Distinct degree factorization in finite fields

- Let $f(x) \in \mathbb{Z}_q[x]$
- Fermat theorem for finite field of size q:

$$x^q - x = \prod_a (x - a)$$

- Thus: $\gcd(f(x), x^q - x)$ will yield (the product of) all linear factors
- Holds for $q = p^d$ as well: $x^{p^d} - x = \prod_a (x - a)$ where $a \in \mathbb{Z}_{p^d}$
  - Every irreducible d-degree polynomial $m_d(x) \in \mathbb{Z}_p[x]$ has all roots in $\mathbb{Z}_{p^d}$
  - Thus: $m_d(x) \mid x^{p^d} - x$ and $\gcd\left(f(x), x^{p^d} - x\right)$ finds all degree d factors
- x^q can be a very high degree polynomial
  - Use $\gcd(f, y) = \gcd(f, y \bmod f)$: compute $x^q \bmod f$ using binary exponentiation

Distinct degree factorization in finite fields

```
def factor_distinct_degree(f, p):
    factors = []
    h = f.var()
    for d in range(1, f.deg() / 2):
        h = h.exp_mod(p, f) # h = h^p mod f
        g = f.gcd(h - f.var())
        if g ≠ 1:
            factors.append((d, g))
            f = f // g
    return factors
```

Equal degree factorization in finite fields

- Let $f(x) = (x - r_1)(x - r_2)..(x - r_n)$ in $\mathbb{Z}_p$
- How do we find the factors r_i?
- Binary search? Test $g = \gcd\left(f(x), (x-1)\cdots\left(x - \frac{p}{2}\right)\right)$
- If $g \neq f$ and $g \neq 1$ we will have found a subfactor
- Use Fermat again: $x^p - x = \prod_i (x - i)$

Equal degree factorization in finite fields

- Let $f(x) = (x - r_1)(x - r_2)..(x - r_n)$ in $\mathbb{Z}_p$
- How do we find the factors r_i?
- Binary search? Test $g = \gcd\left(f(x), (x-1)\cdots\left(x - \frac{p}{2}\right)\right)$
- If $g \neq f$ and $g \neq 1$ we will have found a subfactor
- Use Fermat again: $x^p - x = \prod_i (x - i)$

$$x^p - x = x \underbrace{\left(x^{\frac{p-1}{2}} - 1\right)}_{\text{quadratic residues}} \left(x^{\frac{p-1}{2}} + 1\right)$$

- A random 50% of the elements are quadratic residues
- For random shift t, $\gcd\left(f(x), (x-t)^{\frac{p-1}{2}} - 1\right)$ has $1 - \frac{1}{2^{n-1}}$ chance of finding a true subfactor

Equal degree factorization in finite fields

- For higher degree d use $\gcd\left(f(x), t^{\frac{p^d-1}{2}} - 1\right)$ with $t \in \mathbb{Z}_{p^d}$ where t is a random polynomial of degree $< 2d$.

```
def factor_equal_degree(f, p, d):
    if f.deg() == d:
        return [f]
    while True:
        h = random_poly(2 * d)
        b = exp_mod(h, (p**d - 1) // 2, f)
        g = f.gcd(b - 1)
        if g ≠ 1 and g ≠ f:
            return factor_equal_degree(g) + factor_equal_degree(f // g)
```

- Also works in $GF(q, k)$: substitute p by q^k

Factorization over the integers

- Can we factor in $\mathbb{Z}[x]$ by factoring in $\mathbb{Z}_p[x]$ and reconstructing the result?
- $x^2 - 2x + 3$ is irreducible in $\mathbb{Z}[x]$ but factors as $(x + 2)(x + 5)$ in $\mathbb{Z}_{11}[x]$ and $(x + 30)(x + 75) \in \mathbb{Z}_{103}[x]$
- Try all combinations of factors and do a division test
  - There is a bound for the largest coefficient in a factorization of $f(x)$: $\sqrt{n+1}2^n \, |f|_\infty$
  - Try all combinations of factors such that they yield coefficients smaller than this bound
- Chinese Remainder theorem cannot be used as the factorizations in $\mathbb{Z}_{11}[x]$ cannot be merged with those of $\mathbb{Z}_{103}[x]$

Large prime algorithm

```
def factor(f):
    p = random_prime_above(f.deg() * 2**f.deg() * f.max_coeff()) # pick a large
prime p much larger than the largest coefficient
    fp = f.mod(p)
    if fp.gcd(fp.derivative()) ≠ 1: # make sure f mod p is still square-free
        return factor(f)
    prime_factors = fp.factor() # monic factors in Z_p[x]
    factors, i = [], 0
    for i in 0..len(prime_factors):
        for s in combinations(prime_factors, i):
            g = f.lc() * product(a for a in s)
            h = f.lc() * product(a for a in prime_factors if a not in s)
            if g * h == f.lc() * f: # one could first check if the coefficients
are smaller than the bound
                factors.append(g.primitive())
                f ÷= g
                prime_factors -= s
    return factors
```

Example

- $f = 6x^4 + 5x^3 + 15x^2 + 5x + 4 \in \mathbb{Z}[x]$
- Pick prime 6473
- Factor: $f \equiv 6(x - 819)(x + 605)(x + 2632)(x + 2977) \bmod 6473$
- No single factor divides f
- Try combinations:

$$g = 6(x - 819)(x + 605) = 6x^2 - 1284x - 1863 \bmod 6473$$

$$h = 6(x + 2632)(x + 2977) = 6x^2 + 1289x - 615 \bmod 6473$$

- Clearly does not divide (coefficients too large)

$$g = 6(x - 819)(x + 2977) = 6x^2 + 2x + 2 \bmod 6473$$

$$h = 6(x + 605)(x + 2632) = 6x^2 + 3x + 12 \bmod 6473$$

- Factorization found: $\left(3x^2 + x + 1\right)\left(2x^2 + x + 4\right)$

Hensel lifting

- Assume we have a factorization: $f \equiv g^{(k)} h^{(k)} \bmod p^{k+1}$
- $k = 0$ can be computed with finite field factorization methods
- Error: $f - g^{(k)} h^{(k)} = e_{k+1} p^{k+1} + \ldots$

$$g^{(k+1)} = g^{(k)} + g_{k+1} p^{k+1}$$

$$h^{(k+1)} = h^{(k)} + h_{k+1} p^{k+1}$$

$$f - g^{(k+1)} h^{(k+1)} \bmod p^{k+2} = \left(e_{k+1} - g_{k+1} h^{(k)} - h_{k+1} g^{(k)}\right) p^{k+1} \bmod p^{k+2} = 0 \bmod p^{k+2}$$

Hensel lifting

- Assume we have a factorization: $f \equiv g^{(k)} h^{(k)} \bmod p^{k+1}$
- $k = 0$ can be computed with finite field factorization methods
- Error: $f - g^{(k)} h^{(k)} = e_{k+1} p^{k+1} + \ldots$

$$g^{(k+1)} = g^{(k)} + g_{k+1} p^{k+1}$$

$$h^{(k+1)} = h^{(k)} + h_{k+1} p^{k+1}$$

$$f - g^{(k+1)} h^{(k+1)} \bmod p^{k+2} = \left(e_{k+1} - g_{k+1} h^{(k)} - h_{k+1} g^{(k)}\right) p^{k+1} \bmod p^{k+2} = 0 \bmod p^{k+2}$$

- Thus: $e_{k+1} \equiv g_{k+1} h^{(k)} + h_{k+1} g^{(k)} \bmod p \equiv g_{k+1} h^{(0)} + h_{k+1} g^{(0)}$
- Solve once with EEA: $g^{(0)} s + h^{(0)} t = 1$

$$g^{(k+1)} = g^{(k)} + e_{k+1} t p^{k+1}$$

$$h^{(k+1)} = h^{(k)} + e_{k+1} s p^{k+1}$$

- Repeat until $f - g^{(a)} h^{(a)} = 0$ in $\mathbb{Z}[x]$ or until a degree bound is reached

Hensel lifting example

$$f(x) = x^4 - 1 \equiv \underbrace{\left(x^3 + 2x^2 - x - 2\right)}_{g_0}\underbrace{(x - 2)}_{h_0} \bmod 5$$

- $\operatorname{egcd}\left(g_0, h_0\right) \Rightarrow s = -2, t = 2x^2 - 2x - 1$
- $e_1 = \frac{f - g_0 h_0}{5} \bmod 5 = x^2 - 1$
- $g^{(1)} = g_0 + te_1 \cdot 5 = 10x^4 - 9x^3 - 13x^2 + 9x + 3$
- $h^{(1)} = h_0 + se_1 \cdot 5 = -10x^2 + x + 8$
- $f - g^{(1)}h^{(1)} = 25\left(4x^6 - 4x^5 - 8x^4 + 7x^3 + x^2 - 3x - 1\right) \equiv 0 \bmod 25$
- Works, but degree of $g^{(1)}$ is too high!
- If $\left(g_k, h_k\right)$ is a solution, then $\left(g_k + qg^{(0)}, h_k - qh^{(0)}\right)$ is also a solution for every q
- Determine q, r such that $h_k = qh^{(0)} + r$

Hensel lifting example fixed

$$f(x) = x^4 - 1 \equiv \underbrace{\left(x^3 + 2x^2 - x - 2\right)}_{g_0}\underbrace{(x - 2)}_{h_0} \bmod 5$$

- $\operatorname{egcd}\left(g_0, h_0\right) \Rightarrow s = -2, t = 2x^2 - 2x - 1$
- $e_1 = \frac{f - g_0 h_0}{5} \bmod 5 = x^2 - 1$
- $h_1 = e_1 s = -2x^2 + 2 = (1 + 3x)h_0 + 4 \Rightarrow q = 1 + 3x, r = 4$
- $g_1 = te_1 + qg_0 \bmod p = x^2 + 4$
- $g^{(1)} = g^{(0)} + g_1 \cdot 5 = x^3 + 7x^2 - x - 7 \bmod 25$
- $h^{(1)} = h^{(0)} + r \cdot 5 = x - 7 \bmod 25$

Hensel step

For $f \equiv g^{(k)}h^{(k)} \bmod p^{k+1}$, h_0 monic, $sg_0 + th_0 \equiv 1 \bmod p$, yield $f \equiv g^{(k+1)}h^{(k+1)} \bmod p^{k+2}$:

```
def hensel_step(f, g, h, g0, h0, s, t, p, k):
  e = ((f - g * h) / p**(k+1)).mod(p)
  (q, r) = (s * e).quot_rem(h0) # lc(h0) = 1, so division is defined
  g += (t * e + q * g0) * p**(k+1)
  h += r * p**(k+1)
  return (g, h)
```

Hensel step

Also lift s and t such that $sg^{(k+1)} + th^{(k+1)} \equiv 1 \bmod p^{k+2}$:

```
def hensel_step(f, g, h, g0, h0, s, t, p, k):
  e = ((f - g * h) / p**(k+1)).mod(p)
  (q, r) = (s * e).quot_rem(h0)
  g += (t * e + q * g0) * p**(k+1)
  h += r * p**(k+1)

  e2 = ((s * g + t * h - 1) / p**(k+1)).mod(p)
  (q, r) = (s * e2).quot_rem(h0)
  s -= r * p**(k+1)
  t -= (t * e2 + q * g) * p**(k+1)
  return (g, h, s, t)
```

Useful in the near future!

Coefficient correction

- Hensel lift can be stopped when the error is 0 in $\mathbb{Z}[x]$
- However, the error can never be 0 when the leading coefficient of lc_h is not 1, as lc_h is never updated:

$$f(x) = (2x + 5)\left(6x^2 - 10x + 7\right)$$

p	g	h	$e = f - gh$
5	$2x$	$x^2 + 2$	$10x^3 + 10x^2 - 40x + 35$
25	$12x + 5$	$x^2 - 10x - 3$	$125x^2 + 50x + 50$
125	$12x + 30$	$x^2 - 210x - 103$	$2500x^2 + 7500x + 3125$

- Fixed in practice by multiplying with lc_f or the true leading coefficient of every factor during the lift
- Slower: stop when the p^k is larger than a bound on the coefficients of the factors

Hensel lifting for variables

- Suppose we have $f \in \mathbb{Z}_{p^k}[x, y]$, we can solve $f(x, \alpha) = g(x, \alpha)h(x, \alpha) \in \mathbb{Z}_{p^k}[x]$
- Can we lift this to $f = \tilde{g}(x, y)\tilde{h}(x, y)$?
- Remember: $f(x, \alpha) = f(x, y) \bmod y - \alpha$, so we can lift to $(y - \alpha)^k$!
-

Hensel lifting for variables

- Suppose we have $f \in \mathbb{Z}_{p^k}[x, y]$, we can solve $f(x, \alpha) = g(x, \alpha)h(x, \alpha) \in \mathbb{Z}_{p^k}[x]$
- Can we lift this to $f = \tilde{g}(x, y)\tilde{h}(x, y)$?
- Remember: $f(x, \alpha) = f(x, y) \bmod y - \alpha$, so we can lift to $(y - \alpha)^k$!
- Error: $f - g^{(k)}h^{(k)} = e_{k+1}(y - \alpha)^{k+1} + \ldots$

$$g^{(k+1)} = g^{(k)} + g_{k+1}(y - \alpha)^{k+1}$$

$$h^{(k+1)} = h^{(k)} + h_{k+1}(y - \alpha)^{k+1}$$

$$f - g^{(k+1)}h^{(k+1)} \bmod (y - \alpha)^{k+2} = \left(e_{k+1} - g_k h^{(k)} - h_k g^{(k)}\right)(y - \alpha)^{k+1} \bmod (y - \alpha)^{k+2} = 0 \bmod (y - \alpha)^{k+2}$$

- Thus: $e_{k+1} \equiv g_k h^{(k)} + h_k g^{(k)} \bmod (y - \alpha) \equiv g_k h(x, \alpha) + h_k g(x, \alpha)$
- Solve equation in $\mathbb{Z}_{p^k}[x]$ using the Hensel lifted s and t

Hensel lifting for variables

- Process is fully recursive: for $f\left(x_1, \ldots x_n\right)$ pick $\alpha_2, .. \alpha_n$ and set $\tilde{f} = f\left(x_1, \alpha_2, \ldots, \alpha_n\right)$
- If $\deg_{x_1} \tilde{f}\left(x_1\right) \neq \deg_x f\left(x_1, \ldots, x_n\right)$ or $\tilde{f}\left(x_1\right)$ is not square free: pick new α_i
- Subproblem $e_{k+1} \equiv g_k h\left(x_1, \ldots, x_j, \alpha_{j+1}, \ldots, \alpha_n\right) + h_k g\left(x_1, \ldots, x_j, \alpha_{j+1}, \ldots, \alpha_n\right) \bmod p^k$
  - Solve $\tilde{e}_{k+1} \equiv g_k h\left(x_1, \ldots, \alpha_j, \ldots, \alpha_n\right) + h_k g\left(x_1, \ldots, \alpha_j, \ldots, \alpha_n\right) \bmod p^k$
  - Lift to obtain $g_k\left(x_1, \ldots, x_j \alpha_{j+1}, \ldots, \alpha_n\right)$, $h_k\left(x_1, \ldots, x_j, \alpha_{j+1}, \ldots, \alpha_n\right)$, $s\left(x_1, \ldots, x_j, \alpha_{j+1}, \ldots, \alpha_n\right)$ and $t\left(x_1, \ldots, x_j, \alpha_{j+1}, \ldots, \alpha_n\right)$

Multiple factors

For monic factors f_i of f with $f \equiv lc(f) \prod_i f_i \bmod p$, yield $f \equiv lc(f) \prod_i \tilde{f}_i \bmod p^k$

```
def hensel_lift(f, factors, p, k):
  if len(factors) == 1:
    return factors

  g = f.lc() * factors[:len(factors) / 2]
  h = factors[len(factors) / 2:]
  s, t = egcd(g.mod(p), h.mod(p))
  for _ in range(k):
    (g, h, s, t) = hensel_step(f, g, h, s, t, p) # lift to p^k
  lg = hensel_lift(g, factors[:len(factors) / 2], p, k)
  lh = hensel_lift(h, factors[len(factors) / 2:], p, k)
  return lg + lh
```

Field extensions

Field extensions

- A field can be extended by adjoining a new element α that is not in the field
- For example $\mathbb{Q}(x)$ is a rational polynomial in x
- $\mathbb{Q}(\pi)$ is a rational polynomial in π
- $\mathbb{Q}\left(\sqrt{2}\right)$ can be written as the polynomial $a + b\sqrt{2}$ with $a, b \in \mathbb{Q}$
  - Rational polynomial not needed, as $\frac{1}{\sqrt{2}} = \frac{\sqrt{2}}{2}$

Algebraic numbers

- $\mathbb{Q}[x]$ is a Euclidean domain, so we can do $f(x) \bmod m(x)$ with $f, m \in \mathbb{Q}[x]$
- If $m(x)$ is irreducible over $\mathbb{Q}$, then $\mathbb{Q}[x] \bmod m(x)$ is a field
- Inverses can be computed through the Extended Euclidean algorithm
- This field is called an algebraic number field and $m(x)$ is the minimal polynomial
- For example: $\alpha = \sqrt{2}$ is a root of $m(x) = x^2 - 2 \in \mathbb{Q}(x)$, as $m(\alpha) = 0$
- Polynomial representation: $c_0 + c_1\alpha \bmod m(\alpha)$
- $\alpha \cdot \alpha \bmod m(\alpha) = 2$
- We write the field as $\mathbb{Q}(\alpha) = \frac{\mathbb{Q}[\alpha]}{m(\alpha)}$

Algebraic numbers example

- Multiple extensions: $i\sqrt{2} + \sqrt{3} \in \mathbb{Q}\left(i, \sqrt{2}, \sqrt{3}\right)$
- They can depend on each other (a tower)
- For example $\beta = \sqrt{\sqrt{2} - 1}$:
  - $m(x) = x^2 - 2$, $m(\alpha) = 0$
  - $n(x) = x^2 - \alpha - 1$, $n(\beta) = 0$
- To map to and from $\mathbb{C}$, store root number, e.g. $\sqrt{2} = \text{root}\left(x^2 - 2, 2\right)$
- Computations in $\mathbb{Q}(\alpha, \beta)$:

$$
\begin{aligned}
r_1 &= (\alpha) + (\alpha + 1)\beta \bmod n(\beta) \bmod m(\alpha) \\
r_1^2 &= 2 + (4 + 2\alpha)\beta + (3 + 2\alpha)\beta^2 \bmod n(\beta) \bmod m(\alpha) \\
&= 2 + (4 + 2\alpha)\beta + (3 + 2\alpha)(\alpha + 1) \bmod m(\alpha) \\
&= 9 + 5\alpha + (4 + 2\alpha)\beta
\end{aligned}
$$

Algebraic numbers example

- We want to represent $\alpha = \sqrt{2}, \beta = \sqrt{3}, \gamma = \sqrt{6}$: $\mathbb{Q}(\alpha, \beta, \gamma)$?
- Not a field because $\sqrt{6} = \sqrt{2}\sqrt{3}$!
- $x^2 - 6$ is not irreducible in $\mathbb{Q}(\alpha, \beta)$: it factors as $(x - \alpha\beta)(x + \alpha\beta)$
- To show this we need to be able to factor in a field extension
- We first fuse two extensions into one

Adjoining extensions

- Find a γ such that $\mathbb{Q}\left(\sqrt{2}, \sqrt{3}\right) = \mathbb{Q}(\gamma)$
- Primitive element theorem: $\gamma = c_1\sqrt{2} + c_2\sqrt{3}$ where c_1 and c_2 are such that it leaves all roots distinct
- We pick $\gamma = \sqrt{2} + \sqrt{3} = \alpha + \beta$

$$\alpha^2 - 2 = 0,$$
$$\beta^2 - 3 = 0 = (\gamma - \alpha)^2 - 3 = 0$$

- Needs to have common root, so use resultant in α:

$$\operatorname{res}_\alpha\left(\alpha^2 - 2, (\gamma - \alpha)^2 - 3\right) = \gamma^4 - 10\gamma^2 + 1$$

- This is the new minimal polynomial of γ
- $\gcd\left((\gamma - \alpha)^2 - 3, \alpha^2 - 2\right) \in \mathbb{Q}(\gamma) = c_0 + c_1\alpha \Rightarrow \alpha = -\frac{9}{2}\gamma + \frac{1}{2}\gamma^3$
- $\beta = \gamma - \alpha = \frac{11}{2}\gamma + \frac{1}{2}\gamma^3$

Factoring in algebraic extension

- **Norm** of algebraic number $b(x)$: $\text{norm}(b) = \text{res}_x(b(x), m(x)) \in \mathbb{Q}$
- Norm of polynomial with algebraic number coefficients $p \in \mathbb{Q}(\alpha)[z]$:

$$\text{norm}(p) = \text{res}_x(p(x, z), m(x)) \in \mathbb{Q}[z]$$

- For example: $\text{norm}\left(1 + \sqrt{2}z\right) = \text{res}_\alpha\left(1 + \alpha z, \alpha^2 - 2\right) = 1 - 2z^2$
- $\text{norm}(ab) = \text{norm}(a)\text{norm}(b)$
-
-

Factoring in algebraic extension

- **Norm** of algebraic number $b(x)$: $\text{norm}(b) = \text{res}_x(b(x), m(x)) \in \mathbb{Q}$
- Norm of polynomial with algebraic number coefficients $p \in \mathbb{Q}(\alpha)[z]$:

$$\text{norm}(p) = \text{res}_x(p(x,z), m(x)) \in \mathbb{Q}[z]$$

- For example: $\text{norm}\left(1 + \sqrt{2}z\right) = \text{res}_\alpha\left(1 + \alpha z, \alpha^2 - 2\right) = 1 - 2z^2$
- $\text{norm}(ab) = \text{norm}(a)\text{norm}(b)$
- Thus if $f \in F(\alpha)[z]$ factors, then the norm factors too and vice versa!
- Let $\text{norm}(f) = f_1 \ldots f_n \in F(z)$, then:

$$f = \prod_i^n \gcd\left(f, f_i\right)$$

is a factorization of f in $F(\alpha)[z]$, gcd taken over $F(\alpha)[z]$

- Norm must be square-free, otherwise shift $z \to z - \alpha$ until it is square-free

Function extension

- Functions can be added as a field extension
- For example, $\mathbb{Q}(x, \log(x))$
- $f = \exp(x) + \exp(2x) + \exp(3x)$ could be described as $\mathbb{Q}\left(\exp(x), \exp(2x), \exp\left(\frac{x}{2}\right)\right)$
- Each new 'transcendental' function should be an algebraically independent symbol θ_i, so that we work in a field
- $\theta_1 = \exp(x), \theta_2 = \exp(2x), \theta_3 = \exp(3x), f = \theta_1 + \theta_2 + \theta_3$
- We see that $\theta_2 = \theta_1^2, \theta_3 = \theta_1^3$, thus $f \in \mathbb{Q}\left(x, \theta_1\right)$

Structure theorem

- Let $F_n = F\left(x, \theta_1, \ldots \theta_n\right)$, θ_i is either
  - algebraic over F_{i-1}
  - w_i, with $w_i = \log\left(u_i\right)$ and $u_i \in F_{i-1}$
  - u_i, with $u_i = \exp\left(w_i\right)$ and $w_i \in F_{i-1}$

then

- $g = \log(f)$ with f not constant is independent iff there is no $r = f^k \prod u_j^{k_j} \in F, k, k_j \in \mathbb{Z}, k \neq 0$
  - Otherwise: $g = \log(f) = \log\left(\left(\frac{r}{\prod u_j^{k_j}}\right)^{\frac{1}{k}}\right) = \frac{1}{k}\left(\log(r) - \sum_j k_j \theta_j\right)$
- $f = \exp(g)$ with g not constant is independent iff there is no $r = g + \sum c_i w_i \in F, c_i \in \mathbb{Q}$
  - Otherwise: $f = \exp\left(r - \sum c_i w_i\right) = \exp(r) \prod_i \exp\left(w_i\right)^{-c_i} = \exp(r) \prod_i \theta_i^{-c_i}$

Example

- $f = \exp(\log(x))$
- $\theta_1 = \log(x)$, $\theta_2 = \exp(\theta_1)$
- $\theta_1 + c_1\theta_1 \in F \Rightarrow c_1 = -1$
- Thus: $f = \exp(\log(x)) = x \in \mathbb{Q}(x)$

Example

Let $g = \log\left(\sqrt{x^2+1}+x\right) + \log\left(\sqrt{x^2+1}-x\right)$

- $\theta_1 = \sqrt{x^2+1}, \theta_2 = \log(\theta_1 + x), \theta_3 = \log(\theta_1 - x)$
- Check if θ_3 is independent over $\mathbb{Q}(x, \theta_1, \theta_2)$: there should be no $k \in \mathbb{Z}$ such that

$$r = (\theta_1 - x)^k (\theta_1 + x)^{k_2} \in \mathbb{Q}$$

Differentiate:

$$0 = \frac{r'}{r} = k\frac{(\theta_1 - x)'}{\theta_1 - x} + k_2\frac{(\theta_1 + x)'}{\theta_1 + x}$$

- Solve: $k = k_2 = r = 1$, so:

$$\theta_3 = \frac{1}{k}\left(\log(r) - \sum_j k_j\theta_j\right) = -\theta_2$$

Symbolic integration

Indefinite integration

- Find an antiderivative $g(x)$ of $f(x)$ so that $g(x)' = f(x)$
- The constant of integration is not considered, which would contain branch cut information
- Polynomial integration is straightforward:

$$\int ax^n \, \mathrm{d}x = \frac{a}{n+1} x^{n+1}$$

- a can be in any field that does not contain x

Rational function integration

$$\int \frac{a(x)}{b(x)}\,\mathrm{d}x$$

- Euclidean division:

$$\int p(x)\,\mathrm{d}x + \int \frac{c(x)}{b(x)}\,\mathrm{d}x$$

where $p(x)$ is polynomial, $\deg_x c(x) < \deg_x b(x)$

- Square-free factor $b(x)$ and partial fraction:

$$\int \frac{c(x)}{b(x)}\,\mathrm{d}x = \sum_i \int \frac{c_i(x)}{b_i(x)^i}\,\mathrm{d}x$$

Hermite reduction

- Since $b_i(x)$ is square-free, $\gcd\left(b_i(x), \frac{\mathrm{d}}{\mathrm{d}x} b_i(x)\right) = 1$
- Use EEA to find $s(x)b_i(x) + t(x)b_i(x)' = c_i(x)$

$$\int \frac{c_i(x)}{b_i(x)^i}\,\mathrm{d}x = \int \left(\frac{s(x)}{b_i(x)^{i-1}} + \frac{t(x)b_i(x)'}{b_i(x)^i} \right) \mathrm{d}x$$

Hermite reduction

- Since $b_i(x)$ is square-free, $\gcd\left(b_i(x), \frac{\mathrm{d}}{\mathrm{d}x} b_i(x)\right) = 1$
- Use EEA to find $s(x)b_i(x) + t(x)b_i(x)' = c_i(x)$

$$\int \frac{c_i(x)}{b_i(x)^i}\,\mathrm{d}x = \int \left(\frac{s(x)}{b_i(x)^{i-1}} + \frac{t(x)b_i(x)'}{b_i(x)^i} \right) \mathrm{d}x$$

- Partial integration: $\int uv' = uv - \int u'v$ with $u = t, v = \frac{-1}{(i-1)b_i(x)^{i-1}}$

$$\int \frac{c_i(x)}{b_i(x)^i}\,\mathrm{d}x = \frac{\frac{-t(x)}{i-1}}{b_i(x)^{i-1}} + \int \frac{s(x) + \frac{t(x)'}{i-1}}{b_i(x)^{i-1}}\,\mathrm{d}x$$

- Repeat until $i = 1$

Logarithmic part

- For b monic, square-free, $b = \prod_i (x - \beta_i)$ in some field extension $K[x]$

$$\int \frac{a}{b} = \sum_i \int \frac{a_i}{x - \beta_i} = \sum_i c_i \log(x - \beta_i)$$

Logarithmic part

- For b monic, square-free, $b = \prod_i (x - \beta_i)$ in some field extension $K[x]$

$$\int \frac{a}{b} = \sum_i \int \frac{a_i}{x - \beta_i} = \sum_i c_i \log(x - \beta_i)$$

- This extension could be huge, and larger than needed as terms may fuse in the sum:

$$c \log(x + i) + c \log(x - i) = c \log(x^2 + 1) \Rightarrow \text{no } i \text{ needed}$$

- We need the smaller extension K' that contains unique c_i, the residues of $\frac{a}{b}$

Rothstein/Trager algorithm

$$\int \frac{a}{b} = \sum_i c_i \log(v_i)$$

- c_i are distinct roots of $\operatorname{res}_x(a(x) - zb'(x), b(x))$
- $v_i = \gcd\left(a - c_i b', b\right)$

Risch integration

- Elementary extension: tower of algebraic, logarithmic and exponential extension
- Differential field: operator D exists that acts as a derivative
- Liouville principle: for F a differential field with constant field K, $f \in F$ and if $g' = f$ has a solution with $g \in G$ where G is an elementary extension of F, then there are $v_i \in F$ and $c_i \in K$ such that

$$\int f = v_0 + \sum_i c_i \log(v_i)$$

- No new exponential extensions are allowed to appear, and only new linear logarithmic ones
- Step 1: convert all special functions into exp-log notation

Risch integration

- Each transcendental θ_i should be algebraically independent (use the structure theorem)
- View integration as a rational function in the last extension:

$$f(\theta_n) = \frac{p(\theta_n)}{q(\theta_n)} \in K(x, \theta_1, \ldots, \theta_{n-1})(\theta_n)$$

- Properly normalize: $\gcd(p(\theta_n), q(\theta_n)) = 1$, $\mathrm{lc}(q) = 1$
-

Risch integration

- Each transcendental θ_i should be algebraically independent (use the structure theorem)
- View integration as a rational function in the last extension:

$$f(\theta_n) = \frac{p(\theta_n)}{q(\theta_n)} \in K(x, \theta_1, \ldots, \theta_{n-1})(\theta_n)$$

- Properly normalize: $\gcd(p(\theta_n), q(\theta_n)) = 1$, $\operatorname{lc}(q) = 1$
- Example, where $\theta_1 = \log(x), \theta_2 = \exp(x)$:

$$\frac{\exp(2x) + \left(2 + \sqrt{2}\right)\exp(x) + 2\sqrt{2}}{x^2\exp(2x) + (2 + \log(x))x\exp(x) + 2\log(x)} = \frac{\left(\theta_2^2 + \sqrt{2}\right)/x}{\theta_2 + \frac{\theta_1}{x}} \in \mathbb{Q}\left(\sqrt{2}\right)(x, \theta_1)(\theta_2)$$

Logarithmic extension integration

- θ is logarithmic, $\theta' = \frac{u'}{u}, u \in F_{n-1}$
- Euclidean division:

$$\int f(\theta) = \int \frac{p(\theta)}{q(\theta)} = \int s(\theta) + \int \frac{r(\theta)}{q(\theta)}$$

- Integrate polynomial part $s(\theta)$ first
-
-
-

Logarithmic extension integration

- θ is logarithmic, $\theta' = \frac{u'}{u}, u \in F_{n-1}$
- Euclidean division:

$$\int f(\theta) = \int \frac{p(\theta)}{q(\theta)} = \int s(\theta) + \int \frac{r(\theta)}{q(\theta)}$$

- Integrate polynomial part $s(\theta)$ first
- Integral elementary when $s(\theta) = v_0(\theta)' + \sum_i c_i \frac{v_i'(\theta)}{v_i(\theta)}$ (Liouville)
- Since $s(\theta)$ is a polynomial:
  - $v_0(\theta)$ must be polynomial in θ
  - Maximum degree of v_0 is $\deg(s) + 1$
  - $v_i(\theta)$ must be independent of θ
- Solve system:

$$s_l\theta^l + \ldots + s_0 = \left(q_{l+1}\theta^{l+1} + \ldots + q_0\right)' + \sum c_i \frac{v_i'}{v_i}$$

Logarithmic extension integration

$$s_l\theta^l + \ldots + s_0 = \left(q_{l+1}\theta^{l+1} + \ldots + q_0\right)' + \sum c_i \frac{v_i'}{v_i}$$

$$\begin{aligned} 0 &= q_{l+1}' \\ s_l &= (l+1)q_{l+1}\theta' + q_l' \\ &\vdots \\ s_0 &= q_1\theta' + q_0' + \sum c_i \frac{v_i'}{v_i} \end{aligned}$$

- Integrate the relations top to bottom and substitute:

$$q_{l+1} = b_{l+1} \in K, \text{constant of integration}$$

$$\int s_l = (l+1)b_{l+1}\theta + q_l$$

Recursive integration

$$\int s_l = (l+1)b_{l+1}\theta + q_l$$

- Call algorithm recursively for $\int s_l$ in F_{n-1}
- The solutions of $\int s_l$ must be of the form for the integral to be elementary:

$$\int s_l = c_l\theta + d_l$$

- Then $b_{l+1} = \frac{c_l}{l+1}, q_l = d_l + b_l$ (b_l is the constant of integration)

Continuing solving

Next level condition:

$$s_{l-1} = l q_l \theta' + q'_{l-1}$$

Fill in:

$$s_{l-1} = l\left(d_l + b_l\right)\theta' + q'_{l-1}$$
$$s_{l-1} - l d_l \theta' = l b_l \theta' + q'_{l-1}$$
$$\int \left(s_{l-1} - l d_l \theta'\right) = l b_l \theta + q_{l-1}$$

- Left-hand side is in F_{n-1} ($\theta = \log(u) \Rightarrow \theta' = \frac{u'}{u}$), compute recursively
- Continue until s_0, where additional log terms may appear

Example

- $f = \log(x) = \theta \in \mathbb{Q}(x, \theta)$
- If elementary: $\int \theta = q_2\theta^2 + q_1\theta + q_0 + \sum_i c_i \log(v_i)$
- $$0 = q_{2'} \qquad q_2 = b_2 \in \mathbb{Q},$$
  $$1 = 2b_2\theta' + q_1' \qquad x + b_1 = 2b_2\theta + q_1 \Rightarrow b_2 = 0, q_1 = x + b_1$$
  $$0 = q_1\theta' + b_0 + \sum c_i \frac{v_{i'}}{v_i} \qquad 0 = \int -x\theta' + b_1\theta + b_0 + \sum_i c_i \log(v_i)$$
- $\int -x\theta' = \int -1 = -x$
- Thus: $b_1 = 0, b_0 = -x, c_i = 0$ and

$$\int \log(x) = x\log(x) - x$$

Example 2

- $f = \log(\log(x)) = \theta_2 \in \mathbb{Q}\left(x, \theta_1 = \log(x), \theta_2 = \log\left(\theta_1\right)\right)$
- Steps are the same as last example, until:

$$\int -x\theta_2' = b_1\theta_2 + b_0 + \sum_i c_i \log\left(v_i\right)$$

- Subintegral: $\int -x\frac{\theta_1'}{\theta_1} = -\frac{1}{\log(x)}$ is not elementary as we will see

The rational logarithmic part

- $\int \frac{r(\theta)}{q(\theta)}$ with $\deg(r) < \deg(q)$
- Apply Hermite reduction: square-free factor $q(\theta)$ so that $\gcd\left(q_i(\theta), \frac{d}{d\theta} q_i(\theta)\right) = 1$
- For the reduction we need $\gcd\left(q_i, q_i'\right) = 1$, which holds for a log extension
- We end up with a sum of $\frac{a(\theta)}{b(\theta)}$ with square-free b
- Apply Rothstein-Trager algorithm:
  - $\int \frac{a(\theta)}{b(\theta)}$ is elementary if all roots of the resultant are constant:

$$R(z) = \operatorname{res}_\theta(a(\theta) - z b(\theta)', b(\theta)) \in F[z]$$

  - Then:

$$\frac{a(\theta)}{b(\theta)} = \sum_i c_i \frac{v_i(\theta)'}{v_i(\theta)}$$

where c_i are the distinct roots of $R(z)$ and $v_i(\theta) = \gcd\left(a(\theta) - c_i b(\theta)', b(\theta)\right)$

Examples

- $f = \frac{1}{\log(x)} = \frac{1}{\theta} \in \mathbb{Q}(x, \theta = \log(x))$
  - $R(z) = \operatorname{res}_\theta\left(1 - \frac{z}{x}, \theta\right) = 1 - \frac{z}{x}$
  - Non-constant root, so not elementary
-

Examples

- $f = \frac{1}{\log(x)} = \frac{1}{\theta} \in \mathbb{Q}(x, \theta = \log(x))$
  - $R(z) = \operatorname{res}_\theta\left(1 - \frac{z}{x}, \theta\right) = 1 - \frac{z}{x}$
  - Non-constant root, so not elementary
- $f = \frac{1}{x \log(x)} = \frac{1}{x\theta} \in \mathbb{Q}(x, \theta = \log(x))$
  - $R(z) = \operatorname{res}_\theta\left(\frac{1}{x} - \frac{z}{x}, \theta\right) = \frac{1}{x} - \frac{z}{x}$
  - Root is constant ($z = 1$), so elementary
  - $c_1 = 1, v_1 = \gcd\left(\frac{1}{x} - \frac{1}{x}, \theta\right) = \theta$
  - So: $f = c_1 \log\left(v_1\right) = \log(\log(x))$

Exponential extension integration

- Exponential extension: $\theta = \exp(u), \theta' = \theta u', u \in F_{n-1}$

$$f(\theta) = s(\theta) + \frac{r(\theta)}{q(\theta)}$$

- Square-free decomposition of $q(\theta)$ does not satisfy $\gcd\left(q_i, q_i'\right) = 1$ for $q_i = \theta^k$
- Extract leading factor θ^l from $q(\theta)$, so we get

$$f(\theta) = s(\theta) + \frac{w(\theta)}{\theta^l} + \frac{\tilde{r}(\theta)}{\tilde{q}(\theta)}$$

Exponential extension integration

- Exponential extension: $\theta = \exp(u), \theta' = \theta u', u \in F_{n-1}$

$$f(\theta) = s(\theta) + \frac{r(\theta)}{q(\theta)}$$

- Square-free decomposition of $q(\theta)$ does not satisfy $\gcd\left(q_i, q_i'\right) = 1$ for $q_i = \theta^k$
- Extract leading factor θ^l from $q(\theta)$, so we get

$$f(\theta) = s(\theta) + \frac{w(\theta)}{\theta^l} + \frac{\tilde{r}(\theta)}{\tilde{q}(\theta)}$$

- Reduction of rational part has an additional term that makes sure that $\deg(\tilde{r}) < \deg(\tilde{q})$:

$$\int \frac{\tilde{r}(\theta)}{\tilde{q}(\theta)} = \sum_i c_i \log\left(v_i(\theta)\right) - u \sum_i c_i \deg\left(v_i(\theta)\right)$$

- $\log(\theta + 1)' = \frac{\theta u'}{1+\theta}$ is not a proper rational, but $\log(\theta + 1)' - u' = -\frac{u'}{1+\theta}$ is

Example

$$f(x) = \frac{1}{\exp(x) + 1} = \frac{1}{\theta + 1}$$

$$R(z) = \operatorname{res}_\theta(1 - z\theta, \theta + 1) = -1 - z \Rightarrow \text{elementary}$$

$$c_1 = -1, v_1 = \theta + 1$$

$$\int f(x) = x - \log(\exp(x) + 1)$$

- Mathematica yields $-2 \operatorname{arctanh}(1 + 2e^x)$, which is $x - \log(\exp(x) + 1) + i\pi$

Exponential polynomial part

$$f(x) = \sum_{i=-l}^{k} p_i \theta^i = v_0(\theta)' + \sum_i c_i \frac{v_i(\theta)'}{v_i(\theta)}$$

- $v_0(\theta)$ can only have a constant in θ or θ^j as a denominator
- $v_i(\theta)$ is either in F_{n-1} or θ, which can be absorbed in $v_0(\theta)'$, thus:

$$f(x) = \sum_{j=-l}^{k} \left(q_j \theta^j\right)' + \sum_i c_i \frac{v_i'}{v_i} = \sum_{j=-l}^{k} \left(q_j' + j u' q_j\right) \theta^j + \sum_i c_i \frac{v_i'}{v_i}$$

System of equations:

$$p_j = q_j' + j u' q_j \text{ for } -k \le j \le 1, 1 \le j \le l$$

$$p_0 = q_0' + \frac{\sum_i c_i v_i'}{\sum_i c_i v_i}$$

Solving the system

- Recursively compute $\int p_0$: if not elementary, no solution
- $p_j = q_j' + ju'q_j$ is a differential equation in q_j where the solution must be in F_{n-1}
- Algorithms exist to solve this Risch differential equation
- If no solution, then $\int f$ is not elementary
- We now have an algorithm for input with arbitrary nestings of exp and log!

Integration of algebraic functions

- What about integrating functions like $\sqrt[3]{x^4 + 1}$ or $\frac{1}{\sqrt{x^3+x+1}}$?
- In general: $\frac{p(x,y)}{q(x,y)}$ with y algebraic with minimal polynomial $F(x, y) = 0 \in K[x][y]$
- We can write $\frac{p(x,y)}{q(x,y)}$ as $\frac{n(x,y)}{d(x)}$ using EEA
- To determine the logarithmic terms, we need to identify the poles
- Not always the zeros of the denominator!
  - If $F(x, y) = y^4 - x^3$, then $f = \frac{y^2}{x}$ seemingly has a pole at x
  - However: $f^2 = \frac{x^3}{x^2} = x$, so f has no pole
- Write f in an integral basis of pole-less functions
- This integral basis is hard to find in general

Example

$$f = \frac{2x^4 + 1}{(x^5 + x)\sqrt{x^4 + 1}} = \frac{2x^4 + 1}{(x^5 + x)y}, \quad F(x, y) = y^2 - x^4 - 1 = 0$$

- Integral basis: $\{1, y\}$

$$f = \frac{2x^4 + 1}{x^9 + 2x^5 + x} y$$

- Proceed as usual: square-free factor the denominator and Hermite reduce

$$f = -\frac{y}{2(x^4 + 1)} + \int \frac{y}{x^5 + x}$$

Integration of algebraic functions

- $f = \frac{a(x,y)}{d(x)}$ where d is square-free and the zeroes of d are poles of f
- Clear denominators from $a(x, y)$:

$$f = \frac{g(x, y)}{d(x)h(x)}$$

- Then, the residue is the roots of

$$R(z) = \operatorname{res}_x\left(\operatorname{prim}_z\left(\operatorname{res}_y(g(x, y) - zd'(x)h(x), F(x, y))\right), d(x)\right)$$

- For our example: $z = -1, z = 1$
- Obtaining a solution in terms of logarithms is complicated
- At this stage most computer algebra systems give up
- Case of mixed transcendental and algebraic towers was only solved in 1987

Extension of Risch decision process

- Error function: $\operatorname{erf}(h(x)) = \frac{2}{\sqrt{\pi}} \int_0^{h(x)} e^{-t^2}\, dt$
- Extend Liouville's theorem:

$$w + \sum_i c_i \log(v_i) + \sum d_j \operatorname{erf}(u_j)$$

where $d_j \in K, u_j \in F_{n-1}$

- Error function behaves similar to log extension
- Error terms come from integration of polynomial part exp extension $\frac{a}{(e^p)^k}$

Example

- $\int \operatorname{erf}(x)^2 = \int \theta^2 = b_3\theta^3 + b_2\theta^2 + b_1\theta + b_0$
- b_0 can contain erf
- Solve as before: $b_3 = c_3 \in K$
- $\theta^2 : 1 = c_3\frac{6}{\sqrt{\pi}}e^{-x^2} + b_2' \Rightarrow c_3 = 0, b_2 = x + c_2$
- $\theta^1 : 0 = \left(x + c_2\right)\frac{4}{\sqrt{\pi}}e^{-x^2} + b_1' \Rightarrow c_2 = 0, b_1 = \frac{2}{\sqrt{\pi}}e^{-x^2} + c_1$
- $\theta^0 : 0 = \left(\frac{2}{\sqrt{\pi}}e^{-x^2} + c_1\right)\frac{2}{\sqrt{\pi}}e^{-x^2} + b_0' \Rightarrow c_1 = 0, b_0 = -\sqrt{\frac{2}{\pi}}\operatorname{erf}\left(x\sqrt{2}\right) + c_0$

$$\int \operatorname{erf}(x)^2 = x\operatorname{erf}(x)^2 + \frac{2}{\sqrt{\pi}}e^{-x^2}\operatorname{erf}(x) - \sqrt{\frac{2}{\pi}}\operatorname{erf}\left(x\sqrt{2}\right)$$

Modification to exponential extension

- Recap: we had integrand $f(x) = \sum_{i=-l}^{k} p_i \theta^i$
- If $\theta = \exp(u) = d \exp\left(-w^2\right)$, $d \in K$ its integral is $\propto \operatorname{erf}(w)$
- Thus contribution of $\operatorname{erf}(w)' = d\operatorname{erf}\left(\sqrt{-u}\right)' = -\exp(u)\frac{u'}{\sqrt{\pi}\sqrt{-u}} = -\frac{u'}{\sqrt{\pi}\sqrt{-u}}\theta$
- We modify relation:

$$f(x) = \sum_{j=-l}^{k}\left(q_j' + ju'q_j + -d_j\frac{u'}{\sqrt{\pi}\sqrt{-u}}\right)\theta^j + \sum_i c_i\frac{v_i'}{v_i} + \sum_k d_k\frac{2}{\sqrt{\pi}e^{w_k^2}}w_k'$$

- Solve differential equations as usual

General expressions

General expressions

A tree representation of a mathematical expression:

```
enum Expression {
    Add(Vec<Expression>),
    Mul(Vec<Expression>),
    Fun(usize, Vec<Expression>)
    Var(usize),
    Pow(Box<(Expression, Expression)>),
    Num(i64, i64),
}
```

- Allows for easy modifications and pattern matching
- $\frac{2}{3} + x$ is `Expression::Add(vec![Expression::Num(2,3), Expression::Var(1)])`
- Not memory efficient: every `Expression` is 32 bytes
  - $\frac{2}{3} + x$ uses 96 bytes

Compact linear representation

- Represent an expression as a byte vector: `Vec<u8>`

Use tag-length-value format

```
const NUM_ID: u8 = 1;
const VAR_ID: u8 = 2;
const FUN_ID: u8 = 3;
const MUL_ID: u8 = 4;
const ADD_ID: u8 = 5;
const POW_ID: u8 = 6;
```

Compress two numbers: a `u8` tag sets the size of two numbers

```
const U8_NUM1 : u8 = 0b00000001;
const U32_NUM1: u8 = 0b00000011;
const U64_NUM1: u8 = 0b00000100;
const U8_NUM2 : u8 = 0b00010000;
const U32_NUM2: u8 = 0b00110000;
const U64_NUM2: u8 = 0b01000000;
```

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Function tag

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Size

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Two `u8` numbers (`U8_NUM1 | U8_NUM2`)

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Function name “f”

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Number of arguments

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Variable tag

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: 1 u8 number

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Variable name “x”

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Number tag

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Two `u8` numbers

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Numerator

Compact linear representation

- The first 256 variable names only take up 2 bytes
- Function name and number of arguments are packed together
- For example $f\left(x, \frac{2}{5}\right)$ is coded in only 15 bytes

$$[3, 10, 0, 0, 0, 17, 1, 2, 2, 1, 0, 1, 17, 2, 5]$$

Meaning: Denominator

Abstraction over memory representation

- We want our algorithms to be agnostic of the memory representation

```rust
pub enum Atom {
    Num(Num),
    Var(Var),
    Fun(Fun),
    Pow(Pow),
    Mul(Mul),
    Add(Add),
}
```

```rust
pub enum AtomView<'a> {
    Num(NumView<'a>),
    Var(VarView<'a>),
    Fun(FunView<'a>),
    Pow(PowView<'a>),
    Mul(MulView<'a>),
    Add(AddView<'a>),
}
```

Example: a function

```
pub struct Fun {
  data: Vec<u8>, // data[0] = 3
}

impl Fun {
    fn new(symbol: Symbol) → Fun { }
    fn add_arg(&mut self, arg: AtomView) {  }
}
```

Example: a function

```
pub struct FunView<'a> {
  data: &'a [u8], // data[0] = 3
}

impl<'a> IntoIterator for FunView<'a> {
  type Item = AtomView<'a>;
}

impl<'a> FunView<'a> {
    pub fn get_nargs(&self) → usize {
        self.data[1 + 4..].get_nums_u64().1 as usize
    }
}
```

Walking through the expression tree

Linear representation completely hidden:

```
fn tree_walk(a: AtomView) {
    match a {
        AtomView::Num(_) | AtomView::Var(_) ⇒ {},
        AtomView::Mul(m) ⇒ {
            for a in m { a.tree_walk(); }
        }
        AtomView::Fun(f) ⇒ {
            for a in f { a.tree_walk(); }
        }
        // ...
    }
}
```

- For the remainder, assume we have an `Expression` enum as before

Derivatives

```
impl Expression {
    fn derivative(&self, x: usize) → Expression {
        match self {
            Expression::Add(a) ⇒ {
                let mut res = vec![];
                for e in a {
                    res.push(e.derivative(x));
                }
                Expression::Add(res)
            }
            Expression::Var(a) if *a == x ⇒ Expression::Num(1, 1),
            Expression::Var(_) ⇒ Expression::Num(0, 1),
            Expression::Num(_) ⇒ Expression::Num(0, 1),
            ...
        }
    }
}
```

Normalization

- Simplify patterns such as x^0, $1 + 2$, $x \cdot x^3$
- What more should be done?
  - Factoring after every operation is too expensive
  - Expanding can create a lot of terms
- Take special care not to use relations that may not be valid in general:
  - $\log(\exp(x)) = x$ only for real x
  - $x^0 = 1$, but not true if $x = 0$ which is indeterminate
  - $1^x = 1$, but not if $x = \infty$, which is indeterminate
  - $\frac{x}{x} = 1$, but not if $x = 0$, which is indeterminate
  - Assume values are not special

Normalization

```
fn normalize(&self) → Expression {
    match self {
        Expression::Pow((base, exp)) ⇒ {
          let base_norm = base.normalize();
          let exp_norm = exp.normalize();
            if base_norm == Expression::Num(0, 1) {
                Expression::Num(1, 1)
            } else if exp_norm == Expression::Num(1, 1) {
                base_norm
            } else {
                Expression::Pow(Box::new((base_norm, exp_norm)))
            }
        }
    }
}
```

Normalize sums

- We want to merge $x + y + 2x$ to $3x + y$
- Sort terms and merge neighbors that are equal up to a constant

```
fn normalize_sum(args: &mut [Expression]) → Expression {
    let mut args = args.iter().map(|x| x.normalize()).collect();
    args.sort_by(|a, b| a.cmp_ignore_constant(b)); // sort 2*x and x next to each other
    let mut output = vec![];
    for a in args {
        if let Some(last) = output.last_mut() {
            if last.cmp_ignore_constant(a).is_eq() {
                last.add_constant(a.get_constant());
                continue;
            }
        }
        output.push(a);
    }
    if output.len() == 1 { output.pop().unwrap() } else { Expression::Add(output) }
}
```

Canceling

- Simplification of $\frac{x^2+2x+1}{x+1}$ is not automatic
- Convert to and from a rational polynomial data structure

```
fn cancel(&self) → Expression {
    let rat_poly = self.to_rational_polynomial();
    rat_poly.to_expression()
}
```

- The rational polynomial uses the GCD algorithms we constructed earlier
- Similarly: factorization involves converting to polynomial form

Pattern matching

- A pattern consists of literal parts and wildcards
- A wildcard is a symbol ending with an underscore and can match any `Expression`
  - $x\_$ matches $3, x, \sin(x), x^2 + y$ etc.
- $f(y\_)$ matches $f(x), f(3), f(\sin(x))$ etc.
-

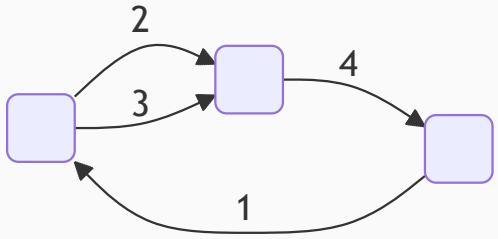
-

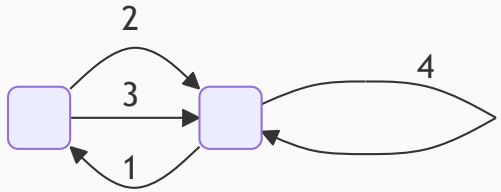
Pattern matching

- A pattern consists of literal parts and wildcards
- A wildcard is a symbol ending with an underscore and can match any `Expression`
  - $x\_$ matches $3, x, \sin(x), x^2 + y$ etc.
- $f(y\_)$ matches $f(x), f(3), f(\sin(x))$ etc.
- Ranged wildcards are symbols ending with `___` and can match an argument subset in `Fun(usize, Vec<Expression>)`, `Add(Vec<Expression>)` or `Mul(Vec<Expression>)`
  - $g(z\_\_\_)$ matches $g(), g(x), g(x, y, z)$ etc.
  - $x * z\_\_\_$ matches $x * y * z, 3x, x * \sin(x) * y$ etc., but does not match x (not a product)
- Wildcards must take symmetry into account
  - $x\_ \cdot z + x\_ \cdot y$ matches $xy + xz$ but does not match term-by-term in this order

Is graph connected?

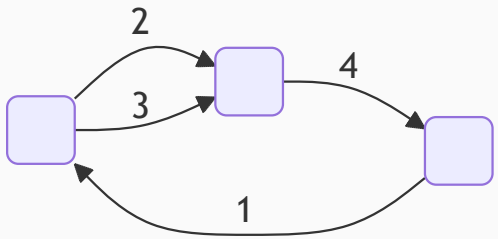
- Pattern matching can be used to code algorithms
  - Fuse vertices to see if graph is connected

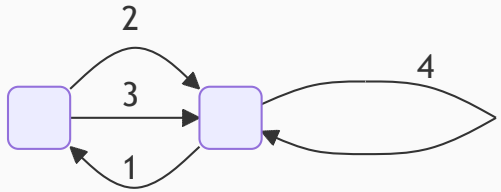




Is graph connected?

- Pattern matching can be used to code algorithms
  - Fuse vertices to see if graph is connected






```
e = E('v(1,2,3)*v(2,3,4)*v(4,1)')
pat = E('v(l1___,x_,r1___)*v(l2___,x_,r2___)')
rhs = E('v(l1___,r1___,l2___,r2___)')

e = e.replace(pat, rhs, repeat=True)
```

yields `v(1,1,2,2)`, with edges that form a fundamental cycle basis!

Pattern matching without symmetries

```
fn match_pattern(&self, pattern: &Expression, wildcard_map: &mut [Option<&Expression>]) ->  bool {
  if pattern.is_wildcard() { // x_
    if let Some(matched) = wildcard_map[pattern.get_wildcard_id()] {
        return self == *matched;
    }
    wildcard_map[pattern.get_wildcard_id()] = Some(self);
    return true;
  }
  match (self, pattern) {
    (Expression::Fun(f, args), Expression::Fun(pf, pargs)) if f == pf && args.len() == pargs.len() => {
        for (a, p) in args.iter().zip(pargs.iter()) {
            if !a.match_pattern(p, wildcard_map) {
                return false;
            }
        }
        return true;
    }
    ...
  }
}
```

- Pattern $f(1, x\_, 2, x\_, 3)$ matches $f(1, 5, 2, 5, 3)$ with $x\_ \to 5$

Pattern matching with symmetries

- Match $f(xy)f(y)$ with pattern $f(y\_)f(y\_ \cdot z\_)$
- Try all combinations of factors
  - $\{\{f(y\_) \leftrightarrow f(xy), f(y\_ \cdot z\_) \leftrightarrow f(y)\}, \{f(y\_) \leftrightarrow f(y), f(y\_ \cdot z\_) \leftrightarrow f(xy)\}\}$
- Try all wildcard associations in $f(y\_ \cdot z\_)$ with those of $f(y\_)$
  - $f(y\_ \cdot z\_) \leftrightarrow f(xy)$ should yield $\{\{y\_ \to x, z\_ \to y\}, \{y\_ \to y, z\_ \to x\}\}$

Pattern matching with symmetries

- Match $f(xy)f(y)$ with pattern $f(y\_)f(y\_ \cdot z\_)$
- Try all combinations of factors
  - $\{\{f(y\_) \leftrightarrow f(xy), f(y\_ \cdot z\_) \leftrightarrow f(y)\}, \{f(y\_) \leftrightarrow f(y), f(y\_ \cdot z\_) \leftrightarrow f(xy)\}\}$
- Try all wildcard associations in $f(y\_ \cdot z\_)$ with those of $f(y\_)$
  - $f(y\_ \cdot z\_) \leftrightarrow f(xy)$ should yield $\{\{y\_ \to x, z\_ \to y\}, \{y\_ \to y, z\_ \to x\}\}$

```
fn match_pattern(&self, pattern: &Expression, wildcard_map: &[Option<&Expression>]) →
Vec<Vec<Option<&Expression>>> {
  if pattern.is_wildcard() { // x_
      if let Some(matched) = wildcard_map[pattern.get_wildcard_id()] {
        return if self == *matched { vec![wildcard_map.to_vec()] } else { vec![] };
      }
      let mut wildcard_map = wildcard_map.to_vec();
      wildcard_map[pattern.get_wildcard_id()] = Some(self);
      return vec![wildcard_map];
  }

  match (self, pattern) {
      ... // on next slide
  }
}
```

Pattern matching with symmetries

- Generate cartesian product of all associations

```
match (self, pattern) {
  (Expression::Mul(args), Expression::Mul(pargs)) if args.len() == pargs.len() ⇒ {
      let mut wildcard_associations = vec![];
      for (i, a) in args.iter().enumerate() { // try all associations for first pattern
        let new_wildcard_maps = a.match_pattern(&pargs[0], wildcard_map);

        let mut stripped_mul = args.clone();
        stripped_mul.remove(i);
        let stripped_mul = Expression::Mul(stripped_mul);
        let mut stripped_pattern = Expression::Mul(pargs[1..].to_vec());

        for new_wildcard_map in new_wildcard_maps {
            let solutions = stripped_mul.match_pattern(&stripped_pattern, &new_wildcard_map);
            wildcard_associations.extend(solutions);
        }
      }
      return wildcard_associations;
  }
    ...
}
```

Expression evaluation

Expression evaluation

Build fast evaluator for e :

$$e = x + \pi + \cos(x) + f(g(x+1), 2x)$$
$$f(y, z) = y^2 + z^2 y^2$$
$$g(y) = 5y$$

- First approach: walk the expression tree, and convert types as needed

Expression evaluation

```
fn evaluate(&self, vars: &HashMap<usize, f64>, funcs: &HashMap<usize, Fn(&[f64]) → f64>) → f64 {
    match self {
        Expression::Add(a) ⇒ a.iter().map(|x| x.evaluate(vars, funcs)).sum(),
        Expression::Mul(a) ⇒ a.iter().map(|x| x.evaluate(vars, funcs)).product(),
        Expression::Fun(id, args) ⇒ {
            let f = funcs.get(id).expect("Function not found");
            let arg_vals: Vec<f64> = args.iter().map(|x| x.evaluate(vars, funcs)).collect();
            f(&arg_vals)
        }
        Expression::Var(id) ⇒ *vars.get(id).expect("Variable not found"),
        Expression::Pow(p) ⇒ {
            let (base, exp) = p;
            base.evaluate(vars, funcs).powf(exp.evaluate(vars, funcs))
        }
        Expression::Num(n, d) ⇒ *n as f64 / *d as f64,
    }
}
```

- No support for removing common subexpresions
- Function calls needed

Linearization

- Linearize the expression by storing subexpressions in temporary variables

```
e = x + pi + cos(x) + f(g(x+1), x*2)
f(y,z) = y^2 + z^2*y^2
g(y) = y * 5
```

```
Z[0] = x + 1
Z[1] = g(Z[0])
Z[2] = x * 2
Z[3] = f(Z[1], Z[2])
Z[4] = cos(x)
e = x + 22/7 + Z[3] + Z[4]
```

Linearization

- Substitute functions

```
e = x + pi + cos(x) + f(g(x+1), x*2)
f(y,z) = y^2 + z^2*y^2
g(y) = y * 5
```

```
Z[0] = x + 1
Z[1] = Z[0] * 5
Z[2] = x * 2
Z[3] = Z[1]^2 + Z[2]^2*Z[1]^2
Z[4] = cos(x)
e = x + 22/7 + Z[3] + Z[4]
```

Linearization

- Move all numbers and parameters into temporary variables
- One instruction type per line

```
Z[0] = x + 1
Z[1] = Z[0] * 5
Z[2] = x * 2
Z[3] = Z[1]^2 + Z[2]^2*Z[1]^2
Z[4] = cos(x)
e = x + 22/7 + Z[3] + Z[4]
```

```
Z[0] = x;
Z[1] = 1;
Z[2] = 2;
Z[3] = 5;
Z[4] = 22/7;
Z[5] = Z[0] + Z[1]
Z[6] = Z[3] * Z[5]
Z[7] = Z[0] * Z[2]
Z[8] = Z[6] * Z[6]
Z[9] = Z[6] * Z[6] * Z[7] * Z[7]
Z[10] = Z[8] + Z[9]
Z[11] = cos(Z[0])
Z[12] = Z[0] + Z[4] + Z[10] + Z[11]
```

Fast representation

```
enum Instruction {
  Add(usize, Vec<usize>), // (out index, input indices)
  Mul(usize, Vec<usize>),
  BuiltinFunction(usize, Symbol, usize)
}
```

- Fast evaluation of instructions:

```
for i in instructions {
  match i {
      Instr::Add(r, v) ⇒ {
          let tmp = Z[v[0]];
          for x in &v[1..] {
              tmp += Z[*x];
          }
          Z[*r] = tmp;
      }
      ...
```

Horner's method

- Writing $y^2 + z^2y^2$ as $y^2(1 + z^2)$ saves two multiplications
- Called Horner's method
- $x^3y^2 + x^2y + x^3z \Rightarrow 9 \times$ and $2 +$
- Scheme order $[x, y, z]$ gives $x^2(y + x(y^2 + z)) \Rightarrow 4 \times$ and $2 +$
- Scheme order $[y, x, z]$ gives $x^3z + y(x^2(1 + xy)) \Rightarrow 7 \times$ and $2 +$
- Finding the optimal scheme is NP-hard
- Choice of Horner scheme also affects the number of common subexpressions

Finding a near-optimal Horner scheme

- Monte Carlo Tree Search is succesful in playing games such as Go
  - Build a search tree of variable orderings
  - Sample along the best path
  - What is the best?
-

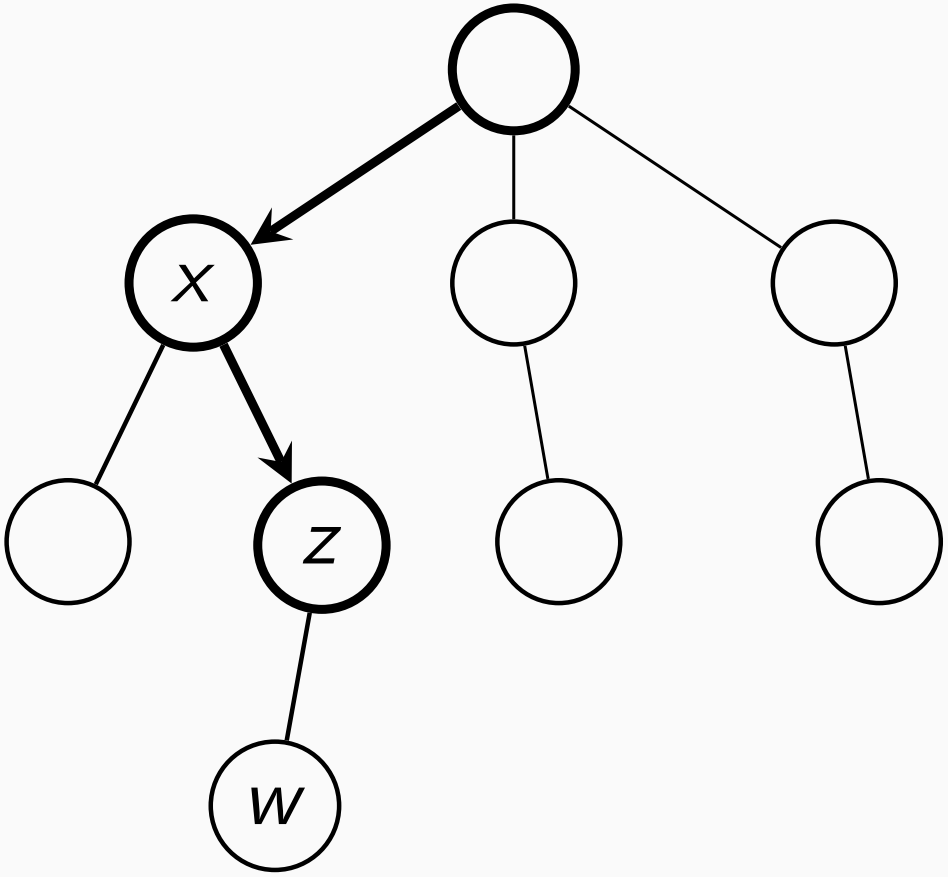
Finding a near-optimal Horner scheme

- Monte Carlo Tree Search is succesful in playing games such as Go
  - Build a search tree of variable orderings
  - Sample along the best path
  - What is the best?
- Solve a multi-armed bandit problem: given K slot machines with unknown reward distributions, maximize total reward
  - Sample according to:

$$\operatorname{argmax}_i \frac{x_i}{n_i} + 2C\sqrt{2\frac{\ln \sum_j n_j}{n_i}}$$

- x_i: total reward of machine i
- n_i: number of times machine i was played
- C: tunable parameter to adjust exploration vs exploitation
- Idea: apply above sampling strategy at every node of the search tree!

Selection



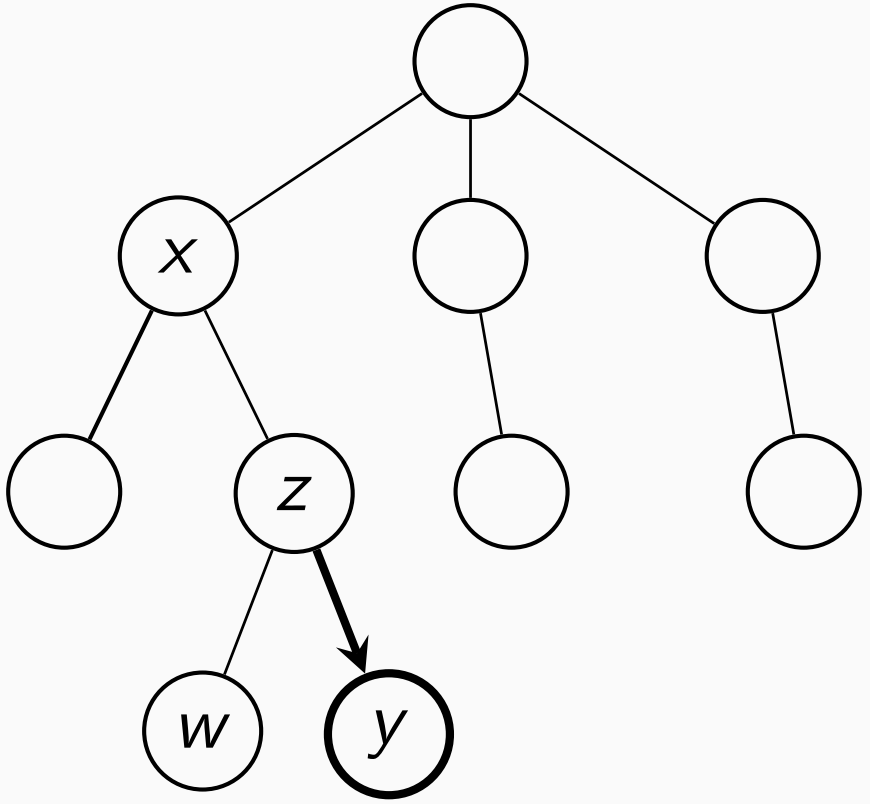
Node selection criterion:

$$\text{argmax}_{\text{children } c \text{ of } s} \frac{x(c)}{n(c)} + 2C\sqrt{\frac{2 \ln n(s)}{n(c)}}$$

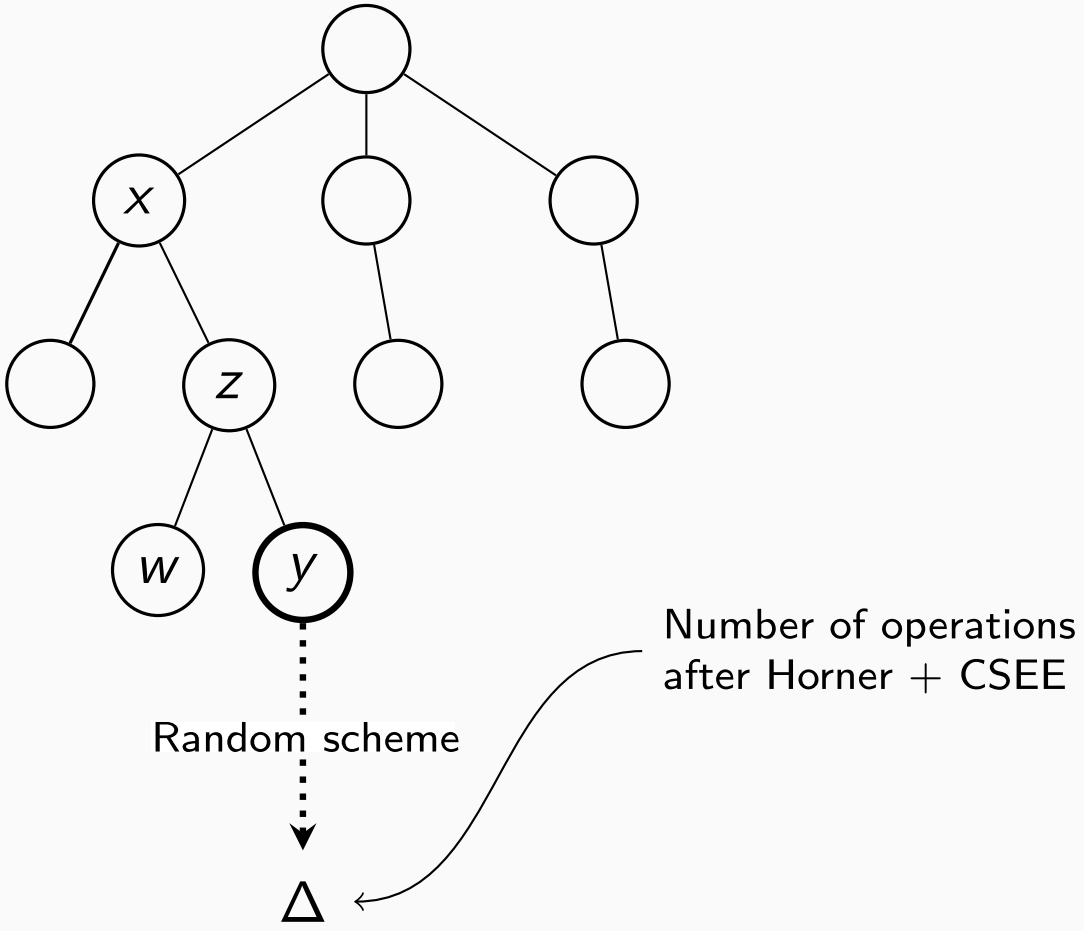
- $x(c)$: score of node c
- $n(c)$: visits of node c
- C: exploration vs exploitation parameter

Expansion

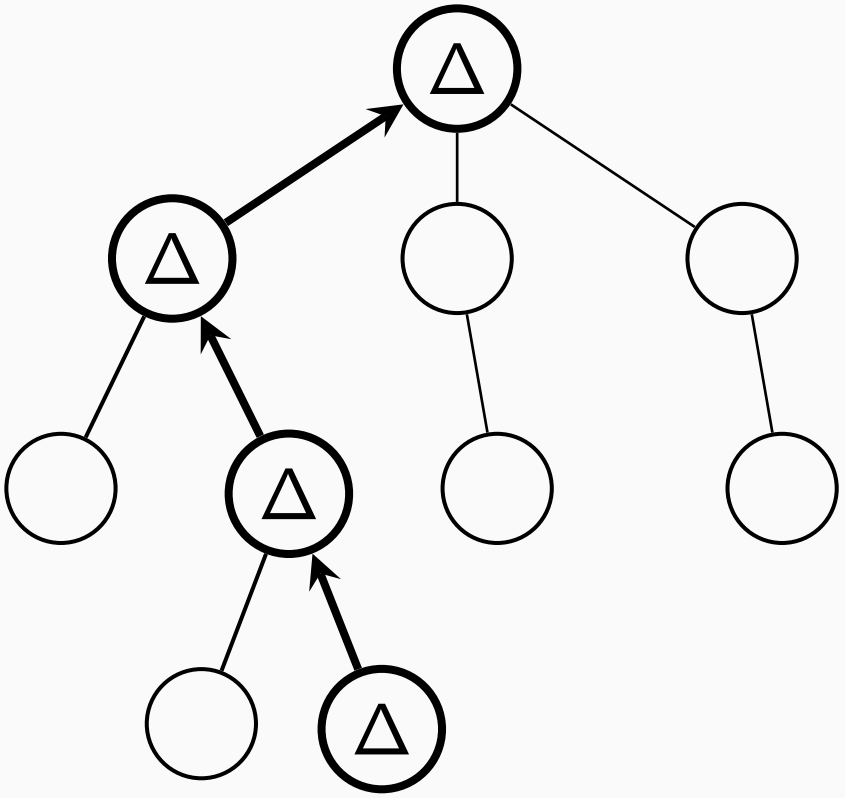
- Add new selected nodes to the tree



Simulation


x
z
w
y
Random scheme
Number of operations
after Horner + CSEE
Δ

Backpropagation



MCTS loop:

- Keep sampling and expanding the tree in a best-first way

Downsides:

- No guarantee of finding the optimal scheme
- Tuning C_p is hard

Common subexpression elimination

- Evaluate at the same time: $e_1 = f(x + 1), e_2 = f(x) + 1, f(y) = y$

```
Z[0] = x              Z[0] = x              Z[0] = x              Z[0] = x
Z[1] = 1              Z[1] = 1              Z[1] = 1              Z[1] = 1
Z[2] = Z[0] + Z[1]    Z[2] = Z[0] + Z[1]    Z[2] = Z[0] + Z[1]    Z[2] = Z[0] + Z[1]
Z[3] = f(Z[2])        Z[3] = Z[2]           Z[3] = Z[2]           Z[3] = Z[2]

Z[4] = f(Z[0])        Z[4] = Z[0]
Z[5] = Z[4] + 1       Z[5] = Z[4] + Z[1]    Z[5] = Z[0] + Z[1]    Z[5] = Z[2]
```

- Common pair `Z[0] + Z[1]` only computed once
- Iteratively find repeated pairs and extract them until no more common pairs are found

Register recycling

- Recycle register `Z[i]` by keeping track by adding `i` to the free pool after the last use of `Z[i]` on the right-hand side

```
Z[5] = Z[0] + Z[1]
Z[6] = Z[3] * Z[5]          // last 5
Z[7] = Z[0] * Z[2]
Z[8] = Z[6] * Z[6]          // last 6
Z[9] = Z[8] * Z[7] * Z[7]   // last 7
Z[10] = Z[8] + Z[9]         // last 8,9
Z[11] = cos(Z[0])
Z[12] = Z[0] + Z[4] + Z[10] + Z[11]
```

```
Z[5] = Z[0] + Z[1]
Z[5] = Z[3] * Z[5]
Z[6] = Z[0] * Z[2]
Z[5] = Z[5] * Z[5]
Z[6] = Z[5] * Z[6] * Z[6]
Z[5] = Z[5] + Z[6]
Z[6] = cos(Z[0])
Z[6] = Z[0] + Z[4] + Z[5] + Z[6]
```

Compilation

- From the list of `Instruction` we can easily generate C++ code

```
for i in instructions {
  match i {
      Instr::Add(r, v) ⇒ {
          print!("Z[{}] = Z[{}]", r, v[0]);
          for x in &v[1..] {
              print!(" + Z[{}]", x);
          }
          println!(";");
      }
      ...
```

- Compiled code is much faster to execute
- Downside: compilation time can be significant

Assembly output

- We can also generate assembly code instead of C++
- Assume memory address of `Z` is stored in register `%rdi`
- Modern x64 hardware has 16 floating point registers `%xmm0` to `%xmm15`
- `Z[8] = Z[3] + Z[5]` becomes

```
movsd   24(%rdi), %xmm0 # load Z[24/8]=Z[3] into xmm0
addsd   40(%rdi), %xmm0 # add  Z[40/8]=Z[5] to xmm0
movsd   %xmm0, 64(%rdi) # store xmm0 into Z[64/8]=Z[8]
```

- Writing assembly directly evades C++ compilation time completely

Register recycling

- We encounter code like:

```
Z[8] = Z[3] + Z[5];
Z[8] = Z[8] * Z[4];
```

- `Z[8]` is used and immediately overwritten
- Writing back and forth into memory is slow
- Keep results in a spare floating point registers

```
movsd   24(%rdi), %xmm0 # load Z[3] into xmm0
addsd   40(%rdi), %xmm0 # add Z[5] to xmm0
mulsd   32(%rdi), %xmm0 # multiply xmm0 by Z[4]
movsd   %xmm0, 64(%rdi) # store xmm0 into Z[8]
```

- On modern hardware `a+=b*c` can be done with a fused multiply-add instruction

Conclusion

Conclusion

We saw how:

- abstractions over rings and fields help build generic algorithms
- systems of polynomial (in)equalities are solved through Gröbner bases and Cylindrical Algebraic Decomposition
- fast greatest common divisor computations are achieved through finite field reconstruction
- polynomial factorization is done in finite fields and how it can be extended to the integers using Hensel lifting
- symbolic integration is performed through the Risch algorithm
- expressions are represented, manipulated and evaluated efficiently

Thank you for your attention