Computational Algebra: Classical Foundations and Quantum Algorithms

Your Name

Preface

This set of lecture notes is designed for a course on Quantum Computational Algebra, which evolves traditional computational algebra toward quantum-enhanced methods. The course bridges abstract algebraic structures with efficient classical algorithms, gradually introducing quantum computing to explore speedups in problems like factoring, solving polynomial systems, and lattice reductions.

Targeted at upper-level undergraduates or graduate students, it assumes prerequisites in abstract algebra, linear algebra, and basic programming. We recommend familiarity with a computer algebra system (CAS) like SageMath and quantum simulation tools like Qiskit. Throughout the course, we will primarily use Python with the SymPy library for symbolic computations, alongside SageMath as the main CAS for advanced algebraic manipulations and integrations with quantum tools.

The progression starts with classical foundations, drawing from Ideals, Varieties, and Algorithms" by Cox et al. [4] and Modern Computer Algebra" by von zur Gathen and Gerhard [5]. Midway, we shift to quantum concepts, supplemented by "Quantum Computation and Quantum Information" by Nielsen and Chuang [10].

Each chapter includes sections with key concepts, algorithms, and connections between classical and quantum approaches. Exercises involve proofs, implementations, and simulations. The course spans 12-15 weeks, with assessments including homework, exams, and a final project.

These notes provide a skeleton for lectures, to be expanded with examples, proofs, and code snippets during teaching.

Contents

CHAPTER 1

Introduction to Computational Algebra

In the modern landscape of mathematics, data science, cryptography, and quantum information, algebraic methods play a foundational role. Yet many of the structures we study—polynomial rings, number fields, ideals, and varieties—are too complex to handle without computational assistance. **Computational algebra** bridges the abstract world of algebra with the practical power of algorithmic computation. These notes are designed to introduce both the theory and practice of symbolic computation, with an eye toward applications in quantum computing and emerging technologies.

1. Overview of Symbolic Computation

Computational algebra, also known as *symbolic computation*, is a field at the intersection of mathematics and computer science. Its goal is to design and analyze algorithms that manipulate algebraic expressions and mathematical objects in an exact, symbolic form. Unlike numerical computation, which approximates quantities using floating-point arithmetic, symbolic methods maintain the exact structure of expressions.

For example, solving the equation $x^2 - 2 = 0$ numerically yields an approximation $x \approx 1.414$, while symbolic computation produces the exact answer $x = \sqrt{2}$. Symbolic approaches are crucial in applications where precision is essential, such as theorem proving, cryptographic protocol design, and the study of algebraic geometry.

Historically, symbolic manipulation dates back to early developments in algebra by mathematicians like Al-Khwarizmi and Newton. The field gained momentum with the advent of modern computing. Early systems like MACSYMA, Reduce, and Scratchpad paved the way for today's powerful Computer Algebra Systems (CAS), including Mathematica, Maple, and the open-source SageMath [3].

Symbolic computation is deeply intertwined with the theory of polynomial ideals, algebraic varieties, and exact algorithms for manipulating algebraic numbers and functions. Its applications extend to areas such as:

- **Algebraic geometry**: computing intersections of varieties or solving systems of polynomial equations.
- **Number theory**: computing GCDs of polynomials, modular arithmetic, or algebraic number fields.
- **Coding theory and cryptography**: using polynomial rings and finite fields in the design and breaking of codes.
- **Quantum computing**: where algebraic representations of states and circuits are essential.

This course builds a bridge between classical symbolic computation and its quantum analogs. Quantum algorithms such as Shor's factoring algorithm and Grover's search illustrate how quantum computers can solve symbolic problems exponentially faster than their classical counterparts. These notes will introduce

key algorithms, theoretical tools, and software systems used in both classical and quantum computational algebra.

2. Computer Algebra Systems (CAS) Tools

Computer Algebra Systems (CAS) are essential tools in modern computational algebra. They automate symbolic manipulations such as polynomial arithmetic, matrix operations, ideal computations, and algebraic geometry. In this course, we focus on two primary systems:

- **SageMath** — an open-source CAS built on Python, integrating libraries for algebra, number theory, and geometry.
- **SymPy** — a pure Python library for symbolic computation, useful for lightweight tasks and scripting.

We also introduce **Qiskit** for quantum algorithm simulations, especially for modules on quantum linear algebra and factoring.

2.1. Polynomial Rings in SageMath. SageMath offers robust support for constructing polynomial rings over various base rings or fields.

2.1.1. *Univariate Polynomial Rings.* To define a polynomial ring in one variable over the rationals:

```
R.<x> = PolynomialRing(QQ)
f = x^2 + 3*x + 2
f.factor()   # Output: (x + 1)*(x + 2)
```

You can perform standard operations:

- `f + x^2` to add polynomials,
- `f * (x - 1)` to multiply,
- `f.derivative()` for derivatives,
- `f.roots()` to find roots (in an algebraic closure).

2.1.2. *Multivariate Polynomial Rings.* To define a polynomial ring in two variables:

```
R.<x, y> = PolynomialRing(QQ, 2)
f = x^2 + y^2 + x*y
g = x^3 - y
h = f * g
h.expand()
```

You can also specify a monomial ordering:

```
R.<x, y> = PolynomialRing(QQ, order='lex')   # Lexicographic order
```

Sage handles:

- Arithmetic operations across multivariate polynomials.
- Substitutions: `f.subs({x:1, y:2})`
- Evaluation over finite fields or number fields.

2.2. Matrix Algebra in SageMath. SageMath provides comprehensive linear algebra functionality.

```
A = matrix(QQ, [[1, 2], [3, 4]])
A.transpose()
A.det()
A.inverse()
A.eigenvalues()
```

Matrices can be defined over various rings, such as $\mathbb{F}_p$:

```
F = GF(7)
B = matrix(F, [[2, 3], [4, 5]])
B^10
```

2.3. Basic SymPy Usage in Python. SymPy is ideal for lightweight symbolic tasks or projects not requiring the full Sage environment.

```
from sympy import symbols, expand, factor, simplify, solve

x, y = symbols('x y')
expr = (x + y)**2
expanded = expand(expr)
factored = factor(expanded)
simplified = simplify((x**2 + 2*x + 1)/(x + 1))

roots = solve(x**2 - 4, x)
```

SymPy can also interface with LaTeX and render output:

```
from sympy import latex
print(latex(expr))
```

2.4. Quantum Circuit Simulation with Qiskit. To prepare for quantum modules, Qiskit provides a Python interface for building and simulating quantum circuits.

```
from qiskit import QuantumCircuit
qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)
qc.measure_all()
qc.draw('mpl')
```

This allows classical simulation of quantum gates and measurements and supports execution on real quantum hardware via IBM Q Experience.

2.5. Installation and Setup.

- **SageMath**: Download from https://www.sagemath.org or use cloud-based versions at https://cocalc.com.
- **SymPy**: Install via `pip install sympy`.
- **Qiskit**: Install with `pip install qiskit`.

We encourage students to experiment interactively with Sage notebooks or Python scripts as they read these notes.

3. Complexity Analysis Basics

An essential aspect of algorithm design is analyzing its **computational complexity**. This tells us how the running time (or space usage) of an algorithm grows with the input size, typically denoted as a function of parameters like n (e.g., the degree of a polynomial or the bit-length of an integer). In computational algebra, complexity analysis is crucial because algorithms often operate on objects with multi-dimensional sizes, such as polynomials where both the degree d and the maximum coefficient bit-length b contribute to the overall cost.

For example, consider the multiplication of two univariate polynomials $f(x), g(x) \in \mathbb{Z}[x]$ of degree at most d with coefficients bounded by 2^b in absolute value. The naive algorithm requires $O(d^2)$ arithmetic operations, but each operation (multiplication or addition of integers) costs $O(b^2)$ time using schoolbook methods, leading to a total time of $O(d^2b^2)$. This highlights the need for precise bounds that account for all input parameters.

3.1. Asymptotic Notation.

To express complexity rigorously, we use asymptotic notations that describe the behavior as the input size approaches infinity. These notations focus on the dominant terms and ignore constant factors, providing a high-level view of scalability.

Formally, let $f, g : \mathbb{N} \to \mathbb{R}^+$ be functions. We say:

- **Big-O Notation ($O(f(n))$):** $g(n) = O(f(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$. This bounds the growth from above.

  Example

  Adding two degree-n polynomials: The running time $T(n)$ satisfies $T(n) = O(n)$, as it involves at most $n+1$ coefficient additions, each constant-time assuming fixed-size coefficients.

- **Big-Omega Notation ($\Omega(f(n))$):** $g(n) = \Omega(f(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $g(n) \geq c \cdot f(n)$ for all $n \geq n_0$. This bounds the growth from below.

  Example

  Naive polynomial multiplication: $T(n) = \Omega(n^2)$, since computing the product requires calculating at least $(n+1)^2/4$ non-trivial coefficient products in the worst case.

- **Big-Theta Notation ($\Theta(f(n))$):** $g(n) = \Theta(f(n))$ if $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$. This provides a tight asymptotic bound.

  Example

  Mergesort for sorting n elements: $T(n) = \Theta(n \log n)$, as the algorithm divides the array recursively ($\log n$ levels) and merges at each level in $O(n)$ time.

To illustrate, consider proving that the sum $S(n) = 1+2+\cdots+n = n(n+1)/2$ is $\Theta(n^2)$. Clearly, $S(n) \leq n^2$ for $n \geq 1$ (so $O(n^2)$ with $c = 1$), and $S(n) \geq n^2/2$ for $n \geq 1$ (so $\Omega(n^2)$ with $c = 1/2$), hence $\Theta(n^2)$.

In computational algebra, input sizes are often composite: for multivariate polynomials in v variables of total degree d, the number of monomials is $\binom{d+v}{v} = \Theta(d^v/v!)$, leading to complexities exponential in v.

3.2. Classical Complexity in Algebraic Algorithms. Many algebraic problems exhibit a range of complexities, from polynomial to exponential. Below, we detail key examples with mathematical derivations where appropriate.

Algorithm	Time Complexity	Space Complexity
Polynomial Addition (degree n)	$O(n)$	$O(n)$
Naive Multiplication (degree n)	$O(n^2)$	$O(n)$
Karatsuba Multiplication	$O(n^{\log_2 3}) \approx O(n^{1.585})$	$O(n)$
FFT-based Multiplication	$O(n \log n \log\log n)$	$O(n)$
Trial Division Factoring ($N = 2^b$)	$O(2^{b/2})$	$O(b)$
Euclidean GCD (integers, bit-length b)	$O(b^2)$ worst-case	$O(b)$
Extended Euclidean (for inverses)	$O(b^2)$	$O(b)$
Gröbner Basis (worst-case, v variables)	$O(d^{2^v})$ for degree d	Exponential

TABLE 1. Complexities of Common Algebraic Operations (assuming unit-cost arithmetic unless noted)

For the Euclidean algorithm on integers $a > b > 0$ with bit-length $b = \log_2 \max(a, b)$, the number of steps is at most $O(b)$ (by Fibonacci worst-case), and each step involves division costing $O(b^2)$ with schoolbook methods, yielding $O(b^3)$ total; however, using fast division, it's $O(b^2)$. Over fields like $\mathbb{Q}$, we must account for rational coefficients: each GCD step can increase numerator/denominator bits by 1, so after k steps, bits are $O(k)$, leading to total time $O(d^3)$ for degree-d polynomials where $k \leq d$.

Symbolic methods are prone to *coefficient explosion*, where intermediate results have coefficients with bit-lengths growing exponentially. For instance, in the subresultant Euclidean algorithm for polynomials $f, g \in \mathbb{Z}[x]$ of degree d and coefficient bound 2^b, the bit-length of coefficients in remainders can grow to $O(d(b + \log d))$, making naive time $O(d^3(b + \log d)^2)$. Modular techniques mitigate this: compute GCD modulo several primes p_i (each $O(d^2 \log p_i)$), then reconstruct using the Chinese Remainder Theorem in $O(d(b + \log d)^2)$, reducing overall to softly linear in input size [5].

3.3. Classical Complexity Classes. Understanding where algebraic problems fit in the complexity hierarchy helps identify hard problems amenable to quantum speedups. These classes are defined relative to Turing machines.

- **P (Polynomial Time)**: Solvable deterministically in $O(n^k)$ time for constant k. Examples: GCD via Euclidean algorithm ($k = 2$ for bit operations), basic polynomial arithmetic like interpolation using Lagrange formula in $O(n^2)$.
- **NP (Nondeterministic Polynomial Time)**: Verifiable in polynomial time given a certificate. Includes integer factorization (FACT: given factors, multiply to verify in $O(b^2)$), Hilbert's Nullstellensatz (deciding if a polynomial system has a complex solution, NP-complete).
- **EXP (Exponential Time)**: Solvable in $O(2^{n^c})$ time for constant c. Examples: General ideal membership testing via Gröbner bases, which in worst case requires enumerating $2^{O(v)}$ monomials for v variables.

Problems like FACT are in NP ∩ coNP but believed outside P, underpinning public-key cryptography (e.g., RSA security assumes FACT hardness). The polynomial hierarchy extends this, with Σ_2^P containing quantifier-alternating problems like minimizing quadratic forms over integers.

3.4. Quantum Complexity and Speedups. Quantum computing redefines tractable problems through parallelism via superposition and entanglement, modeled by quantum circuits.

- **BQP (Bounded-error Quantum Polynomial Time)**: Problems solvable on a quantum computer in $O(\texttt{Poly}(n))$ time with error $< 1/3$. For FACT of $N = 2^b$, Shor's algorithm uses period-finding: find order r of $a \bmod N$ where r divides $\phi(N)$, via quantum phase estimation on unitary $U|x\rangle = |a^x \mod N\rangle$, in $O(b^3)$ gates (QFT on $O(b)$-qubit register costs $O(b^2)$).
- **QMA (Quantum Merlin-Arthur)**: Quantum analog of NP, with quantum proofs verifiable in BQP. Relevant for ground state energy of algebraic Hamiltonians or variety isomorphism testing.

Key quantum primitives include:

- **Quantum Fourier Transform (QFT)**: Approximates the discrete Fourier transform $\hat{f}(k) = \frac{1}{\sqrt{M}} \sum_{j=0}^{M-1} f(j)\omega^{jk}$ where $\omega = e^{2\pi i/M}$, using $O((\log M)^2)$ gates (vs. classical FFT $O(M \log M)$), crucial for Shor's phase estimation $e^{2\pi i\phi} \approx \hat{r}/r$.
- **Grover's Search**: For unstructured search over N items, amplifies amplitude of marked states via $O(\sqrt{N})$ iterations of oracle and diffusion, providing quadratic speedup; e.g., find root of polynomial mod p in $O(\sqrt{p})$ vs. classical $O(p)$.

However, relativized separations (e.g., parity requires $\Omega(N)$ queries classically/quantumly in some oracles) show quantum advantages are problem-specific.

3.5. Practical Considerations. In practice, asymptotic bounds hide constants: Karatsuba ($O(n^{1.585})$) outperforms naive $O(n^2)$ only for $n > 100$ typically, due to recursion overhead. Tools like SageMath allow empirical profiling—e.g., time polynomial multiplication for increasing d to observe crossovers.

For bit-precise analysis, consider models like the multi-tape Turing machine, where integer multiplication costs $O(b \log b \log\log b)$ with FFT.

Quantum practically faces NISQ (Noisy Intermediate-Scale Quantum) limitations: Shor's requires thousands of logical qubits with error rates $< 10^{-10}$, so hybrid variational algorithms (e.g., VQE for algebraic optimization) combine classical gradient descent with quantum sampling.

Coefficient management: Probabilistic Monte Carlo methods, like randomized GCD over finite fields with success probability $1 - 1/p$, run in $O(\log^2 p)$ expected time, avoiding deterministic worst-cases.

This foundational knowledge equips you to evaluate algorithms throughout the course, blending theory with practice. For in-depth exploration, refer to [5] for classical analyses and [10] for quantum frameworks.

Exercises

1.1. Prove that matrix multiplication of $n \times n$ matrices over a ring is $O(n^3)$ naively. Hint: Count the number of scalar multiplications in the triple sum $\sum_{i,j,k} a_{ik}b_{kj}$.

1.2. Analyze the complexity of Horner's method for polynomial evaluation. Show it's $\Theta(n)$ by deriving upper and lower bounds on operations.

1.3. Implement trial division in Python for factoring small composites (up to 50 bits). Estimate the time for a 1024-bit semiprime using big-O bounds.

1.4. Prove the Euclidean algorithm for integers terminates in $O(\log b)$ steps for bit-length b. Use the Fibonacci sequence to exhibit the worst case.

1.5. Implement naive and Karatsuba multiplication in Python (using lists for dense polynomials). Time them for degrees 10 to 1000 and plot to verify asymptotic improvements.

CHAPTER 2

Polynomial Arithmetic and Data Structures

Polynomials lie at the heart of computational algebra. They serve not only as basic algebraic objects but also as the foundation for constructing more advanced structures such as rings, ideals, algebraic varieties, and modules. Their manipulation, through arithmetic and structural operations, is central to symbolic computation and has profound implications in algebraic geometry, number theory, coding theory, and even quantum computing. This chapter introduces the fundamental aspects of polynomial arithmetic and its algorithmic foundations. We focus on how polynomials are represented in software systems, the algorithms for performing arithmetic operations efficiently, and classical techniques for evaluation and interpolation. These foundational ideas form the backbone of more advanced topics, such as polynomial factorization, Gröbner bases, and resultants, which we will encounter in later chapters.

By the end of this chapter, students will be able to:

- Understand and implement different representations of polynomials.
- Perform basic arithmetic operations on polynomials and analyze their complexities.
- Apply evaluation and interpolation techniques, including previews of quantum enhancements.
- Develop Python code for polynomial manipulation using SymPy and compare classical methods.

1. Polynomial Representations

A polynomial over a ring R, such as $f(x) = \sum_{i=0}^{n} a_i x^i$ with coefficients $a_i \in R$, can be represented in several ways depending on the structure and sparsity of the polynomial. The choice of representation affects both the performance of algorithms and the amount of memory required.

In practice, two main types of representations are used: dense and sparse. In dense representations, every coefficient is stored, including zeros. This is efficient when most coefficients are nonzero. Sparse representations, on the other hand, store only the nonzero terms and are more efficient for polynomials with a high degree and few nonzero terms.

Dense representations typically use arrays indexed by the exponent. For example, the polynomial $f(x) = 3x^3 + 2x^2 + x + 4$ is represented as the list $[4, 1, 2, 3]$, with the ith entry corresponding to the coefficient of x^i. This allows constant-time access and fast arithmetic, but can be inefficient in memory when the polynomial has few nonzero terms and high degree.

In SageMath, a dense polynomial can be constructed and examined as follows:

SageMath Dense Polynomial Example

```
R.<x> = PolynomialRing(ZZ)
f = 3*x^3 + 2*x^2 + x + 4
print(f.coefficients(sparse=False)) # [4, 1, 2, 3]
```

Sparse representations store only the degrees and coefficients of nonzero terms. This is often implemented using dictionaries, where the keys are degrees and the values are coefficients. For instance, $f(x) = x^{1000} + 1$ can be stored as {0: 1, 1000: 1}. While this saves memory, accessing and modifying coefficients becomes slightly more expensive.

Here is a custom implementation of a sparse polynomial in Python:

Custom Sparse Polynomial in Python

```
from collections import defaultdict
class SparsePoly:
    def __init__(self):
        self.coeffs = defaultdict(int)

    def set_coeff(self, deg, coeff):
        if coeff != 0:
            self.coeffs[deg] = coeff
        elif deg in self.coeffs:
            del self.coeffs[deg]
    def __str__(self):
        if not self.coeffs:
            return "0"
        terms = sorted(self.coeffs.items(), reverse=True)
        return " + ".join(f"{c}x^{d}" if d > 0 else f"{c}"
                          for d, c in terms if c != 0).
                              replace("+ -", "- ")
p = SparsePoly()
p.set_coeff(0, 1)
p.set_coeff(1000, 1)
print(p) # x^1000 + 1
```

In multivariate cases, dense representations require multi-dimensional arrays, quickly becoming infeasible for high-degree polynomials in many variables. Sparse representations use tuples of exponents as dictionary keys, e.g., $(2, 1)$ representing x^2y.

For instance, in SageMath, multivariate polynomials can be handled sparsely:

SageMath Multivariate Sparse Polynomial

```
R.<x,y> = PolynomialRing(QQ, order='lex')
f = x^2*y + y^100
print(f.coefficients(sparse=True)) # [1, 1]
print(f.exponents()) # [(2, 1), (0, 100)]
```

Modern systems like SageMath and SymPy automatically select the appropriate representation based on the density of the polynomial. For advanced data

structures, linked lists or trees may be used for sparse multivariate polynomials to allow efficient insertions and deletions.

2. Basic Operations

Once a polynomial is represented, we can perform algebraic operations such as addition, multiplication, and division. The algorithms used differ depending on whether the representation is dense or sparse.

2.1. Addition and Subtraction. For dense polynomials, addition is implemented as element-wise addition of arrays and takes $O(n)$ time for polynomials of degree n. For sparse polynomials, addition involves merging sorted lists or combining dictionary keys, typically in $O(t_1 + t_2)$ time where t_i is the number of nonzero terms in the ith polynomial.

An example using SymPy:

SymPy Polynomial Addition

```
from sympy import Poly, symbols
x = symbols('x')
p1 = Poly(x**2 + 3*x + 2)
p2 = Poly(2*x**2 + x)
print(p1 + p2) # Poly(3*x**2 + 4*x + 2, x, domain='ZZ')
```

Subtraction follows similarly by negating coefficients.

2.2. Multiplication. The naive multiplication algorithm computes all pairwise products of terms and adds them, resulting in $O(n^2)$ complexity. More efficient methods include Karatsuba's algorithm and FFT-based multiplication.

Karatsuba's algorithm is a divide-and-conquer approach that reduces multiplication to three subproblems of half the size, achieving $O(n^{\log_2 3}) \approx O(n^{1.585})$ complexity. For two polynomials $f = f_1 x^{n/2} + f_0$ and $g = g_1 x^{n/2} + g_0$, the product is computed as:

$$fg = (f_1 g_1)x^n + [(f_0 + f_1)(g_0 + g_1) - f_0 g_0 - f_1 g_1]x^{n/2} + f_0 g_0.$$

FFT-based multiplication uses the Fast Fourier Transform to evaluate the polynomials at special points, multiply pointwise, and interpolate. This reduces the time complexity to $O(n \log n)$ but requires a field with enough roots of unity (e.g., complex numbers or finite fields with suitable structure).

In practice, systems like SageMath use FFT for large degrees over supported fields.

2.3. Division and Remainder. The division algorithm states that for polynomials $f, g \in R[x]$ with $g \neq 0$, there exist unique polynomials q, r such that $f = qg + r$ and $\deg(r) < \deg(g)$. The algorithm proceeds by subtracting appropriate multiples of g from f and is analogous to long division.

The complexity is $O(nm)$ for polynomials of degrees n and m, respectively. Over fields, exact division is possible if $r = 0$.

Handling coefficient rings requires care: over integers, one must manage content and primitive parts to avoid fraction fields unless necessary.

3. Evaluation and Interpolation

Evaluating a polynomial at a point is a fundamental operation. A naive method requires computing powers of the evaluation point explicitly, resulting in $O(n^2)$ complexity. Horner's rule improves this by rewriting the polynomial as a nested expression and evaluating it using only $O(n)$ additions and multiplications.

For example, $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0 = ((a_3x + a_2)x + a_1)x + a_0$.

Horner's Rule in Python

```
def horner(f_coeffs, a):
    result = 0
    for coeff in reversed(f_coeffs):
        result = result * a + coeff
    return result
print(horner([4, 1, 2, 3], 2)) # Evaluates f(2) for f(x) = 3
    x^3 + 2x^2 + x + 4 = 36
```

Polynomial interpolation is the inverse problem: given a set of points $\{(x_i, y_i)\}$, find a polynomial f such that $f(x_i) = y_i$ for all i. There are several classical methods for interpolation:

Lagrange interpolation constructs the interpolating polynomial as a sum of Lagrange basis polynomials:

$$f(x) = \sum_i y_i \prod_{j \neq i} \frac{x - x_j}{x_i - x_j}.$$

This method is conceptually simple but has $O(n^2)$ complexity.

Newton interpolation builds the polynomial incrementally using divided differences, with the same complexity but better adaptability to incremental updates. The divided difference table allows expressing $f(x) = a_0 + a_1(x - x_0) + \cdots + a_n(x - x_0)\cdots(x - x_{n-1})$.

For specialized points (e.g., roots of unity), FFT-based interpolation achieves $O(n \log n)$ time, previewing quantum Fourier transform applications where QFT enables even faster evaluations in superposition.

In SymPy, interpolation is straightforward:

SymPy Interpolation

```
from sympy import interpolating_poly
interpolating_poly(3, x, X=[0,1,2], Y=[1,4,9]) #
    Interpolates quadratic through (0,1),(1,4),(2,9)
```

Understanding these operations is critical for later topics in algebraic algorithms, especially GCD computation, factorization, and solving systems of polynomial equations. Quantum previews, like using QFT for fast evaluation, hint at speedups in Chapters 7 and 8.

4. Python Project

Develop a Python class using SymPy to represent and manipulate sparse polynomials. Implement efficient multiplication using the Karatsuba algorithm and test it against SymPy's built-in methods for large polynomials.

Extend the class to support multivariate polynomials and compare the performance of naive vs. Karatsuba multiplication on polynomials of degree up to 1024. Include timing benchmarks and a report on when each method is preferable. Optionally, integrate with Qiskit to simulate quantum-enhanced evaluation using a simple QFT circuit.

Exercises

2.1. Implement a dense polynomial class in Python with addition and multiplication. Compare memory usage to sparse for $x^{100} + 1$.

2.2. Use the master theorem to prove that Karatsuba's recurrence $T(n) = 3T(n/2) + O(n)$ solves to $O(n^{\log_2 3})$.

2.3. Derive Horner's rule for $f(x) = a_3x^3 + a_2x^2 + a_1x + a_0$ and show $O(n)$ operations.

2.4. Compute the Lagrange interpolating polynomial for points $(0, 1), (1, 2), (2, 5)$. Then implement it in Python.

2.5. Discuss why sparse representations are preferred for multivariate polynomials. What is the complexity of addition in terms of number of variables and terms?

2.6. Prove that over a field, the interpolating polynomial of degree at most $n - 1$ through n distinct points is unique.

2.7. Implement FFT-based multiplication in Python using NumPy (for classical simulation) and discuss its relation to QFT.

CHAPTER 3

Greatest Common Divisors and Euclidean Algorithms

The greatest common divisor (GCD) is a fundamental concept in algebra, essential for factorization, simplification of fractions, and solving Diophantine equations. This chapter explores the Euclidean algorithm for integers and polynomials, building directly on the arithmetic operations from Chapter 2 (e.g., division and remainder computation). We address challenges like coefficient explosion and introduce optimizations, drawing from von zur Gathen and Gerhard [5] for algorithmic details and Cox et al. [4] for geometric connections via resultants. These techniques are crucial for factorization in Chapter 4 and solving systems in Chapter 6.

1. Euclidean Algorithm for Integers

The Euclidean algorithm computes $\gcd(a, b)$ using repeated division, relying on efficient remainder operations from Chapter 2.

THEOREM 1.1 (Termination and Correctness). *For $a \geq b \geq 0$, the algorithm terminates in $O(\log b)$ steps, returning $\gcd(a, b)$.*

PROOF. By induction: Base case $b = 0$, $\gcd(a, 0) = a$. Inductive step: $\gcd(a, b) = \gcd(b, a \mod b)$, and $a \mod b < b$ strictly decreases the arguments. The worst-case number of steps is approximately $1.44 \log_2 b$ (Fibonacci sequence). □

Example 1.2. $\gcd(252, 105)$: $252 \mod 105 = 42$, $105 \mod 42 = 21$, $42 \mod 21 = 0 \to \gcd = 21$.

Standard implementation in Python:

```
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a
```

1.1. Extended Version. The extended Euclidean algorithm computes Bézout coefficients s, t such that $\gcd(a, b) = sa + tb$, useful for modular inverses and linear Diophantine equations.

Recursive implementation:

```
def extended_gcd(a, b):
    if not a:
        return b, 0, 1
    d, x, y = extended_gcd(b % a, a)
    return d, y - (b // a) * x, x
```

For example, `extended_gcd(252, 105)` returns $(21, -2, 5)$, so $21 = -2 \cdot 252 + 5 \cdot 105$.

Applications: Finding modular inverses if $\gcd(a, m) = 1$ (e.g., inverse of 17 mod 312), and in finite field arithmetic introduced in Chapter 2.

1.2. Coefficient Growth and Modular Reductions. Bézout coefficients grow in size up to $O(\log \max(a, b))$ bits, but Python's arbitrary-precision integers handle this seamlessly. For efficiency in large inputs, compute GCD modulo small primes first and lift solutions using techniques previewing Hensel lifting in Chapter 4. Coefficient explosion can be mitigated by binary GCD variants, which avoid large quotients.

Example 1.3. For large Fibonacci numbers, the standard algorithm performs well, but binary GCD reduces multiplications: Replace division with shifts and subtractions.

1.3. Exercises. Exercises

(1) Prove that $\gcd(a, b) = \gcd(b, a - bq)$ for any integer q.
(2) Implement the extended GCD in Python; use it to find the inverse of 17 modulo 312.
(3) Show the worst-case number of steps using Fibonacci numbers: $\gcd(F_{n+1}, F_n) = 1$ requires n steps.
(4) Estimate the bit complexity: Each operation costs $O((\log a)^2)$ per step using schoolbook arithmetic.
(5) Compare the standard Euclidean algorithm with the binary GCD on large inputs (e.g., 1000-bit numbers) for runtime.

2. Polynomial GCDs

Extending the algorithm to polynomial rings $K[x]$ over a field K, using polynomial division from Chapter 2.

2.1. Euclidean Algorithm over Fields. Replace integers with polynomials: Compute remainders until zero. For polynomials f, g of degrees $d \geq m$, the quotient is obtained by dividing leading terms, then subtracting.

Complexity: $O(d^2)$ polynomial operations, but over $\mathbb{Q}$, rational coefficients lead to $O(d^3)$ time due to fraction handling.

Using SymPy (as an alternative to SageMath for lightweight examples):

```
from sympy import symbols, gcd
x = symbols('x')
f = x**3 + 3*x**2 + 2*x + 1
g = x**2 + x + 1
print(gcd(f, g))  # x + 1
```

For multivariate polynomials, treat as univariate in one variable, but this is incomplete; full treatment requires Gröbner bases in Chapter 5.

2.2. Subresultants for Multivariate Cases. To avoid fraction growth in $\mathbb{Z}[x]$, use the polynomial remainder sequence (PRS) with subresultants, which are signed determinants of Sylvester submatrices. This keeps coefficients integer and bounds growth to $O(d^2 \log \|f\|_\infty)$.

THEOREM 2.1. *The GCD is proportional to the last non-zero subresultant in the sequence.*

This approach improves efficiency for Hensel lifting in Chapter 4 and connects to resultant computations below.

2.3. Resultant Computations and Sylvester Matrices. The resultant $\operatorname{res}(f, g)$ of two polynomials is the determinant of the Sylvester matrix, vanishing iff f and g share a common root. It generalizes GCD for detecting common factors and is computed via Euclidean algorithm on the coefficients.

Sylvester matrix for $\deg f = n$, $\deg g = m$ is $(n + m) \times (n + m)$ with shifted coefficients.

Example 2.2. For $f = x^2 + x + 1$, $g = x - 1$: Sylvester matrix

$$\operatorname{Syl}(f, g) = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 0 \end{pmatrix}$$

(det = 3), non-zero so coprime.

In SymPy:

```
from sympy import resultant
print(resultant(f, g))  # For above, compute accordingly
```

Applications: Elimination in algebraic geometry (Chapter 6) and factorization over extensions.

2.4. Exercises. Exercises

(1) Compute the polynomial GCD of $x^3 + x^2 - x - 1$ and $x^2 - 1$ over $\mathbb{Q}$.
(2) Show how the naive Euclidean algorithm over $\mathbb{Q}$ produces fractions, and how subresultants avoid this in $\mathbb{Z}[x]$.
(3) Implement the polynomial Euclidean algorithm in Python using lists for coefficients; handle leading coefficient normalization.
(4) For multivariate $f(x, y) = x^2 - y$, $g(x, y) = x - y^2$, compute the GCD treating y as fixed.
(5) Compute the resultant of $x^2 + ax + b$ and $x^2 + cx + d$; relate to common roots.

3. Advanced Techniques

3.1. Half-GCD Algorithms. The half-GCD algorithm, also known as HGCD, is a divide-and-conquer variant of the Euclidean algorithm designed to compute the GCD of two polynomials (or integers) more efficiently by halving the "size" (degree or bit length) at each step. It represents the steps of the Euclidean algorithm using matrix transformations and recurses on smaller subproblems. This approach is particularly effective when combined with fast polynomial multiplication techniques, such as Karatsuba or FFT-based methods from Chapter 2.

Consider two polynomials $f, g \in K[x]$ over a field K, with $\deg f = n \geq \deg g = m$. For simplicity, assume f and g are monic (leading coefficient $\operatorname{lc}(f) = \operatorname{lc}(g) = 1$). The HGCD algorithm computes a 2×2 matrix R with entries in $K[x]$ such that

$$R \begin{pmatrix} f \\ g \end{pmatrix} = \begin{pmatrix} h \\ k \end{pmatrix},$$

where h and k are consecutive remainders in the Euclidean sequence, $\gcd(f, g) = \gcd(h, k)$ (up to units), $\deg k < \deg h/2$, $\deg h \leq n/2$, and $\det R = \pm 1$.

The Euclidean algorithm can be viewed through linear algebra: Each step $r_{i-1} = q_i r_i + r_{i+1}$ (with negative sign convention) corresponds to a transformation

$$\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_i \end{pmatrix} \begin{pmatrix} r_{i-1} \\ r_i \end{pmatrix}.$$

Chaining these matrices allows skipping multiple steps efficiently.

The HGCD algorithm is defined recursively:

ALGORITHM 3.1 (HGCD for Polynomials). **Input:** Monic polynomials f, g with $\deg f \geq \deg g$.
Output: Matrix R as above.

If $\deg g < \lceil \deg f/2 \rceil$, return the identity matrix $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Otherwise:

(1) Compute the quotient $q = f \div g$ and remainder $r = f - qg$ (using polynomial division from Chapter 2).
(2) Recursively compute $S = \mathrm{HGCD}(g, r)$.
(3) Let $\begin{pmatrix} s \\ t \end{pmatrix} = S \begin{pmatrix} g \\ r \end{pmatrix}$.
(4) If $\deg s < \lceil \deg f/2 \rceil$, return $S \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$.
(5) Otherwise, compute $p = s \div t$ and $u = s - pt$.
(6) Recursively compute $T = \mathrm{HGCD}(t, u)$.
(7) Return $T \begin{pmatrix} 0 & 1 \\ 1 & -p \end{pmatrix} S \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$.

Remark 3.1. The ceiling $\lceil \cdot \rceil$ ensures progress for odd degrees. In practice, over fields like $\mathbb{Q}$, subresultant techniques (from §2) are integrated to control coefficient growth, making polynomials primitive or monic at each step.

Proposition 3.2 (Correctness)**.** *The returned matrix R satisfies the degree conditions and preserves the GCD, as each matrix has* $\det = \pm 1$*, and the remainders are part of the Euclidean sequence.*

PROOF. By induction on $n = \deg f$. Base case: If $\deg g < n/2$, $h = f$, $k = g$, degrees satisfy. Inductive step: The first recursion on (g, r) halves $\deg g \leq n/2$ (since $\deg r < \deg g$), and subsequent if needed halves further. The matrix compositions chain the transformations correctly. □

The time complexity satisfies the recurrence $T(n) = 2T(n/2) + O(M(n))$, where $M(n)$ is the multiplication cost (e.g., $M(n) = O(n \log n)$ with FFT, or $O(n^{1.585})$ with Karatsuba). By the master theorem, $T(n) = O(n \log^2 n)$ for FFT multiplication, as $O(M(n)) = O(n \log n)$ dominates.

To compute the full GCD using HGCD: - While $g \neq 0$: - Compute $R = \mathrm{HGCD}(f, g)$, get $h = R_{11}f + R_{12}g$, $k = R_{21}f + R_{22}g$. - Set $f = h/\operatorname{lc}(h)$ (normalize), $g = k/\operatorname{lc}(h)$. - Return f.

This avoids full remainder sequences for large degrees, integrating with Chapter 2's fast operations.

Example 3.3. Let $f = x^4 + 2x^3 + 3x^2 + 2x + 1$, $g = x^2 + x + 1$ (monic). Then $q = x^2 + x + 1$, $r = (x^2 + x + 1) - (x^2 + x + 1)g = 0$ wait, bad example.

Better: $f = x^5 + x^4 + x + 1$, $g = x^3 + x^2 + 1$. Compute steps manually to illustrate matrix.

3.2. Exercises. Exercises

(1) Prove the correctness of HGCD more formally: Show that the degrees halve and that the GCD is preserved under the matrix transformations.
(2) Derive the time complexity recurrence and solve it assuming Karatsuba multiplication $M(n) = O(n^{\log_2 3})$.

(3) Implement HGCD for univariate polynomials over $\mathbb{Q}$ in Python using SymPy (handle non-monic cases with content removal). Test on high-degree random polynomials and compare to built-in `gcd`.
(4) Adapt HGCD for integers: Replace degrees with bit lengths ($\lfloor \log_2 a \rfloor + 1$). Implement in Python and compare runtime on large Fibonacci pairs (e.g., 1000-bit numbers).
(5) Extend HGCD to multivariate polynomials by fixing a variable order (preview Gröbner bases in Chapter 5); discuss challenges.

4. Python Project

Implement the extended Euclidean algorithm in Python with SymPy for both integers and polynomials. Create a function to compute resultants and test it on examples from algebraic geometry, visualizing performance.

```
from sympy import symbols, gcd, Poly, resultant
x = symbols('x')

def poly_extended_gcd(f, g):
    # Implement using SymPy or custom for learning
    return gcd(f, g)  # Extend to coefficients

# Test resultant
f = x**3 + 3*x**2 + 2*x + 1
g = x**2 + x + 1
print(resultant(f, g))

# Report: Compare with SageMath if available, analyze time for
    large degrees.
```

- Test on polynomials from exercises; visualize coefficient growth.
- Submit a report on implementations.

CHAPTER 4

Projective Spaces and Weighted Projective Spaces

Projective geometry extends affine geometry by adding *points at infinity*, resolving issues like parallel lines and providing a unified framework for algebraic equations. This chapter introduces projective spaces, their homogeneous coordinates, and the weighted variant, which allows for more flexible constructions in toric geometry and singularity theory. These concepts are foundational for understanding geometric objects in later chapters on Gröbner bases and algebraic geometry applications. We draw from Cox et al. [4] for basic projective theory and Fulton [**?**] for toric aspects, emphasizing computational implementations in SageMath, which supports projective schemes natively.

1. Projective Spaces

Projective spaces homogenize affine spaces, compactifying them and enabling the study of properties invariant under projection.

1.1. Definition and Homogeneous Coordinates.

Definition 1.1. The *projective space* of dimension n over a field K, denoted $\mathbb{P}^n_K$ or $\mathbb{P}^n$, is the set of equivalence classes of nonzero points in K^{n+1} under scalar multiplication: $\mathbb{P}^n = (K^{n+1} \setminus \{0\})/K^*$, where $(\lambda x_0, \ldots, \lambda x_n) \sim (x_0, \ldots, x_n)$ for $\lambda \in K^*$.

Points are represented by *homogeneous coordinates* $[x_0 : x_1 : \cdots : x_n]$, where not all $x_i = 0$, and scaling doesn't change the point. Affine space K^n embeds via $U_i = \{[x_0 : \cdots : x_n] \mid x_i \neq 0\}$, dehomogenizing by setting $x_i = 1$. The *hyperplane at infinity* is $\mathbb{P}^n \setminus U_0$.

Example 1.2. The projective line $\mathbb{P}^1$ over $\mathbb{Q}$: Points $[x : y]$, with affine part $U_1 = \{[x : 1] \mid x \in \mathbb{Q}\} \cong \mathbb{Q}$, and infinity $[1 : 0]$. Parallels meet at infinity, e.g., lines $y = mx + c$ intersect at $[1 : 0]$ as $m \to \infty$.

Example 1.3. Projective plane $\mathbb{P}^2$: Contains equations like $x^2 + y^2 - z^2 = 0$, which dehomogenizes to $x^2 + y^2 = 1$ on U_z.

Polynomials on $\mathbb{P}^n$ are homogeneous: $f(x_0, \ldots, x_n)$ such that $f(\lambda \mathbf{x}) = \lambda^d f(\mathbf{x})$ for degree d. The ring is the graded ring $K[x_0, \ldots, x_n]$.

1.2. Implementation in SageMath. SageMath provides the `ProjectiveSpace` class for constructing and manipulating projective spaces.

```
from sage.all import *
P = ProjectiveSpace(2, QQ, names='x,y,z')
print(P) # Projective Space of dimension 2 over Rational Field
pt = P([1, 2, 3]) # Point [1 : 2 : 3]
print(pt)
```

Affine patches:

```
U = P.affine_patch(2) # U_z, coordinates x/z, y/z
```

This allows computational experiments, such as point counting over finite fields or intersection theory. For pure Python (SymPy), represent points as normalized lists (e.g., divide by last nonzero), but Sage is preferred for schemes.

1.3. Exercises. Exercises

(1) Show that $[1:0] \in \mathbb{P}^1$ is the point at infinity: Relate to limits of $x \to \infty$ in affine.
(2) Prove $\mathbb{P}^n$ is covered by $n+1$ affine charts $U_i \cong K^n$.
(3) Implement a function in Python to normalize homogeneous coordinates (divide by gcd or last coord).
(4) Compute the intersection of two lines in $\mathbb{P}^2$: $x+y-z=0$, $x-y+z=0$.
(5) Use Sage to define $\mathbb{P}^3_{\mathbb{Q}}$; find rational points.

2. Normalizing points in the Projective space

When working with points in projective space over the integers or rationals, it is often useful to represent them in a canonical form by normalizing the coordinates. This process minimizes the size of the integers involved, aiding in computations, storage, and comparisons.

Definition 2.1. For a point $p = [x_0 : x_1 : \cdots : x_n]$ in $\mathbb{P}^n(\mathbb{Q})$ with integer coordinates $x_i \in \mathbb{Z}$ (after clearing denominators), the *normalized representation* is obtained by dividing all coordinates by their greatest common divisor $d = \gcd(x_0, x_1, \ldots, x_n)$, resulting in a primitive tuple where $\gcd(x_0/d, \ldots, x_n/d) = 1$.

Normalization ensures uniqueness up to sign (since $[-x_0 : -x_1 : \cdots : -x_n] = [x_0 : x_1 : \cdots : x_n]$), which can be resolved by conventions like making the first nonzero coordinate positive.

Proposition 2.2. *Normalization preserves the point in $\mathbb{P}^n$ and reduces the naive height $h(p) = \max |x_i|$ for the normalized tuple.*

Proof. Since scaling by $1/d$ is allowed in homogeneous coordinates, the point remains the same. The gcd division minimizes the integers while keeping them integral, hence minimizes the max absolute value. □

Remark 2.3. Over finite fields $\mathbb{F}_p$, normalization might involve reducing modulo p, but gcd is replaced by ensuring coordinates are in $\{0, 1, \ldots, p-1\}$ with last nonzero coord 1.

Example 2.4. For point $[2:4:6]$ in $\mathbb{P}^2(\mathbb{Q})$, $\gcd(2,4,6) = 2$, normalize to $[1:2:3]$.

Example 2.5. Point $[0:3:0:6]$ in $\mathbb{P}^3$, $\gcd(0,3,0,6) = 3$, normalize to $[0:1:0:2]$.

Implementation in Python (using math.gcd for simplicity, extend to multiple via reduce):

```
from math import gcd
from functools import reduce

def normalize_proj_point(coords):
    if all(c == 0 for c in coords):
        raise ValueError("All coordinates zero")
    d = reduce(gcd, (abs(c) for c in coords if c != 0))
    norm_coords = [c // d for c in coords]
    # Make first nonzero positive
    first_nonzero = next(i for i, c in enumerate(norm_coords) if c
        != 0)
```

```
    if norm_coords[first_nonzero] < 0:
        norm_coords = [-c for c in norm_coords]
    return norm_coords
```

Test: `normalize_proj_point([2, -4, 6])` → [1, -2, 3] or adjust sign.
In SageMath, points are automatically normalized:

```
P = ProjectiveSpace(2, ZZ)
pt = P([2,4,6])  # Automatically [1:2:3]
```

This normalization is crucial for databases of rational points or moduli spaces.

Exercises

4.1. Normalize $[12 : 18 : 24 : 30]$ in $\mathbb{P}^3$; verify gcd=6.

4.2. Prove that normalization is unique up to units in $\mathbb{Z}$ (i.e., ± 1).

4.3. Implement normalization handling fractions: Clear denominators first.

4.4. Discuss why normalization fails over reals; suggest alternatives.

4.5. Use Sage to generate 10 random points in $\mathbb{P}^2(\mathbb{Q})$ and normalize them.

3. Weighted Projective Spaces

Weighted projective spaces generalize by assigning weights to coordinates, allowing orbifold structures and modeling singularities.

3.1. Definition and Properties.

Definition 3.1. Given positive integers $w_0, \dots, w_n$ (weights), the *weighted projective space* $\mathbb{P}(w_0, \dots, w_n)_K$ is $(K^{n+1} \setminus \{0\})/K^*$, where the action is $\lambda \cdot (x_0, \dots, x_n) = (\lambda^{w_0} x_0, \dots, \lambda^{w_n} x_n)$.

Points $[x_0 : \cdots : x_n]$, with equivalence under weighted scaling. If all $w_i = 1$, recovers standard $\mathbb{P}^n$. Often assume $\gcd(w_i) = 1$. Affine patches $U_i = \{x_i \neq 0\}$, but charts are quotients $\mathring{A}^n/\mu_{w_i}$ (cyclic group), introducing singularities if $w_i > 1$. Homogeneous polynomials satisfy $f(\lambda^{w_0} x_0, \dots) = \lambda^d f(x_0, \dots)$ for some degree d multiple of $\operatorname{lcm}(w_i / \gcd(d, w_i))$ or via grading. Weighted spaces arise in toric geometry, moduli spaces (e.g., $\mathbb{P}(1, 1, 2)$ for elliptic curves with level structure).

Example 3.2. $\mathbb{P}(1, 1)$ is $\mathbb{P}^1$. $\mathbb{P}(1, 2)$: Points $[x : y]$, $\lambda(x, y) = (\lambda x, \lambda^2 y)$. Singularity at $[0 : 1]/\mu_2$.

Example 3.3. $\mathbb{P}(1, 2, 3)$: Used for weighted equations, like $x^6 + y^3 + z^2 = 0$ (homogeneous under weights).

The coordinate ring is graded by the weights.

Proposition 3.4. *$\mathbb{P}(w_0, \dots, w_n)$ is constructible as a quotient or fan.*

Proof. As GIT quotient $(\mathring{A}^{n+1} \setminus 0)/\mathbb{CC}^*$, with weights defining the action [**?**]. □

Singularities at fixed points of the action.

3.2. Implementation in SageMath. SageMath supports weighted projective spaces via the toric varieties library.

```
from sage.all import *
P = toric_varieties.P( (1,2,3) ) # or WeightedProjectiveSpace
    ([1,2,3], QQ)
print(P) # Weighted projective space P(1,2,3)
pt = P.point([1,1,1])
print(pt)
```

Define weighted homogeneous ideals:

```
R = P.coordinate_ring() # Graded ring
x, y, z = R.gens()
I = R.ideal([x^6 + y^3 + z^2]) # Degree 6 under weights 1,2,3
X = P.subscheme(I)
print(X.dimension())
```

This enables computations like resolution of singularities or cohomology. For pure Python, simulate with classes for weighted points, normalizing via gcd of exponents adjusted by weights, but Sage is preferred for schemes.

3.3. Exercises. Exercises

(1) Show $\mathbb{P}(1,1,2) \cong \mathbb{P}^2/\mu_2$, describing the quotient map.
(2) Compute the singular locus of $\mathbb{P}(1,4,6)$: Find orbifold points.
(3) Implement a Python class for weighted points: Normalize coordinates under scaling.
(4) Use Sage to define $\mathbb{P}(2,3,5)$ and a weighted equation; check if smooth.
(5) Prove that for coprime weights, the space is well-formed (no worse singularities).

4. Normalizing points in the Weighted Projective space

Normalization in weighted projective spaces extends the standard case by accounting for the weights in the gcd computation, ensuring a canonical integer representation for rational points.

Definition 4.1. For a point $p = [x_0 : x_1 : \cdots : x_n]$ in $\mathbb{P}(w_0, \ldots, w_n)(\mathbb{Q})$ with integer coordinates $x_i \in \mathbb{Z}$ (after clearing denominators) and weights $w = (w_0, \ldots, w_n)$, the *weighted greatest common divisor* $\operatorname{wgcd}(x_0, \ldots, x_n)$ is the largest positive integer d such that d^{w_i} divides x_i for each $i = 0, \ldots, n$.

The point is *normalized* if $\operatorname{wgcd}(x) = 1$. To normalize, divide each x_i by d^{w_i}.

Remark 4.2. This differs from standard gcd: It respects the weighted scaling. For weights all 1, wgcd = gcd.

Proposition 4.3. *The wgcd exists and is unique, and normalization preserves the point since scaling by $1/d$ adjusts coordinates as $((1/d)^{w_0} x_0, \ldots) = (x_0/d^{w_0}, \ldots)$.*

Proof. Let primes factor each $|x_i| = \prod p_j^{\alpha_{j,i}}$. Then $d = \prod p_j^{\min_i \lfloor \alpha_{j,i}/w_i \rfloor}$. This d satisfies $d^{w_i} \leq \alpha_{j,i}$ for each prime, hence divides. Maximality by construction. □

Example 4.4. For weights $(1,2,3)$, point $[6 : 36 : 216]$. The factorizations are $6 = 2^1 \cdot 3^1$, $36 = 2^2 \cdot 3^2$, $216 = 2^3 \cdot 3^3$. For $p = 2$: $\min\lfloor 1/1, 2/2, 3/3 \rfloor = \min(1,1,1) = 1$. For $p = 3$: $\min\lfloor 1/1, 2/2, 3/3 \rfloor = \min(1,1,1) = 1$. Thus $d = 2^1 \cdot 3^1 = 6$. Normalize to $[6/6^1 : 36/6^2 : 216/6^3] = [1 : 1 : 1]$.

Example 4.5. For weights $(2, 3)$, point $[4 : 9]$. The factorizations are $4 = 2^2$, $9 = 3^2$. For $p = 2$: $\min\lfloor 2/2, 0/3\rfloor = \min(1, 0) = 0$. For $p = 3$: $\min\lfloor 0/2, 2/3\rfloor = \min(0, 0) = 0$. Thus $d = 1$, already normalized.

Algorithm: Factor each $|x_i|$, compute $\min(alpha_{p,i}/w_i)$ over i for each prime p, product $p^{\min}$.

In Python (naive, assumes small numbers; for large, use sympy.factorint):

```
from math import gcd
from collections import defaultdict
import sympy as sp

def weighted_gcd(coords, weights):
    if len(coords) != len(weights):
        raise ValueError("Mismatch")
    if all(c == 0 for c in coords):
        raise ValueError("Zero")
    # Handle signs: take abs
    abs_coords = [abs(c) for c in coords]
    primes_factors = [sp.factorint(a) if a != 0 else {} for a in
        abs_coords]
    all_primes = set()
    for fac in primes_factors:
        all_primes.update(fac.keys())
    exponents = {}
    for p in all_primes:
        min_exp = float('inf')
        for i, fac in enumerate(primes_factors):
            alpha = fac.get(p, 0)
            min_exp = min(min_exp, alpha // weights[i])
        if min_exp > 0:
            exponents[p] = min_exp
    d = 1
    for p, e in exponents.items():
        d *= p ** e
    return d

def normalize_weighted_point(coords, weights):
    d = weighted_gcd(coords, weights)
    norm_coords = [coords[i] // (d ** weights[i]) for i in range(
        len(coords))]
    # Sign convention: first nonzero positive
    first_nonzero = next(i for i, c in enumerate(norm_coords) if c
         != 0)
    if norm_coords[first_nonzero] < 0:
        norm_coords = [-c for c in norm_coords]
    return norm_coords
```

Test: `normalize_weighted_point([6,36,216], [1,2,3])` $\to$ [1,1,1]

This is efficient for storing points in moduli spaces [**?**].

4.1. Exercises. Exercises

(1) Compute wgcd of [8:81:243] with weights (3,4,5); normalize.
(2) Prove wgcd $= 1 \iff$ no $d > 1$ with $d^{w_i} | x_i$ for all i.
(3) Implement wgcd handling zero coordinates (skip if $x_i = 0$).
(4) Discuss uniqueness: For well-formed weights, unique up to roots of unity.
(5) Use code on $[12 : 144 : 1728]$ weights $(1, 3, 6)$; verify [1:1:1].

CHAPTER 5

Gröbner Bases and Buchberger's Algorithm

Gröbner bases are a cornerstone of computational algebra, providing a systematic way to solve systems of polynomial equations, test ideal membership, and perform eliminations. Named after Bruno Buchberger, who developed the algorithm in his 1965 PhD thesis, they generalize the Euclidean algorithm to multivariate polynomial rings. This chapter introduces the key concepts, including monomial orders and reductions, presents Buchberger's algorithm with proofs, and explores applications in elimination theory and solving systems. We draw primarily from Cox, Little, and O'Shea [4] for foundational theory, Buchberger [2] for the original algorithm, and Kreuzer and Robbiano [8] for advanced computations and optimizations. These tools are essential for the next chapter on solving polynomial systems and have connections to lattice reductions in Chapter 11.

1. Ideals and Monomial Orders

Gröbner bases rely on polynomial ideals and well-ordered monomials to enable algorithmic manipulations in multivariate rings.

1.1. Polynomial Rings and Ideals. Let K be a field (e.g., $\mathbb{Q}$ or $\mathbb{F}_p$). The *polynomial ring* in n variables is $K[x_1, \dots, x_n]$, consisting of finite sums of terms $c\alpha$, where $c \in K$ and $\alpha = x_1^{a_1} \cdots x_n^{a_n}$ is a monomial with multi-index $(a_1, \dots, a_n) \in \mathbb{N}^n$.

An *ideal* $I \subseteq K[x_1, \dots, x_n]$ is a subset closed under addition and multiplication by ring elements: If $f, g \in I$, then $f + g \in I$; if $h \in R$, $f \in I$, then $hf \in I$. By Hilbert's basis theorem, every ideal in $K[x_1, \dots, x_n]$ is finitely generated: $I = \langle f_1, \dots, f_s \rangle = \{h_1 f_1 + \cdots + h_s f_s \mid h_i \in K[x_1, \dots, x_n]\}$.

The *variety* of I is $V(I) = \{\mathbf{a} \in K^n \mid f(\mathbf{a}) = 0 \, \forall f \in I\}$. Conversely, the ideal of a variety V is $I(V) = \{f \in K[x_1, \dots, x_n] \mid f(\mathbf{a}) = 0 \ \forall \mathbf{a} \in V\}$. The Nullstellensatz relates them over algebraically closed fields [4].

1.2. Monomial Orders. A *monomial order* $>$ on the monomials of $K[x_1, \dots, x_n]$ is a total well-ordering (every nonempty set has a least element) satisfying: If $\alpha > \beta$, then $\alpha\gamma > \beta\gamma$ for all monomials γ.

Common orders: - *Lexicographic (lex)*: $x_1^{a_1} \cdots x_n^{a_n} > x_1^{b_1} \cdots x_n^{b_n}$ if the first differing exponent $a_i > b_i$. - *Deglex (graded lex)*: First compare total degree $\sum a_i > \sum b_i$, tie-break by lex. - *Degrevlex (graded reverse lex)*: Compare total degree, then reverse lex on exponents.

Example 1.1. In $\mathbb{Q}[x, y, z]$ with lex $x > y > z$: $x^2 y > x y^3 z^4$ since first exponents 1=1, then 2¿1 for y. With deglex: Total deg 3 ¡ 8, so reverse.

For a fixed order $>$, every nonzero polynomial f has a *leading term* $\mathrm{LT}(f) = c\alpha$ (monomial with coeff), *leading monomial* $\mathrm{LM}(f) = \alpha$, and *leading coefficient* $\mathrm{lc}(f) = c$. The *leading ideal* of I is $\langle \mathrm{LT}(I) \rangle = \langle \mathrm{LT}(f) \mid f \in I \setminus \{0\} \rangle$.

Monomial orders enable division algorithms similar to univariate cases.

1.3. Reductions and Normal Forms. The multivariate division algorithm reduces f by a set $G = \{g_1, \ldots, g_s\}$: While $\mathrm{LT}(f)$ is divisible by some $\mathrm{LT}(g_i)$, subtract a multiple to cancel $\mathrm{LT}(f)$. Output $f = \sum a_i g_i + r$, where no term of r is divisible by any $\mathrm{LT}(g_i)$ (remainder r is reduced w.r.t. G).

However, r may not be unique without additional structure.

A *Gröbner basis* G for I is a generating set where $\langle \mathrm{LT}(I) \rangle = \langle \mathrm{LT}(g_1), \ldots, \mathrm{LT}(g_s) \rangle$. Then, reductions yield unique normal forms [4].

THEOREM 1.2 (Existence). *Every ideal $I \neq \{0\}$ has a Gröbner basis w.r.t. any monomial order.*

PROOF. By Dickson's lemma (below), the monomial ideal $\langle \mathrm{LT}(I) \rangle$ is finitely generated by monomials $\mathrm{LM}(f_1), \ldots, \mathrm{LM}(f_s)$ for some $f_i \in I$. Then $G = \{f_1, \ldots, f_s\}$ generates $\langle \mathrm{LT}(I) \rangle$. □

Lemma 1.3 (Dickson's Lemma)**.** *Every monomial ideal in $K[x_1, \ldots, x_n]$ is generated by finitely many monomials.*

PROOF. By induction on n. For $n = 1$, monomials x^k for $k \geq k_0$ minimal. For n, let $J_m = \{(a_2, \ldots, a_n) \mid x_1^m x_2^{a_2} \cdots x_n^{a_n} \in I\}$ ideals in fewer variables, finitely generated. Collect generators across m, show finite set suffices [4]. □

Ideal membership: $f \in I$ iff reduction of f w.r.t. Gröbner basis is zero.

1.4. Exercises. Exercises

(1) Compare lex and deglex on monomials x^2y, xy^2, y^3 in $\mathbb{Q}[x, y]$.
(2) Prove that lex is a monomial order: Show it's total, well-ordered, and multiplicative.
(3) Perform multivariate division: Divide $x^2y + xy^2 + y^2$ by $\{xy - 1, y^2 - 1\}$ w.r.t. lex $x > y$.
(4) Show that if G is a Gröbner basis, reductions are unique (use contradiction on two remainders).
(5) Compute a minimal monomial generating set for $\langle x^4y^2, x^3y^3, x^2y^5 \rangle$ in $\mathbb{Q}[x, y]$.

2. Buchberger's Algorithm

Buchberger's algorithm computes a Gröbner basis from any generating set.

2.1. S-Pairs and Reductions. For f, g with $\mathrm{LM}(f) = \alpha$, $\mathrm{LM}(g) = \beta$, the *least common multiple* $\mathrm{LCM}(\alpha, \beta)$ is the monomial with max exponents. The *S-pair* is $S(f, g) = \frac{\mathrm{LCM}}{\mathrm{LT}(f)} f - \frac{\mathrm{LCM}}{\mathrm{LT}(g)} g$.

Buchberger's criterion: G is a Gröbner basis iff every S-pair $S(g_i, g_j)$ reduces to zero w.r.t. G.

THEOREM 2.1 (Buchberger's Criterion). *A basis $G = \{g_1, \ldots, g_s\}$ for I is Gröbner iff for all $i \neq j$, the remainder of $S(g_i, g_j)$ on division by G is zero.*

PROOF. ($\Rightarrow$) If Gröbner, $\mathrm{LT}(I)$ generated by $\mathrm{LT}(G)$, so $\mathrm{LT}(S(g_i, g_j))$ divisible (cancels leading terms), and full reduction to zero by definition.

($\Leftarrow$) Suppose not Gröbner: Some $f \in I$ with $\mathrm{LT}(f)$ not in $\langle \mathrm{LT}(G) \rangle$. Choose minimal such f by monomial order. Write $f = \sum a_k g_k$, assume $\mathrm{LT}(f) = \mathrm{LT}(a_i g_i)$ for some i (otherwise contradict minimality). But then adjust, leading to S-pair contradictions [2], [4]. □

The algorithm:

ALGORITHM 2.1 (Buchberger's Algorithm). **Input:** $F = \{f_1, \ldots, f_s\}$ generating I, monomial order $>$.
Output: Gröbner basis G for I.
Set $G = F$.
While there are pairs $g_i, g_j \in G$ with nonzero remainder r of $S(g_i, g_j)$ w.r.t. G: Add r to G.
Return G.

2.2. Termination via Dickson's Lemma.

THEOREM 2.2 (Termination). *Buchberger's algorithm terminates after finitely many steps.*

PROOF. Each added $r \neq 0$ introduces a new $\mathrm{LM}(r)$ not divisible by previous $\mathrm{LT}(G)$, so the leading ideal strictly increases. By Dickson's lemma, monomial ideals are Noetherian (no infinite ascending chains), so terminates [4]. □

In practice, compute all S-pairs, reduce using current G, add nonzero remainders.

Complexity: Can be doubly exponential in worst case, but often practical.

2.3. Optimizations: F4/F5 Variants. Buchberger is naive; optimizations include: - *Buchberger criteria*: Skip pairs where $\mathrm{LCM}(\mathrm{LM}(f), \mathrm{LM}(g))$ coprime (no common variables). - *F4 algorithm* (Faugarè): Matrix-based, reduces multiple polynomials simultaneously using linear algebra over Macaulay matrices [8]. - *F5 algorithm*: Signature-based, avoids unnecessary reductions by tracking syzygies.

These reduce to singly exponential in many cases.

SageMath example:

```
R.<x,y,z> = PolynomialRing(QQ, order='lex')
I = ideal(x*y - z, y*z - x, z*x - y)
G = I.groebner_basis()  # Computes via Buchberger/F4
```

2.4. Exercises. Exercises

(1) Compute S-pair of $x^2 - y, x - y^2$ w.r.t. lex $x > y$; reduce w.r.t. themselves.
(2) Prove Buchberger criterion for two generators: Relate to resultant vanishing.
(3) Implement basic Buchberger in Python with SymPy: Handle pairs, reductions.
(4) Show termination fails without well-ordering: Give infinite loop example.
(5) Optimize: Prove coprime LCM criterion skips safely (S-pair reduces to zero).

3. Elimination Theory

Elimination removes variables from polynomial systems, linking to resultants from Chapter 3.

3.1. Resultants and Discriminants. For $f, g \in K[x]$ univariate, $\mathrm{res}(f, g)$ detects common roots. Multivariate: Treat as univariate in one variable, coefficients in $K[y_1, \ldots]$.

The resultant $\mathrm{res}_x(f, g)$ w.r.t. x vanishes iff f, g share a root in x (or both zero).

Computed via Sylvester determinant, or Euclidean algorithm (product of differences of roots).

Discriminant $\mathrm{disc}(f) = \mathrm{res}(f, f')$ measures multiple roots.

For bivariate systems $f(x,y)=0, g(x,y)=0$, compute $\operatorname{res}_y(f,g)$ to eliminate y, get condition on x.

Example 3.1. Eliminate y from $x-y^2=0$, $xy-1=0$: $\operatorname{res}_y(x-y^2, xy-1) = x^2(-1/x) - (-1/x)x^2$ wait, proper computation via Sylvester.

Macaulay matrices generalize to higher degrees/systems: For homogeneous polynomials, the matrix whose det is multiple of resultant.

3.2. Solving Bivariate Systems. For two equations in two variables, compute resultant to get univariate in one, solve, back-substitute.

Proposition 3.2. *If f, g generate a zero-dimensional ideal (finite solutions), the resultant degree bounds number of solutions (Bézout).*

Example code in SymPy:

```
from sympy import symbols, resultant
x, y = symbols('x y')
f = x - y**2
g = x*y - 1
print(resultant(f, g, y))  # Result in x
```

Limitations: High degree, numerical instability; Gröbner better for general.

3.3. Exercises. Exercises

(1) Compute $\operatorname{res}(x^2+xy+1, xy-1)$ w.r.t. y via Sylvester.
(2) Show $\operatorname{disc}(x^2+bx+c) = b^2-4c$.
(3) Solve system $x^2+y^2=1$, $xy=1/2$ via elimination.
(4) Prove resultant homogeneous: $\operatorname{res}(\lambda f, \mu g) = \lambda^{\deg g}\mu^{\deg f}\operatorname{res}(f,g)$.
(5) Implement bivariate solver using resultants in Python.

4. Gröbner Bases for Systems

Gröbner bases shine in solving systems via elimination.

4.1. Elimination Orders. An order is *elimination order* for $x_1,\dots,x_k$ if monomials involving them are larger than those without.

Lex order eliminates naturally: Gröbner basis includes polynomials in fewer variables.

THEOREM 4.1 (Elimination Theorem). *If G is Gröbner w.r.t. elimination order for $x_1 > \cdots > x_k > x_{k+1},\dots,x_n$, then $G \cap K[x_{k+1},\dots,x_n]$ is Gröbner for $I \cap K[x_{k+1},\dots,x_n]$.*

PROOF. Leading terms in eliminated variables don't appear in intersection; generators cover [4]. □

Solve by successive elimination: Get univariate, factor, back-substitute.

4.2. Solving via Triangular Forms. A *reduced Gröbner basis* is monic, no term divisible by other LTs. In lex order for radical zero-dim ideals, it's triangular: $g_1(x_1), g_2(x_1,x_2), \dots, g_n(x_1,\dots,x_n)$.

Solve like Gaussian elimination.

For positive dimension, need variety decomposition.

4.3. Variety Decomposition Basics. Irreducible varieties correspond to prime ideals; primary decomposition breaks I into primaries.

Algorithmically hard, but Gröbner helps compute dimension, radical ($\sqrt{I}$), via extensions.

Dim I = max deg of monomials not in $\langle \mathrm{LT}(I) \rangle$ (monomial basis size).

Example 4.2. For twisted cubic $I = \langle xz - y^2, xw - yz \rangle$, degrevlex basis, dim=2 (curve).

4.4. Exercises. Exercises

(1) Compute lex Gröbner for $\langle x^2 + y^2 - 1, x - y \rangle$ in $\mathbb{Q}[x, y]$; solve system.
(2) Prove elimination theorem for k=1: Show intersection generates.
(3) Find dim of $\langle xy, xz, yz \rangle$ via monomial basis.
(4) Decompose $\langle x^2 - x, xy \rangle = \langle x \rangle \cap \langle x - 1, y \rangle$.
(5) Implement solver: Use SageMath Gröbner, back-substitute roots.

5. Python Project

Compute Gröbner bases in SageMath for a system representing geometry (e.g., circle intersections). Script to automate, visualize varieties.

```
R.<x,y> = PolynomialRing(QQ, order='lex')
I = ideal(x**2 + y**2 - 1, (x-1)**2 + y**2 - 1)
G = I.groebner_basis()
print(G)
# Solve: univariate in y, etc.
# Visualize: Use matplotlib for points.
```

- Test on robotics (intersection of constraints).
- Report on basis, solutions, compare orders for speed.

CHAPTER 6

Introduction to Quantum Computing

This chapter provides a foundational introduction to quantum computing, tailored for students with a background in linear algebra and abstract algebra. We focus on concepts essential for understanding quantum algorithms in computational algebra, such as those used in factoring and solving linear systems. The material draws from Nielsen and Chuang [10] and aims to bridge classical algebraic structures with their quantum counterparts.

1. Quantum Fundamentals

Quantum computing operates on principles of quantum mechanics, where information is processed using quantum bits, or *qubits*, rather than classical bits. Unlike a classical bit, which is either 0 or 1, a qubit can exist in a superposition of states.

The mathematical framework for qubits is rooted in linear algebra over the complex numbers, analogous to vector spaces in abstract algebra but with an inner product structure.

Definition 1.1 (Hilbert Space for a Qubit)**.** The state of a qubit is a vector in the 2-dimensional complex Hilbert space $\mathcal{H} = \mathbb{C}^2$, equipped with the standard inner product $\langle u, v\rangle = u^\dagger v$, where $u^\dagger$ denotes the conjugate transpose.

Definition 1.2 (Qubit)**.** A *qubit* is the basic unit of quantum information, represented as a unit vector in $\mathbb{C}^2$. The standard (computational) basis states are denoted as

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

where $|\cdot\rangle$ is the Dirac notation for a ket vector. A general qubit state is a superposition:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle,$$

with $\alpha, \beta \in \mathbb{C}$ satisfying the normalization condition $|\alpha|^2 + |\beta|^2 = 1$.

The normalization ensures that probabilities sum to 1, as we will see in the Born rule. Dirac notation, also known as bra-ket notation, is fundamental in quantum mechanics and provides a compact way to express states and operations.

Definition 1.3 (Dirac Notation)**.** A ket $|\psi\rangle$ represents a column vector in $\mathcal{H}$. The corresponding bra $\langle\psi|$ is the conjugate transpose, a row vector. The inner product between two states $|\psi\rangle$ and $|\phi\rangle$ is

$$\langle\psi|\phi\rangle = \psi^\dagger \phi.$$

The outer product is denoted $|\psi\rangle\langle\phi| = |\psi\rangle\langle\phi|$, which is a rank-1 operator.

Example 1.4. For $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the bra is $\langle\psi| = \alpha^*\langle 0| + \beta^*\langle 1|$, where $*$ denotes complex conjugate. The norm squared is $\langle\psi|\psi\rangle = |\alpha|^2 + |\beta|^2 = 1$.

Definition 1.5 (Superposition)**.** *Superposition* refers to the property that a quantum state can be expressed as a linear combination of basis states with complex coefficients (amplitudes). For example, the equal superposition state is

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle).$$

This is analogous to linear combinations in vector spaces, but with probabilistic interpretation upon measurement.

Measurement in quantum mechanics is probabilistic and causes the state to collapse.

THEOREM 1.6 (Born Rule). *Given a qubit in state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$, the probability of measuring outcome 0 (in the computational basis) is $|\alpha|^2$, and the probability of measuring 1 is $|\beta|^2$. After measurement, if outcome k is observed ($k = 0$ or 1), the state collapses to $|k\rangle$.*

PROOF. The Born rule is a postulate of quantum mechanics, but it can be motivated as follows: The measurement in the basis $\{|0\rangle, |1\rangle\}$ corresponds to projectors $P_0 = |0\rangle\langle 0|$ and $P_1 = |1\rangle\langle 1|$. The probability of outcome 0 is the expectation value $\langle\psi|P_0|\psi\rangle = |\langle 0|\psi\rangle|^2 = |\alpha|^2$. Similarly for outcome 1. Post-measurement, the state is projected onto the eigenspace of the observed outcome, normalized: for outcome 0, it becomes $\frac{P_0|\psi\rangle}{||P_0|\psi\rangle||} = |0\rangle$ (since $||P_0|\psi\rangle|| = |\alpha|$ but normalization absorbs it; assuming $\alpha \neq 0$). □

Remark 1.7. The collapse is irreversible and introduces randomness, distinguishing quantum from classical computation. In algebraic terms, measurements can be seen as projections in a Hilbert module over $\mathbb{C}$.

For systems with multiple qubits, the state space is the tensor product of individual spaces, leading to exponential growth in dimension—a key source of quantum computational power.

Proposition 1.8 (Multi-Qubit State Space)**.** *The Hilbert space for n qubits is $\mathcal{H}^{\otimes n} = (\mathbb{C}^2)^{\otimes n} \cong \mathbb{C}^{2^n}$, with basis states $|x_1 x_2 \dots x_n\rangle$ where each $x_i \in \{0, 1\}$. A general state is*

$$|\psi\rangle = \sum_{x\in\{0,1\}^n} \alpha_x |x\rangle,$$

with $\sum_x |\alpha_x|^2 = 1$.

PROOF. The tensor product construction follows from linear algebra: $\dim(\mathcal{H}_1 \otimes \mathcal{H}_2) = \dim(\mathcal{H}_1)\dim(\mathcal{H}_2) = 2\times 2 = 4$ for two qubits, and inductively 2^n for n qubits. The basis is the tensor product of single-qubit bases, e.g., $|00\rangle = |0\rangle \otimes |0\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$. Normalization extends naturally: $\langle\psi|\psi\rangle = \sum_x |\alpha_x|^2 = 1$. □

This exponential dimension enables quantum parallelism: a quantum algorithm can process 2^n amplitudes simultaneously, useful for algebraic problems like factoring via period-finding.

However, not all multi-qubit states are simple products; some exhibit entanglement.

Definition 1.9 (Entanglement)**.** An n-qubit state $|\psi\rangle \in (\mathbb{C}^2)^{\otimes n}$ is *separable* (non-entangled) if it can be written as a tensor product $|\psi\rangle = |\phi_1\rangle \otimes |\phi_2\rangle \otimes \cdots \otimes |\phi_n\rangle$ for

single-qubit states $|\phi_i\rangle$. Otherwise, it is *entangled*. For two qubits, $|\psi\rangle$ is entangled if it cannot be expressed as $|a\rangle \otimes |b\rangle$. A classic example is the Bell state:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle).$$

Measuring the first qubit determines the second instantly, exhibiting correlations stronger than classical (violating Bell inequalities).

Lemma 1.10 (Criterion for Bipartite Separability)**.** *A two-qubit state $|\psi\rangle = \sum_{ij} c_{ij}|ij\rangle$ is separable if and only if there exist $\alpha_i, \beta_j \in \mathbb{C}$ such that $c_{ij} = \alpha_i \beta_j$ for all i, j.*

PROOF. If separable, $|\psi\rangle = (\sum_i \alpha_i |i\rangle) \otimes (\sum_j \beta_j |j\rangle) = \sum_{ij} \alpha_i \beta_j |ij\rangle$, so $c_{ij} = \alpha_i \beta_j$. Conversely, if $c_{ij} = \alpha_i \beta_j$, then it factors as above (assuming normalization separately for each factor). For the Bell state, $c_{00} = c_{11} = 1/\sqrt{2}$, $c_{01} = c_{10} = 0$; suppose $c_{00} = \alpha_0 \beta_0 = 1/\sqrt{2}$, $c_{01} = \alpha_0 \beta_1 = 0 \implies \beta_1 = 0$, but then $c_{11} = \alpha_1 \beta_1 = 0$, contradiction. Thus, entangled. □

Remark 1.11. Entanglement is crucial for quantum speedups in algorithms like Shor's, where it enables efficient sampling from correlated distributions in algebraic groups (e.g., period-finding in $\mathbb{Z}_N^\times$). It also connects to tensor decompositions in multilinear algebra.

Exercises

6.1. Verify that the state $|\psi\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$ is separable by finding explicit single-qubit factors. Compute the probabilities of measuring 00, 01, etc., using the Born rule.

6.2. Using matrix representations, show that the Bell state $|\Phi^+\rangle$ cannot be written as a product state. Discuss how this relates to the concept of irreducible representations in group theory.

6.3. Prove that for any two-qubit state, the partial trace over one qubit yields a density matrix (operator) with trace 1. (Hint: Define the density matrix $\rho = |\psi\rangle\langle\psi|$ and trace out.)

2. Quantum Gates and Circuits

Quantum computations are performed using quantum gates, which are unitary operators on the Hilbert space. Gates manipulate qubit states reversibly and can be composed into circuits to implement algorithms. This section expands on key gates, their properties, and circuit construction, emphasizing connections to linear algebra and group theory relevant to computational algebra.

Definition 2.1 (Quantum Gate)**.** A *quantum gate* acting on k qubits is a unitary operator $U : (\mathbb{C}^2)^{\otimes k} \to (\mathbb{C}^2)^{\otimes k}$ such that $U^\dagger U = U U^\dagger = I$, where I is the identity operator. Unitarity preserves the norm of states, ensuring probabilities remain valid under evolution.

Proposition 2.2 (Unitarity Implies Reversibility)**.** *Every quantum gate U is invertible, with inverse $U^\dagger$. Thus, quantum computation is inherently reversible, unlike some classical operations (e.g., AND gate).*

PROOF. From $U^\dagger U = I$, multiplying both sides by U on the right yields $U^\dagger = U^{-1}$. Similarly from $U U^\dagger = I$. □

Single-qubit gates are 2×2 unitary matrices. They form the special unitary group $SU(2)$, up to a global phase, which is isomorphic to the rotation group $SO(3)$—a connection useful in algebraic topology and representation theory.

Important examples include the Pauli gates, which are both unitary and Hermitian ($U^\dagger = U$).

Example 2.3 (Pauli Gates)**.** The Pauli matrices are:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

- The X gate (bit-flip) maps $|0\rangle \mapsto |1\rangle$ and $|1\rangle \mapsto |0\rangle$, analogous to classical NOT. - The Z gate (phase-flip) maps $|0\rangle \mapsto |0\rangle$ and $|1\rangle \mapsto -|1\rangle$. - The Y gate combines bit and phase flips: $Y = iXZ$.

Lemma 2.4 (Pauli Gates are Unitary and Hermitian)**.** *Each Pauli gate $\sigma \in \{X, Y, Z\}$ satisfies $\sigma^\dagger = \sigma$ and $\sigma^2 = I$.*

Proof. For X: $X^\dagger = X$ (real symmetric), and $X^2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = I$. For Z: Similar, $Z^\dagger = Z$, $Z^2 = I$. For Y: $Y^\dagger = \begin{pmatrix} 0 & i \\ -i & 0 \end{pmatrix}^\dagger = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = Y$ (since conjugate transpose flips signs of i), and $Y^2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = \begin{pmatrix} (-i)(i) & 0 \\ 0 & (-i)(i) \end{pmatrix} = I$. Unitarity follows from $\sigma^\dagger \sigma = \sigma^2 = I$. □

Remark 2.5. The Pauli gates, together with I, form a basis for the vector space of 2×2 Hermitian matrices and generate the Clifford group under multiplication, relevant for quantum error correction codes based on algebraic groups.

Another essential single-qubit gate is the Hadamard gate, which creates superpositions.

Example 2.6 (Hadamard Gate)**.** The Hadamard gate is

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

It transforms basis states as follows:

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle, \quad H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle.$$

Applying H to a superposition state can interfere amplitudes constructively or destructively.

Proposition 2.7 (Properties of Hadamard Gate)**.** *The Hadamard gate is unitary, Hermitian, and its own inverse: $H^2 = I$.*

Proof. First, $H^\dagger = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}^\dagger = H$ (real symmetric), so Hermitian. Then, $H^2 = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \frac{1}{2} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} = I$. Unitarity follows as $H^\dagger H = H^2 = I$. □

Remark 2.8. The Hadamard gate is crucial in algorithms like the Quantum Fourier Transform (QFT), which generalizes the discrete Fourier transform over abelian groups, connecting to algebraic number theory in Shor's algorithm.

To create entanglement and perform multi-qubit operations, we use controlled gates, which condition an operation on the state of a control qubit.

Definition 2.9 (Controlled-U Gate)**.** For a single-qubit unitary U, the controlled-U (CU) gate on two qubits (control first) applies U to the target if the control is $|1\rangle$, and does nothing otherwise. Its matrix is

$$\mathrm{CU} = \begin{pmatrix} I & 0 \\ 0 & U \end{pmatrix},$$

where I and U are 2×2 blocks.

The most common is the controlled-NOT (CNOT), where $U = X$.

Definition 2.10 (CNOT Gate)**.** The CNOT gate is

$$\mathrm{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

It flips the target if control is $|1\rangle$: $\mathrm{CNOT}|xy\rangle = |x(y \oplus x)\rangle$, where $\oplus$ is modulo-2 addition.

Proposition 2.11 (CNOT is Unitary)**.** *CNOT satisfies* $(CNOT)^\dagger CNOT = I$.

Proof. CNOT is real, so $(\mathrm{CNOT})^\dagger = \mathrm{CNOT}^T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ (same as CNOT, permutation matrix). Then $(\mathrm{CNOT})^2 = I$, as it swaps basis states in pairs reversibly. □

Example 2.12 (Creating Bell State with Circuit)**.** To create the Bell state $|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$: Start with $|00\rangle$, apply H to the first qubit ($\frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$), then CNOT with first as control ($\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$).

Quantum circuits are compositions of gates applied in sequence or parallel to qubits.

Definition 2.13 (Quantum Circuit)**.** A *quantum circuit* is a sequence of quantum gates applied to an initial state $|\psi_0\rangle$, evolving it to $U_m \cdots U_1|\psi_0\rangle$, where each U_i is a gate (possibly acting on subsets of qubits via tensor products with identities). Circuits are visualized with horizontal wires for qubits (time flows left to right) and symbols for gates (e.g., H for Hadamard, $\bullet - - - \oplus$ for CNOT).

Theorem 2.14 (Universality of Quantum Gates). *Any unitary operator on n qubits can be approximated to arbitrary precision using a finite set of universal gates, such as $\{H, S, T, CNOT\}$, where $S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$ (phase gate) and $T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}$ ($\pi/8$ gate).*

Remark 2.15. Proofs of universality rely on Lie group theory (e.g., generating dense subgroups of $SU(2^n)$) and are detailed in [10]. This enables simulation of any quantum algorithm, connecting to computability in algebraic settings.

For practical simulation and experimentation, we use Qiskit, an open-source framework from IBM.

Simple Hadamard Circuit in Qiskit

```
from qiskit import QuantumCircuit, Aer, execute
from qiskit.visualization import plot_histogram

qc = QuantumCircuit(1, 1)  # 1 qubit, 1 classical bit
qc.h(0)                    # Apply Hadamard to qubit 0
qc.measure(0, 0)           # Measure qubit 0 to classical
    bit 0

simulator = Aer.get_backend('qasm_simulator')
result = execute(qc, simulator, shots=1024).result()
counts = result.get_counts(qc)
plot_histogram(counts)
```

This circuit demonstrates superposition: measurements yield approximately 50% $|0\rangle$ and 50% $|1\rangle$.

Bell State Circuit in Qiskit

```
qc = QuantumCircuit(2, 2)
qc.h(0)          # Superposition on qubit 0
qc.cx(0, 1)      # CNOT with control 0, target 1
qc.measure([0,1], [0,1])

result = execute(qc, simulator, shots=1024).result()
counts = result.get_counts(qc)
plot_histogram(counts)  # ~50% '00', ~50% '11'
```

This illustrates entanglement: correlated measurements.

Circuits can be visualized using `qc.draw('mpl')` in Qiskit, aiding in debugging quantum algorithms for algebraic problems, such as implementing group operations in hidden subgroup problems.

Exercises

6.4. Verify that the Pauli Y gate can be written as $Y = iXZ$ and compute its action on $|+\rangle$. Show that the Pauli gates satisfy the commutation relations $[X, Y] = 2iZ$ (cyclic permutations).

6.5. Construct the matrix for the controlled-Z (CZ) gate and prove it is unitary. Use it in a circuit to create another Bell state, e.g., $\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$.

6.6. In Qiskit, simulate a circuit with Hadamard on two qubits followed by measurements. Compare the outcome distribution to the tensor product of single-qubit superpositions, highlighting the absence of entanglement.

6.7. Prove that the set $\{H, \text{CNOT}\}$ generates the Clifford group for two qubits, which stabilizes the Pauli group under conjugation (relevant for error-correcting codes).

3. Quantum Linear Algebra Review

Quantum states and operations have a natural interpretation in linear algebra over complex numbers, providing a bridge to classical computational algebra.

Definition 3.1 (Hilbert Space)**.** A *Hilbert space* $\mathcal{H}$ is a complete inner product space over $\mathbb{C}$. For n qubits, $\mathcal{H} = (\mathbb{C}^2)^{\otimes n} \cong \mathbb{C}^{2^n}$, with the inner product $\langle\psi|\phi\rangle = \psi^\dagger\phi$.

Quantum states are unit vectors in $\mathcal{H}$, and measurements correspond to projections onto subspaces.

Definition 3.2 (Unitary Operator)**.** A linear operator $U : \mathcal{H} \to \mathcal{H}$ is *unitary* if $U^\dagger U = UU^\dagger = I$. All quantum gates are unitary, ensuring reversible computation.

The connection to classical algebra is evident: quantum circuits implement matrix multiplications in high dimensions, but with exponential parallelism. For instance, applying a gate U to n qubits is equivalent to the tensor product $U \otimes I^{\otimes(n-1)}$, mirroring operations in algebraic tensor algebras.

Inner products enable computations like fidelity between states, useful in quantum error correction and algebraic decoding. In later chapters, we explore algorithms like HHL, which solve linear systems $Ax = b$ by encoding matrices into unitaries and leveraging quantum phase estimation—a quantum analog of eigenvalue decomposition in classical linear algebra [9].

Exercises

6.8. Using Qiskit (integrated with Python), build and simulate a simple quantum circuit that demonstrates superposition and measurement. Visualize the results and compare with classical probability simulations in SymPy.

6.9. Construct a two-qubit circuit in Qiskit that creates a Bell state using Hadamard and CNOT gates. Simulate measurements and discuss how the results illustrate entanglement, contrasting with separable classical product states.

6.10. Using SymPy, represent the Hadamard gate as a matrix and compute its action on the basis vectors. Then, verify unitarity by checking $H^\dagger H = I$. Extend this to a symbolic representation of a general single-qubit state.

CHAPTER 7

Quantum Algorithms Basics

This chapter introduces some of the fundamental quantum algorithms that illustrate the power of quantum computing over classical methods, with a special emphasis on their applications in computational algebra. We begin with the *Quantum Fourier Transform* (QFT), a central primitive that enables efficient phase estimation. We then discuss *Grover's algorithm* for unstructured search, which provides a quadratic speedup. Finally, we present *Shor's algorithm*, a period-finding algorithm that underlies polynomial-time quantum factoring and discrete logarithm computations.

These algorithms exemplify how quantum mechanics—through superposition, entanglement, and interference—can lead to dramatic speedups, often exponential, for problems with deep connections to algebraic structures such as groups, fields, and number theory. The material builds on the quantum fundamentals developed in Chapter 6 and follows standard references such as Nielsen and Chuang [**10**].

1. Quantum Fourier Transform (QFT)

The *Quantum Fourier Transform* (QFT) is the quantum analogue of the discrete Fourier transform (DFT). Just as the DFT decomposes a function into its frequency components, the QFT transforms the amplitudes of quantum states into the Fourier domain. It is the backbone of many quantum algorithms, most notably phase estimation and Shor's algorithm.

Definition 1.1 (Quantum Fourier Transform)**.** For an n-qubit register, the QFT acts on the computational basis states as

$$\mathrm{QFT}\,|j\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i jk/2^n} |k\rangle,$$

where $j \in \{0, 1, \ldots, 2^n - 1\}$. In matrix form, QFT is the $2^n \times 2^n$ unitary matrix with entries

$$[\mathrm{QFT}]_{jk} = \frac{1}{\sqrt{2^n}} \omega^{jk}, \quad \omega = e^{2\pi i/2^n}.$$

Proposition 1.2 (Unitarity of QFT)**.** *The operator* QFT *is unitary, i.e.*

$$(\mathrm{QFT})^\dagger \,\mathrm{QFT} = I.$$

Proof. The inverse QFT is obtained by replacing ω with ω^{-1}. Orthogonality of roots of unity gives

$$\sum_{k=0}^{2^n-1} \omega^{k(m-l)} = 2^n \delta_{ml}.$$

With normalization by $1/\sqrt{2^n}$, the rows and columns of QFT are orthonormal, proving unitarity. □

Remark 1.3. The QFT generalizes the DFT over the cyclic group $\mathbb{Z}/2^n\mathbb{Z}$, and more broadly, it is a special case of Fourier analysis on finite abelian groups. This

perspective is essential in algebraic number theory and the hidden subgroup problem.

The QFT can be implemented efficiently on a quantum circuit with $O(n^2)$ gates, which is exponentially faster than the naive classical computation of the DFT.

ALGORITHM 1.1 (QFT Circuit). For n qubits labeled $q_0, \ldots, q_{n-1}$ (with q_0 the most significant bit):

1: **for** $j = 0$ to $n-1$ **do**
2: Apply H to q_j
3: **for** $k = j+1$ to $n-1$ **do**
4: Apply controlled-R_{k-j+1} from q_k to q_j, where $R_m = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^m} \end{pmatrix}$
5: **end for**
6: **end for**
7: Swap qubits to reverse order (depending on convention).

Circuit depth: $O(n^2)$, versus $O(n2^n)$ for a classical FFT.

Example 1.4 (QFT on 2 Qubits)**.** For $n = 2$, the QFT is

$$\mathrm{QFT}_2 = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{pmatrix}.$$

Applying QFT_2 to $|00\rangle$ gives

$$\mathrm{QFT}_2\,|00\rangle = \tfrac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle).$$

1.1. Phase Estimation. The QFT is a crucial ingredient in phase estimation, which estimates the eigenphase ϕ of a unitary operator U given an eigenvector $|u\rangle$ such that $U|u\rangle = e^{2\pi i\phi}|u\rangle$.

Quantum Algorithm 1.5 (Phase Estimation)**.** Input: unitary U, eigenvector $|u\rangle$, precision t qubits.

1: Prepare $|0\rangle^{\otimes t}|u\rangle$
2: Apply Hadamard gates: $\frac{1}{\sqrt{2^t}}\sum_{k=0}^{2^t-1}|k\rangle|u\rangle$
3: Apply controlled-U^k: $\frac{1}{\sqrt{2^t}}\sum_{k=0}^{2^t-1} e^{2\pi i\phi k}|k\rangle|u\rangle$
4: Apply inverse QFT on the first register

Output: an estimate of ϕ with error $O(2^{-t})$.

THEOREM 1.6 (Accuracy of Phase Estimation). *With probability at least* $1-\epsilon$*, the algorithm outputs* $\tilde{\phi}$ *such that*

$$|\tilde{\phi} - \phi| \le 2^{-t} + \tfrac{1}{\epsilon}2^{-2t}.$$

Remark 1.7. Phase estimation underlies order-finding in groups, which is the central step in Shor's algorithm [10].

QFT in Qiskit

```
from qiskit import QuantumCircuit
from qiskit.circuit.library import QFT

qc = QuantumCircuit(3)
qc.append(QFT(3), [0,1,2])
qc.draw('mpl')
```

This produces the QFT circuit for 3 qubits.

Exercises

7.1. Implement the QFT for $n = 3$ manually in Qiskit (without the library) and verify its action on $|001\rangle$ using statevector simulation.

7.2. Show that the QFT is its own inverse up to conjugation and bit-reversal.

2. Grover's Search Algorithm

Grover's algorithm achieves a quadratic speedup for unstructured search, finding a marked element in a database of size N with $O(\sqrt{N})$ queries, compared to $O(N)$ classically.

Definition 2.1 (Unstructured Search Problem)**.** Given an oracle O_f such that

$$O_f|x\rangle|b\rangle = |x\rangle|b \oplus f(x)\rangle,$$

where $f(x) = 1$ if x is marked and 0 otherwise, find x with $f(x) = 1$.

Quantum Algorithm 2.2 (Grover's Algorithm)**.** Input: $n = \log_2 N$ qubits, oracle O_f, M marked items.

1: Initialize $|\psi_0\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle$

2: Repeat $r \approx \frac{\pi}{4}\sqrt{N/M}$ times:

- Apply oracle O_f: phase-flip marked states
- Apply diffusion operator $D = H^{\otimes n}(2|0\rangle\langle 0|^{\otimes n} - I)H^{\otimes n}$

3: Measure the state; with high probability obtain a marked x.

THEOREM 2.3 (Optimality). *Grover's algorithm requires $O(\sqrt{N})$ queries, and any quantum algorithm for unstructured search requires $\Omega(\sqrt{N})$ queries.*

SKETCH. The state evolves in the 2D span of solutions and non-solutions. Each iteration rotates the state vector by angle $\theta \approx 2\sqrt{M/N}$. After $O(1/\theta)$ steps, the probability of observing a solution is maximized. Optimality follows from quantum query lower bounds [6]. □

Remark 2.4. For algebraic applications, Grover can accelerate tasks such as searching for minimal polynomials, roots in finite fields, or satisfying assignments to algebraic constraints.

Grover in Qiskit

```
from qiskit import QuantumCircuit
from qiskit.circuit.library import GroverOperator, Diagonal

oracle = Diagonal([-1 if i == 3 else 1 for i in range(8)])
    # mark 011
grover_op = GroverOperator(oracle)

qc = QuantumCircuit(3)
qc.h([0,1,2])
qc.append(grover_op, [0,1,2])
qc.measure_all()
```

Exercises

7.3. Derive the exact iteration count when $M = 1$: $r = \lfloor \frac{\pi}{4} \arcsin(1/\sqrt{N}) \rfloor$.

7.4. Apply Grover to find a root of a quadratic polynomial modulo p and analyze the query complexity.

3. Shor's Algorithm

Shor's algorithm factors an integer N in polynomial time on a quantum computer by reducing factoring to period-finding.

Quantum Algorithm 3.1 (Shor's Algorithm)**.** Input: Composite N.

1: Pick random $a \in [2, N-1]$ with $\gcd(a, N) = 1$
2: Use phase estimation on $U : |y\rangle \mapsto |a^y \bmod N\rangle$ to find order r of a modulo N
3: Use continued fractions to recover r from measured phase
4: If r is even, compute $\gcd(a^{r/2} \pm 1, N)$ for factors
5: Repeat if necessary

Runtime: $O((\log N)^3)$.

THEOREM 3.2 (Correctness of Period-Finding). *With high probability, phase estimation yields a convergent s/r to the eigenphase, from which the order r can be reconstructed.*

PROOF. Eigenvalues of U are $e^{2\pi ik/r}$, $k = 0, \ldots, r-1$. Phase estimation yields k/r, and continued fractions reconstruct r when the error is $O(1/r^2)$. □

Remark 3.3. Shor's algorithm also solves discrete logarithms: given $a, b \in \mathbb{F}_p$, find x such that $a^x \equiv b \pmod{p}$. More generally, it solves the hidden subgroup problem for abelian groups, connecting to number fields and elliptic curves.

Exercises

7.5. Simulate Shor's algorithm for $N = 15$, $a = 2$. Compute the order and deduce the factors.

7.6. Compare classical and quantum complexities of the discrete logarithm problem over $\mathbb{F}_p$.

CHAPTER 8

Hidden Subgroup Problem and Group-Theoretic Algorithms

The *Hidden Subgroup Problem* (HSP) provides a unifying framework for many of the most powerful quantum algorithms. Factoring, discrete logarithms, Pell's equation, and more generally, a variety of algebraic problems can be reformulated as instances of HSP. The central technique involves the application of the Quantum Fourier Transform (QFT) to states encoding cosets of hidden subgroups. In this chapter, we first present the abstract framework, then examine applications in number theory and algebraic extensions.

1. Hidden Subgroup Problem (HSP) Framework

The HSP is defined over a finite group G, with a subgroup $H \leq G$ hidden by a function $f : G \to S$, where S is some finite set.

Definition 1.1 (Hidden Subgroup Problem)**.** Let G be a finite group. A function $f : G \to S$ hides a subgroup $H \leq G$ if

$$f(x) = f(y) \iff xH = yH.$$

That is, f is constant and distinct on left cosets of H. The *Hidden Subgroup Problem* (HSP) asks to determine H given oracle access to f.

Example 1.2 (Classical Examples)**.**

- In $\mathbb{Z}/N\mathbb{Z}$, the function $f(x) = a^x \bmod N$ hides the subgroup generated by the order of a. This is the basis of Shor's algorithm for factoring.
- In $\mathbb{Z}$, the function $f(x) = a^x$ over $\mathbb{CC}^\times$ hides the subgroup $r\mathbb{Z}$, where r is the order. This corresponds to order finding.

The quantum algorithm for the abelian HSP proceeds as follows.

Quantum Algorithm 1.3 (Abelian HSP Algorithm [12])**.** Input: group G, oracle f hiding $H \leq G$.

1: Prepare uniform superposition over G: $\frac{1}{\sqrt{|G|}} \sum_{g \in G} |g\rangle|0\rangle$
2: Query oracle: $\frac{1}{\sqrt{|G|}} \sum_{g \in G} |g\rangle|f(g)\rangle$
3: Measure second register: leaves uniform superposition over a random coset gH
4: Apply QFT over G to first register
5: Measure: yields random character χ of G satisfying $\chi(h) = 1$ for all $h \in H$
6: Repeat to gather enough information to reconstruct H

THEOREM 1.4 (Correctness for Abelian HSP). *If G is finite abelian, the algorithm outputs a generating set for H with high probability in polynomially many samples.*

Remark 1.5. The abelian HSP reduces to lattice problems: the measured characters impose linear equations modulo group orders, solvable using linear algebra. For nonabelian groups, the problem becomes much harder and remains an active research area.

2. Applications to Number Theory

Several classical number-theoretic problems can be cast as instances of the HSP.

2.1. Order Finding and Factoring. Let $G = (\mathbb{Z}/N\mathbb{Z})^\times$. For $a \in G$, define $f(x) = a^x$. Then f hides the subgroup $r\mathbb{Z}$, where r is the order of a. Applying the abelian HSP algorithm yields r, which suffices for Shor's factoring algorithm.

Example 2.1 (Order Finding)**.** For $N = 15$, $a = 2$, the function $f(x) = 2^x \bmod 15$ hides the subgroup $4\mathbb{Z}$. The HSP algorithm recovers $r = 4$, leading to the factors $\gcd(2^{r/2} \pm 1, 15) = 3, 5$.

2.2. Pell's Equation. The Pell equation

$$x^2 - Dy^2 = 1, \quad D \notin \mathbb{Z}^2,$$

is related to the computation of the fundamental unit in $\mathbb{Z}[\sqrt{D}]$. The unit group is infinite cyclic, and computing its generator is an instance of order-finding in a suitable group embedding.

THEOREM 2.2 (Quantum Pell's Equation [3]). *Quantum algorithms for the HSP yield polynomial-time solutions for Pell's equation, in contrast to classical exponential algorithms.*

Remark 2.3. This shows that quantum algorithms penetrate into Diophantine analysis, providing efficient computation of units in quadratic fields.

3. Algebraic Extensions

The HSP generalizes naturally to algebraic number fields and their unit and ideal groups.

3.1. Unit Groups of Number Fields. Let K be a number field with ring of integers $\mathcal{O}_K$. Dirichlet's unit theorem states that the unit group

$$\mathcal{O}_K^\times \cong \mu_K \times \mathbb{Z}^{r+s-1},$$

where μ_K is the finite group of roots of unity and r, s are the real and complex embeddings. Computing a basis of the free part reduces to a hidden subgroup problem in an additive lattice.

Remark 3.1. Quantum algorithms can approximate the logarithmic embeddings of units using phase estimation and HSP methods, giving polynomial-time algorithms for problems that are classically hard.

3.2. Principal Ideal Problem. Given an ideal $I \subseteq \mathcal{O}_K$, determining whether I is principal and, if so, finding a generator, is a central problem in computational number theory. This problem can be formulated as an HSP in the class group $\mathrm{Cl}(K)$. Quantum algorithms based on HSP provide polynomial-time solutions in certain families of number fields.

3.3. Connections to Class Group Computations. The class group $\mathrm{Cl}(K)$ is a finite abelian group, and its structure can be determined by solving an HSP. In particular:

- The function $f : \mathbb{Z}^m \to \mathrm{Cl}(K)$, mapping integer vectors to ideals modulo principal ideals, hides the relation lattice among generators.
- Quantum algorithms reveal the subgroup corresponding to relations, thereby reconstructing $\mathrm{Cl}(K)$.

THEOREM 3.2 (van Dam–Hallgren–Ip [12]). *For certain number fields, the class group and unit group can be computed in quantum polynomial time via reductions to the abelian HSP.*

Remark 3.3. These results connect quantum computation with algebraic number theory and arithmetic geometry. They suggest potential breakthroughs in the computation of zeta functions, regulators, and invariants of arithmetic schemes.

Exercises

8.1. Formulate the discrete logarithm problem in $\mathbb{F}_p^\times$ as an HSP instance and outline the quantum solution.

8.2. Show how the HSP for $\mathbb{Z}$ with oracle $f(x) = a^x$ recovers the order of a modulo N.

8.3. Discuss how class group computations via HSP might impact cryptosystems based on ideal lattices.

CHAPTER 9

Quantum Linear Algebra

This chapter delves into quantum algorithms for core linear algebra tasks, such as solving linear systems and matrix decompositions, with applications to computational algebra. We emphasize how these algorithms leverage quantum parallelism and phase estimation to achieve exponential speedups for certain problems, while highlighting connections to classical algebraic structures like matrices over rings and Diophantine equations. The material builds on the quantum fundamentals and algorithms from previous chapters, drawing from Harrow et al. [7], Lipton and Regan [9], and van Dam and Hallgren [12].

1. Harrow-Hassidim-Lloyd (HHL) Algorithm

The Harrow-Hassidim-Lloyd (HHL) algorithm is a quantum method for solving linear systems of equations $A\mathbf{x} = \mathbf{b}$, where A is an $N \times N$ matrix and $\mathbf{b}$ is a vector. It provides an exponential speedup over classical methods for certain sparse, well-conditioned systems, outputting a quantum state encoding the solution.

Definition 1.1 (Linear System Problem)**.** Given a Hermitian matrix $A \in \mathbb{C}^{N\times N}$ (w.l.o.g., as non-Hermitian cases can be reduced), a vector $\mathbf{b} \in \mathbb{C}^N$, and precision ϵ, find $\mathbf{x} = A^{-1}\mathbf{b}$ up to error ϵ in the 2-norm.

Remark 1.2. Assumptions: A is sparse (O(poly log N) entries per row), well-conditioned (condition number $\kappa = \|A\|\|A^{-1}\| = O(\texttt{Poly}\log N)$), and $\|\mathbf{b}\| = 1$. The output is the quantum state $|\mathbf{x}\rangle$ proportional to $\mathbf{x}$.

The HHL algorithm uses phase estimation to diagonalize A in the quantum domain, invert eigenvalues, and reconstruct the solution.

Quantum Algorithm 1.3 (HHL Algorithm)**.** Input: Quantum oracles for A (e.g., Hamiltonian simulation e^{-iAt}) and state preparation of $|\mathbf{b}\rangle = \sum_i b_i|i\rangle$.

1: Prepare $|0\rangle^{\otimes t}|\mathbf{b}\rangle$, where $t = O(\log N + \log(1/\epsilon))$ ancilla qubits.
2: Apply phase estimation on A: estimate eigenvalues λ_j of A, yielding $\sum_j \beta_j|\tilde{\lambda}_j\rangle|u_j\rangle$, where $|\mathbf{b}\rangle = \sum_j \beta_j|u_j\rangle$ in eigenbasis $\{|u_j\rangle\}$.
3: Add ancilla and conditionally rotate to encode $1/\lambda_j$: $\sum_j \beta_j|\tilde{\lambda}_j\rangle|u_j\rangle|0\rangle \to \sum_j \beta_j|\tilde{\lambda}_j\rangle|u_j\rangle\left(\sqrt{1-\frac{C^2}{\tilde{\lambda}_j^2}}|0\rangle + \frac{C}{\tilde{\lambda}_j}|1$
where $C = O(1/\kappa)$.
4: Uncompute phase estimation (inverse QPE) conditioned on the ancilla being $|1\rangle$, yielding $|\mathbf{x}\rangle \approx \sum_j \frac{\beta_j}{\lambda_j}|u_j\rangle$.

Output: $|\mathbf{x}\rangle$ with error $O(\epsilon)$. Runtime: $O(\log N \cdot \kappa^2/\epsilon)$.

Theorem 1.4 (Correctness and Efficiency of HHL). *If A is s-sparse with $\kappa(A) = \kappa$, HHL solves $A\mathbf{x} = \mathbf{b}$ in time $O(\log N \cdot s^2\kappa^2/\epsilon)$, with success probability $\Omega(1/\kappa^2)$.*

Proof. Phase estimation approximates λ_j to $O(1/\kappa\epsilon)$ accuracy using $O(\kappa^2/\epsilon)$ applications of e^{-iAt} (via Hamiltonian simulation, costing $O(s^2 \log N)$ per step).

Eigenvalue inversion via rotation introduces $O(\epsilon)$ error, amplified by κ. Uncomputing projects to the inverted subspace. Probability scaling from min eigenvalue $1/\kappa$; repeat $O(\kappa^2)$ times with amplitude amplification for constant success. □

Remark 1.5. HHL exponentially faster than classical $O(N)$ for dense matrices, but output is quantum—extracting classical $\mathbf{x}$ requires $O(N)$ measurements. Useful for expectation values $\langle \mathbf{x}|M|\mathbf{x}\rangle$.

Simple HHL in Qiskit

```
from qiskit import QuantumCircuit
from qiskit.algorithms.linear_solvers.hhl import HHL
from qiskit.quantum_info import Statevector
import numpy as np

matrix = np.array([[1, -1/3], [-1/3, 1]])
vector = np.array([1, 0])
naive_hhl_solution = HHL().solve(matrix, vector)
print(naive_hhl_solution.state)  # Circuit for solution
    state
```

This solves a 2x2 system; simulate for the quantum state encoding $\mathbf{x}$.

Exercises

9.1. Derive the error bound for eigenvalue inversion: show that if $|\tilde{\lambda} - \lambda| < \delta$, then $|1/\tilde{\lambda} - 1/\lambda| < O(\delta/\lambda^2)$.

9.2. Discuss adapting HHL for non-Hermitian A by solving the extended system with $\begin{pmatrix} 0 & A \\ A^\dagger & 0 \end{pmatrix}$.

2. Quantum Singular Value Decomposition (QSVD)

Quantum Singular Value Decomposition (QSVD) extends classical SVD to quantum settings, decomposing a matrix into singular values and vectors. It enables efficient manipulation of singular values via quantum algorithms, with applications to pseudoinverses and matrix norms.

Definition 2.1 (Singular Value Decomposition)**.** For $A \in \mathbb{C}^{m\times n}$, the SVD is $A = U\Sigma V^\dagger$, where U, V are unitary, $\Sigma = \mathrm{diag}(\sigma_1, \ldots, \sigma_r)$ with singular values $\sigma_i \geq 0$.

In quantum computing, QSVD often refers to algorithms that estimate or transform singular values, such as Quantum Singular Value Transformation (QSVT), a framework for applying polynomials to singular values.

Definition 2.2 (Quantum Singular Value Transformation)**.** Given block-encoded access to A (a unitary U with A as top-left block), QSVT applies a polynomial p to the singular values: output state encodes $p(A)|\psi\rangle/\|p(A)|\psi\rangle\|$.

A variational approach (VQSVD) uses hybrid quantum-classical optimization to find singular vectors.

Quantum Algorithm 2.3 (Variational QSVD)**.** Input: Matrix A via quantum access.

1: Parameterize ansatze $U(\theta), V(\phi)$ for left/right singular vectors.
2: Minimize cost $C(\theta, \phi) = \|A - U(\theta)\Sigma V(\phi)^\dagger\|^2$ via classical optimizer, measuring overlaps quantumly.

3: Output approximated singular values from diagonal Σ.

Proposition 2.4 (Efficiency of QSVT)**.** *For degree-d polynomial p, QSVT uses $O(d)$ queries to the block-encoding of A, achieving transformations like pseudoinverse ($p(x) = 1/x$ approximated).*

PROOF. QSVT interleaves projections and phase factors to polynomial-approximate functions on singular values, with query complexity linear in degree [9]. □

Applications include computing matrix norms ($\max \sigma_i$) and pseudoinverses for least squares ($A^+ = V\Sigma^+U^\dagger$, where $\Sigma^+ = \text{diag}(1/\sigma_i)$ for $\sigma_i > 0$).

Remark 2.5. QSVD enables quantum speedups for principal component analysis in algebraic data analysis, but requires efficient matrix encoding.

Simple SVD Simulation in Qiskit (Classical Wrapper)

```
from qiskit.quantum_info import random_unitary
import numpy as np
from numpy.linalg import svd

# Simulate quantum-inspired: random matrix
A = np.random.rand(4,4)
U, S, Vh = svd(A)
print("Singular values:", S)
# Quantum version would use phase estimation on A A^\dagger
```

For full quantum, use experimental circuits for small systems.

Exercises

9.3. Prove that singular values of A are square roots of eigenvalues of $AA^\dagger$.

9.4. Discuss approximating $1/x$ polynomial for pseudoinverse in QSVT, bounding degree for precision ϵ.

3. Connections to Classical Algebra

Quantum linear algebra algorithms like HHL and QSVD connect deeply to classical algebraic computations, extending to matrices over rings and solving systems like Diophantine equations, though with limitations.

Remark 3.1 (Adapting HHL for Rings)**.** For matrices over rings like $\mathbb{Z}$ or $\mathbb{Q}$, embed into $\mathbb{C}$ via modular techniques or use quantum algorithms for lattice problems (e.g., shortest vector approximating Diophantine solutions).

Proposition 3.2 (Quantum Solving of Diophantine Systems)**.** *HHL can approximate solutions to $A\mathbf{x} = \mathbf{b}$ over $\mathbb{R}$, extendable to integer solutions via rounding, but exact Diophantine requires handling ill-conditioning.*

PROOF. For integer $A, \mathbf{b}$, normalize and solve in quantum, measure to sample near-integers, verify classically. Limitations: exponential bits for exactness [12]. □

Key limitations: - Sparsity: Oracles assume O(poly log N) access. - Hermiticity: Required for phase estimation; non-Hermitian needs doubling dimension. - Condition number: Runtime quadratic in κ. - Output readout: Quantum state, not classical vector.

Remark 3.3. In cryptography, HHL threatens lattice-based schemes by approximating short vectors. For algebraic geometry, accelerates solving polynomial systems via resultant matrices.

Exercises

9.5. Adapt HHL for a 2x2 integer matrix; discuss quantum vs classical Gaussian elimination.

9.6. Explore limitations: show that for $\kappa = \Theta(N)$, HHL no better than classical.

CHAPTER 10

Lattice Reduction and Quantum Attacks

Lattices play a central role in computational number theory, cryptography, and optimization. Classical algorithms such as LLL provide polynomial-time approximation to hard problems like shortest vectors. Lattice-based cryptography, including schemes based on Learning with Errors (LWE) and NTRU, derives its security from the conjectured hardness of such lattice problems. Quantum algorithms, although not known to fully break these systems (unlike Shor's algorithm for RSA), offer partial improvements and hybrid attacks. This chapter surveys classical lattice reduction, the LLL algorithm, and known quantum approaches.

1. Classical Lattice Reduction

Definition 1.1 (Lattice)**.** A *lattice* $L \subseteq \mathbb{R}^n$ is the set of all integer linear combinations of linearly independent vectors $b_1, \ldots, b_m \in \mathbb{R}^n$:

$$L = \mathbb{Z}b_1 + \cdots + \mathbb{Z}b_m = \{\sum_{i=1}^{m} z_i b_i \mid z_i \in \mathbb{Z}\}.$$

The set $\{b_1, \ldots, b_m\}$ is called a *basis* of L.

Different bases can generate the same lattice, but may differ drastically in geometric properties. For instance, the vectors might be long and nearly parallel, or short and nearly orthogonal.

Definition 1.2 (Gram-Schmidt Orthogonalization)**.** Given a basis $b_1, \ldots, b_m$, the Gram–Schmidt process produces orthogonal vectors $b_1^*, \ldots, b_m^*$ defined recursively by

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*, \quad \mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{\langle b_j^*, b_j^* \rangle}.$$

Remark 1.3. The Gram–Schmidt vectors b_i^* provide geometric insight into the lattice and are fundamental to reduction algorithms. However, they do not themselves form a basis of L unless the original basis is orthogonal.

1.1. Hard Lattice Problems.

Two central computational problems are:

- **Shortest Vector Problem (SVP)**: Given a lattice L, find a nonzero $v \in L$ of minimum norm.
- **Closest Vector Problem (CVP)**: Given L and $t \in \mathbb{R}^n$, find $v \in L$ minimizing $\|t - v\|$.

Both are NP-hard in general, but efficient approximation algorithms exist.

2. LLL Algorithm

The Lenstra–Lenstra–Lovász (LLL) algorithm [3] provides a polynomial-time method for producing a reduced lattice basis with guaranteed quality bounds.

Definition 2.1 (LLL-Reduced Basis)**.** Let $\delta \in (1/4, 1)$. A basis $b_1, \ldots, b_m$ with Gram–Schmidt vectors b_i^* is *LLL-reduced* if:

(1) Size reduction: $|\mu_{i,j}| \leq 1/2$ for $j < i$.
(2) Lovász condition: $\delta\|b_{i-1}^*\|^2 \leq \|b_i^*\|^2 + \mu_{i,i-1}^2\|b_{i-1}^*\|^2$ for $2 \leq i \leq m$.

THEOREM 2.2 (LLL Approximation). *Given an m-dimensional lattice $L \subseteq \mathbb{R}^n$, the LLL algorithm outputs a reduced basis in time polynomial in n and $\log \|b_i\|$. The first vector b_1 satisfies*

$$\|b_1\| \leq 2^{(m-1)/2} \cdot \lambda_1(L),$$

where $\lambda_1(L)$ is the length of the shortest nonzero vector in L.

Remark 2.3. The factor $2^{(m-1)/2}$ is exponential in dimension, but for moderate ranks it provides good practical approximations. Many applications rely on this efficiency despite worst-case hardness.

2.1. Applications of LLL.

- **Integer Relations:** Given reals $\alpha_1, \ldots, \alpha_n$, LLL finds integer vectors z with $\sum z_i\alpha_i \approx 0$, enabling detection of algebraic dependencies.
- **Knapsack Problems:** Certain subset-sum instances can be reformulated as SVP in lattices and solved using LLL.
- **Cryptanalysis:** Early knapsack-based cryptosystems were broken using LLL reductions.

3. Quantum Algorithms for Lattices

Quantum computing has not yet yielded algorithms that solve SVP or CVP in polynomial time. However, several approaches yield partial improvements or hybrid attacks.

3.1. Quantum Walks for SVP.

Remark 3.1. Quantum walks provide quadratic speedups over classical random walks for certain search spaces. Applied to sieving algorithms for SVP, they reduce the running time.

THEOREM 3.2 (Quantum Sieving Speedup). *For n-dimensional SVP, classical sieving runs in time $2^{0.292n+o(n)}$. Quantum walk methods reduce this to $2^{0.265n+o(n)}$.*

3.2. Grover-Accelerated Enumeration. Enumeration methods for SVP and CVP can be accelerated quadratically using Grover's search algorithm. If classical enumeration requires $O(N)$ steps, the quantum hybrid achieves $O(\sqrt{N})$.

Example 3.3 (Grover Hybrid for SVP)**.** Given a reduced basis, search for short lattice vectors among $O(N)$ candidates. Grover reduces this to $O(\sqrt{N})$, yielding exponential savings in practice.

3.3. Quantum Attacks on Lattice-Based Cryptography.

- **NTRU:** Reduction attacks using lattice basis embeddings are sped up by Grover-type search.
- **LWE:** Learning with Errors is reducible to worst-case lattice problems. Known quantum algorithms do not fully break LWE, but provide speedups in specific parameter regimes.
- **Hybrid Attacks:** Combine classical lattice reduction (LLL, BKZ) with quantum sieving or Grover acceleration to weaken cryptographic parameters.

Remark 3.4. Unlike RSA and elliptic curve cryptography, lattice-based systems remain resistant to *known* quantum algorithms. This is why they are leading candidates for post-quantum cryptography.

Exercises

10.1. Use Gram–Schmidt orthogonalization to show that the LLL reduction produces nearly orthogonal basis vectors.

10.2. Apply LLL to the lattice generated by $(105, 0)$ and $(13, 1)$ and recover an integer relation between 105 and 13.

10.3. Discuss why SVP admits only subexponential-time quantum algorithms, in contrast to factoring which becomes polynomial-time under Shor's algorithm.

CHAPTER 11

Advanced Quantum Algebraic Algorithms

This chapter explores advanced quantum algorithms that integrate deeply with algebraic structures, pushing the boundaries of computational algebra into quantum-enhanced domains. We examine quantum walks for graph-based problems, quantum methods for tackling nonlinear polynomial systems, and recent developments including quantum-inspired classical techniques and hybrid solvers. These topics highlight emerging intersections between quantum computing and algebra, such as optimization over varieties and group-theoretic computations. The material draws from Nielsen and Chuang [10] and van Dam et al. [12], supplemented by cutting-edge advancements as of 2025.

1. Quantum Walks on Graphs

Quantum walks generalize classical random walks to quantum settings, exploiting superposition and interference for faster mixing and hitting times on graphs. They come in discrete and continuous variants and have applications in graph isomorphism testing, element distinctness, and algebraic problems like solving systems via Cayley graphs.

Definition 1.1 (Discrete Quantum Walk)**.** On a graph $G = (V, E)$ with $|V| = N$, a discrete quantum walk uses a Hilbert space $\mathcal{H} = \mathcal{H}_C \otimes \mathcal{H}_V$, where $\mathcal{H}_C$ is the coin space (dimension $d = \deg(G)$) and $\mathcal{H}_V = \mathbb{C}^N$ for vertices. The walk operator is $W = S(C \otimes I)$, where C is a coin flip unitary (e.g., Grover coin $C = 2|s\rangle\langle s| - I$, $|s\rangle = \frac{1}{\sqrt{d}} \sum |i\rangle$), and S is the shift: $S|c\rangle|v\rangle = |c\rangle|v + c\rangle$ (mod graph).

Definition 1.2 (Continuous Quantum Walk)**.** A continuous quantum walk evolves under the Schrödinger equation $i\frac{d}{dt}|\psi(t)\rangle = H|\psi(t)\rangle$, where $H = -\gamma A$ is the Hamiltonian, A the adjacency matrix of G, and γ a jumping rate. The state at time t is $|\psi(t)\rangle = e^{-iHt}|\psi(0)\rangle$.

Proposition 1.3 (Unitarity of Quantum Walks)**.** *Both discrete (W) and continuous (e^{-iHt}) walks are unitary evolutions, preserving norms.*

Proof. For discrete: C unitary implies $C \otimes I$ unitary; S is a permutation matrix, hence unitary; product unitary. For continuous: H Hermitian (A symmetric), so e^{-iHt} unitary by Stone's theorem. □

Quantum walks achieve quadratic speedups over classical walks for hitting times.

Theorem 1.4 (Speedup in Hitting Time). *For certain graphs (e.g., hypercube), quantum walks hit a marked vertex in $O(\sqrt{N})$ time, vs. classical $O(N)$.*

Proof. Interference amplifies amplitude at targets; analyzed via spectral decomposition of walk operator [10]. □

Applications include graph isomorphism: Evolve walks on two graphs; if evolution differs, graphs non-isomorphic. In algebra, walks on Cayley graphs solve group problems like hidden shifts.

Quantum Algorithm 1.5 (Quantum Walk Search)**.** Input: Graph G, marked vertex oracle.

1: Initialize uniform superposition $|s\rangle = \frac{1}{\sqrt{N}} \sum_v |v\rangle$.
2: For $O(\sqrt{N})$ steps: Apply walk operator interspersed with oracle phase flips on marked vertices.
3: Measure to find marked vertex.

Remark 1.6. Quantum walks connect to algebraic spectral graph theory, using eigenvalues of Laplacians for quantum simulation.

Quantum Walk on Line in Qiskit

```
from qiskit import QuantumCircuit, QuantumRegister
from qiskit.circuit.library import PhaseGate

# Simple 4-vertex line: position qubits + coin
pos = QuantumRegister(2, 'pos')
coin = QuantumRegister(1, 'coin')
qc = QuantumCircuit(pos, coin)
qc.h(pos)  # Superposition positions
qc.h(coin) # Coin flip
# Shift and coin operations...
```

Simulate for interference patterns.

Exercises

11.1. Prove that the Grover coin is unitary and compute its action on a balanced state.

11.2. Discuss using quantum walks for testing if two polynomials generate the same ideal via graph representations.

2. Nonlinear Polynomial Systems

Solving nonlinear polynomial systems is central to algebraic geometry and computationally hard (NP-hard in general). Quantum algorithms approach this via optimization: reformulate as finding minima of objective functions, using adiabatic quantum computing (AQC) or variational methods.

Definition 2.1 (Nonlinear Polynomial System)**.** Given polynomials $f_1, \ldots, f_m \in \mathbb{C}[x_1, \ldots, x_n]$, find $\mathbf{x}$ such that $f_i(\mathbf{x}) = 0$ for all i, or determine if solutions exist.

Quantum methods map to optimization: Minimize $\sum_i |f_i(\mathbf{x})|^2$ or use Groebner basis as constraints.

Quantum Algorithm 2.2 (Adiabatic Quantum Computing for Optimization)**.** Input: Hamiltonian H_P encoding the problem (e.g., $H_P = \sum_i f_i^\dagger f_i$ as operators).

1: Start with ground state of easy Hamiltonian H_0 (e.g., transverse field $\sum X_i$).
2: Evolve adiabatically: $H(t) = (1 - t/T)H_0 + (t/T)H_P$ for time T.
3: Measure final state for approximate ground state (solution).

Runtime: $T = O(1/\Delta^2)$, where Δ is min spectral gap.

THEOREM 2.3 (Adiabatic Theorem). *If $T \gg 1/\Delta_{\min}^2$, the final state is close to the ground state of H_P.*

PROOF. By adiabatic approximation: Slow evolution keeps system in instantaneous ground state if gaps positive [12]. □

For high-degree equations, QAOA (Quantum Approximate Optimization Algorithm) approximates solutions by variational circuits.

Remark 2.4. Limitations: Exponential time for worst-case gaps; useful for heuristic speedups in algebraic cryptanalysis or variety sampling.

Exercises

11.3. Reformulate solving $x^2 + y^2 = 1$ as a quantum optimization problem.

11.4. Compare AQC gap for quadratic vs cubic systems.

3. Recent Developments

Recent years have seen rapid progress in quantum algebraic algorithms, driven by hardware advances and theoretical breakthroughs. As of 2025, key trends include quantum-inspired classical algorithms that dequantize certain speedups and hybrid quantum-classical solvers tailored for commutative algebra tasks like ideal membership and variety decomposition.

Quantum-inspired algorithms use classical sampling techniques to mimic quantum advantages without qubits. For instance, dequantized versions of recommendation systems or low-rank approximations apply to algebraic tensor decompositions, achieving polynomial-time approximations where full quantum might be exponential.

Remark 3.1. A 2025 study demonstrated unconditional exponential quantum scaling advantage for certain algebraic problems, leveraging improved phase estimation [7].

Hybrid solvers combine quantum circuits for hard subroutines (e.g., eigenvalue estimation) with classical symbolic computation. For commutative algebra, variational quantum eigensolvers (VQE) compute ground states of Hamiltonians encoding polynomial ideals, aiding in solving systems over finite fields.

Proposition 3.2 (Hybrid Groebner Basis)**.** *A hybrid algorithm interleaves quantum search (Grover) for S-pairs with classical reduction, reducing exponential branching.*

PROOF. Quantum selects critical pairs faster; classical verifies. Complexity drops from $O(d^{2^v})$ to $O(\sqrt{d^{2^v}})$ in favorable cases. □

Notable 2025 advancements: - Distributed quantum algorithms for multi-processor algebraic computations, enabling scalable factoring beyond single devices [11]. - Quantum frameworks for analyzing complex algebraic networks, applying walks to ideal lattices. - Special issues on quantum logic and programming, intersecting with algebraic semantics (QPL 2024/2025).

Remark 3.3. Optimism persists for quantum optimization in algebra, with arXiv previews suggesting breakthroughs in 2025 [12]. Challenges include error correction and dequantization threats.

Exercises

11.5. Survey a 2025 paper on distributed quantum algebra and summarize its algebraic application.

11.6. Propose a hybrid solver for ideal membership using VQE.

CHAPTER 12

Applications and Quantum Algebraic Geometry

1. Quantum Error Correction and Codes

Algebraic geometric codes; Reed-Solomon on quantum channels; stabilizer formalism [10].

2. Cryptography Implications

Breaking RSA with Shor; post-quantum schemes (lattices, multivariate polynomials); algebraic attacks [11].

Bibliography

[1] E. R. Berlekamp, Factoring polynomials over finite fields, *Bell System Technical Journal*, 46(8):1853–1859, 1967.

[2] B. Buchberger, Gr"obner Bases: An Algorithmic Method in Polynomial Ideal Theory, in *Multidimensional Systems Theory*, pages 184–232, Reidel Publishing Company, 1985.

[3] H. Cohen, *A Course in Computational Algebraic Number Theory*, Graduate Texts in Mathematics, Springer, 1993.

[4] D. A. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*, Undergraduate Texts in Mathematics, Springer, 4th edition, 2015.

[5] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, 3rd edition, 2013.

[6] L. K. Grover, A fast quantum mechanical algorithm for database search, in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 212–219, 1996.

[7] A. W. Harrow, A. Hassidim, and S. Lloyd, Quantum algorithm for linear systems of equations, *Physical Review Letters*, 103(15):150502, 2009.

[8] M. Kreuzer and L. Robbiano, *Computational Commutative Algebra 1*, Springer, 2000.

[9] R. J. Lipton and K. W. Regan, *Quantum Algorithms via Linear Algebra: A Primer*, MIT Press, 2014.

[10] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 10th anniversary edition, 2010.

[11] P. W. Shor, Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[12] W. van Dam and S. Hallgren, Quantum algorithms for algebraic problems, *Reviews of Modern Physics*, 82:1–47, 2010.