# A Java API for Polynomial Arithmetic

Claire Whelan Adam Duffy Andrew Burnett Tom Dowling

NUI Maynooth
Co. Kildare
Ireland
tdowling@cs.may.ie

## ABSTRACT

This paper looks at the development of a Java application programming interface (API) for performing unbounded Polynomial Arithmetic. It shows how Java can be used to perform large integer mathematical operations by using the `BigInteger` class. By demonstrating how the API might be used in a real application the paper shows how easy the API is to use with very little knowledge of polynomial arithmetic.

## Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software; D.3.2 [**Programming Languages**]: Language Classification—*JAVA*

## General Terms

Performance, Languages, Algorithms

## Keywords

Polynomial, arithmetic, java, api

## 1. INTRODUCTION

Polynomials and polynomial arithmetic play an important role in many areas of mathematics. Two such areas are cryptography and primality testing. This paper will demonstrate the suitability of Java for implementing computationally intensive polynomial based algorithms. This will be achieved by the development of an easy to use, efficient API for performing arithmetic operations on unbounded polynomials.

The paper is organised as follows. In section 2 the basic operations of polynomial arithmetic are described. Section 3 focuses on the API design. The implementation details are discussed in section 4. Section 5 demonstrates how the API is used in a practical application. Finally, conclusions and future work are mentioned in section 6.


*PPPJ* '03 Kilkenny City, Ireland


## 2. POLYNOMIAL ARITHMETIC

A polynomial over a set, $S$, is an expression of the form

$$u(x) = u_n x^n + \cdots + u_1 x + u_0$$

where the coefficients $u_n, \ldots, u_1, u_0$ and $x$ are elements of $S$. It is assumed that $S$ is a commutative ring with identity. Each term $u_i x^i$ is known as a monomial. The degree of $u(x)$, $deg(u)$, is the value of the largest exponent with non-zero coefficient in the polynomial $u(x)$. For the purposes of this implementation only polynomials in one variable, are considered. The leading term of a polynomial is the first monomial with a non-zero coefficient and the highest exponent value. A polynomial is considered to be *monic* if the leading term has a coefficient of one.

The basic polynomial operations are addition, subtraction, multiplication, division, exponentiation and greatest common divisor [5]. The processes of addition, subtraction, division and multiplication are defined in the usual way.

### 2.1 Division

Given two polynomials, $u(x)$ and $v(x)$, over a field, with $v(x) \neq 0$, we can divide $u(x)$ by $v(x)$ to obtain a quotient polynomial $q(x)$ and a remainder polynomial $r(x)$ which satisfies the condition

$$u(x) = q(x)v(x) + r(x)$$

where $deg(r) < deg(v)$.

#### 2.1.1 Polynomial Modulus

The polynomial modulus, $u(x)$ mod $v(x)$, can be defined as the remainder, $r(x)$, after division of $u(x)$ by $v(x)$.

### 2.2 Polynomial Inverse

Let $u(x)$ be some polynomial. Then the inverse is $v(x)$ such that

$$u(x).v(x) = 1x^0$$

If $u(x)$ is defined over the ring of integers this inverse may not always exist, however if it is defined over an integer field the inverse will always exist. For an account of ring and field theory see [6].

### 2.3 Greatest Common Divisor

The algorithm for division can be extended to an algorithm that computes greatest common divisors (GCD).

$$gcd(u(x), v(x)) = \begin{cases} u(x) & \text{if } v(x) = 0; \\ gcd(v(x), r(x)) & \text{otherwise.} \end{cases}$$

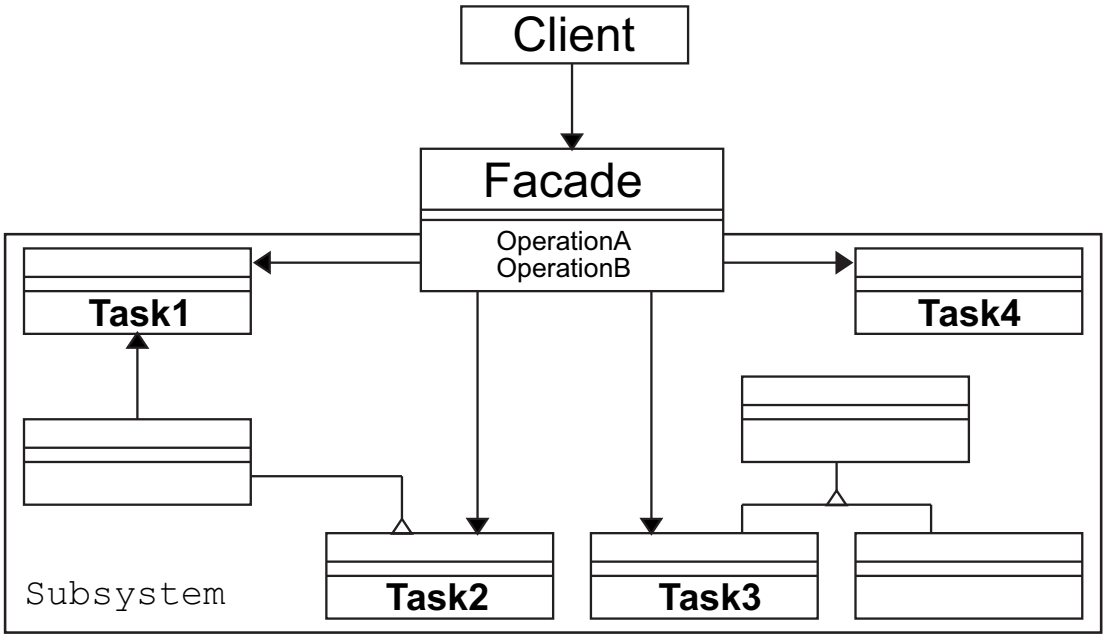


**Figure 1: Facade Design Pattern**

where $r(x)$ is obtained using the modulus algorithm above. We may write any nonzero polynomial $u(x)$ as

$$u(x) = cont(u).pp(u(x))$$

where $cont(u)$, the content of $u(x)$, is an element of $S$, and $pp(u(x))$, the primitive part of $u(x)$ is a primitive polynomial over $S$. The content of $u(x)$ is defined as the GCD of all coefficients in $u(x)$. The primitive part of a polynomial is obtained by dividing the polynomial by the greatest common divisor of its coefficients. A primitive polynomial is a polynomial which generates all elements of an extension field from a base field. When $u(x) = 0$, then $cont(u) = pp(u(x)) = 0$. The problem of finding greatest common divisors of arbitrary polynomials can be reduced to the problem of finding greatest common divisors of primitive polynomials.

$$\begin{aligned} cont(gcd(u,v)) &= a.gcd(cont(u), cont(v)) \\ pp(gcd(u(x), v(x))) &= b.gcd(pp(u(x)), pp(v(x))) \end{aligned}$$

where a and b are units. See [5] for details.

## 2.4 Exponentiation

Given a polynomial, $u$, and an integer, $i$, the exponentiation, $e(x)$ of that polynomial is

$$e(x) = u(x)^i = u(x).u(x)\ldots u(x)$$

To speed up the exponentiation method a standard power laddering algorithm was used. Multiplications were speeded up by implementing a binary segmentation algorithm, [3].

## 2.5 Polynomials over Fields

Algorithms for polynomials over rings, in some cases, contain restrictions on the type of polynomials that can be used. For example some modulus algorithms require that the modulus polynomial be monic. In order to overcome these restrictions the polynomial is defined over a field. This then allows us to be guaranteed of finding the inverse of the leading coefficient modulo the field size. For example $4x^2 + 3$ is not monic, but if it was defined over the field of integers mod 7 we could rewrite it as $x^2 + 6 \pmod 7$ because $4^{-1} = 2 \pmod 7$. Which is obviously monic. To convert a polynomial over a ring to a polynomial over a field all that is required is to mod the coefficients by the order of the field. To clarify, where $u(x)$ is a polynomial over the integers

$$u(x) = u_n x^n + u_{n-1}x^{n-1} + \cdots + u_0 x^0$$

where $u_n, \ldots, u_0 \in \mathbb{Z}$. The corresponding polynomial $u(x)_p$ over the field $\mathbb{Z}_p$ is given by

$$u(x)_p = (u_n \bmod p)x^n + (u_{n-1} \bmod p)x^{n-1} + \cdots + (u_0 \bmod p)x^0$$

where $u_i \in [0, p-1]$.

# 3. DESIGN CONSIDERATIONS

The purpose of the `BigPolynomial` API is to allow client objects to perform polynomial arithmetic without knowledge of the underlying mathematics. In order to satisfy this requirement, the *Facade* design pattern was used [4]. Another requirement in the development of the `BigPolynomial` API was to allow client objects to specify polynomials with an arbitrary number of monomial terms. This necessitated the use of an AVL tree data structure to get over the `int` size restriction on `array` and `Hashtable` data structures in Java and to enable $O(\log_2 n)$ searching.

## 3.1 Facade Design Pattern

The purpose of the Facade design pattern is to provide a simplified interface to the overall functionality of a complex subsystem, figure 1. This shields client objects from subsystem components, making the system easier to use. It also promotes weak coupling between the subsystem and it's clients. This allows the subsystems to change without affecting the clients [4].

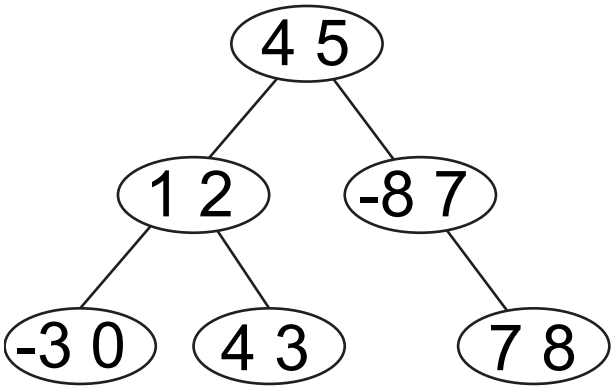


**Figure 2: AVL Tree representing the polynomial** $7x^8 - 8x^7 + 4x^5 + 4x^3 + x^2 - 3$

### 3.2 AVL Tree

Each node in the AVL tree contains one monomial. The monomials are sorted according to the value of their exponents with the lowest value in the leftmost node and the highest value in the rightmost node. The value of the coefficients has no bearing on the position of the monomial in the tree. In order to reduce space requirements, monomials with zero coefficients are not stored in the tree. An example of what such a tree might look like is shown above in figure 2.

## 4. BIGPOLYNOMIAL API

This section examines the main classes and methods in the BigPolynomial API. The basic polynomial class is `Polynomial`. This class allows you to create polynomials with `long` coefficients and exponents. It provides methods to calculate greatest common divisor, inverse and polynomial modulus amongst others. It is however limited by size, within range of `long` primitive, and also in the restrictions on the type of polynomials that can be used for some of the methods. This is because the polynomials are defined over the ring of integers. For example in the `mod()` method the polynomial that is used as the modulus must be monic.

A more practical class is `BigPolynomial`. It uses an AVL tree data structure to allow very large polynomials to be created with `BigInteger` coefficients and exponents. This class implements the same basic methods as `Polynomial` and also suffers from the same limitations on the type of polynomials that can be used. An example of how to multiply two `BigPolynomial`'s follows;

```
BigPolynomial u = new BigPolynomial();
u.put( new BigMonomial( "5", "3" ) );
u.put( new BigMonomial( "2", "1" ) );
u.put( new BigMonomial( "7", "0" ) );

BigPolynomial v = new BigPolynomial();
v.put( new BigMonomial( "8", "2" ) );
v.put( new BigMonomial( "3", "1" ) );

BigPolynomial r = u.multiply( v );
```

The above code is equivalent to $5x^3 + 2x + 7$ multiplied by $8x^2 + 3x$ which would produce a result of $40x^5 + 15x^4 + 16x^3 + 62x^2 + 21x$.

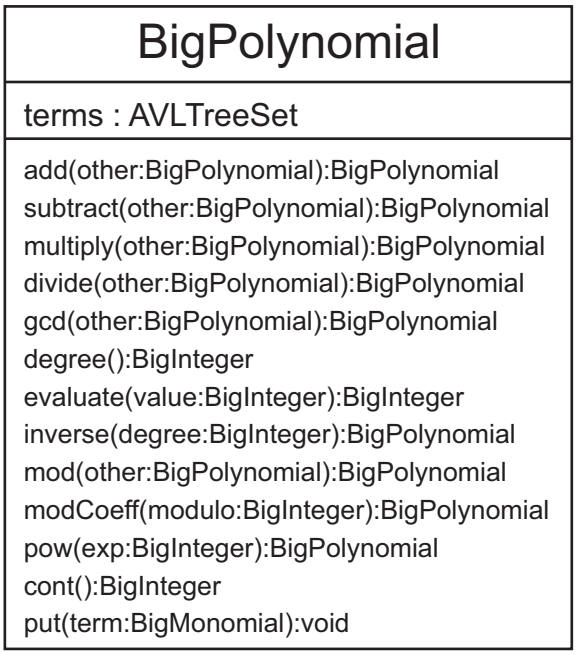


**Figure 3:** `BigPolynomial` **class diagram**

Addition, subtraction and division can be performed in a similar way. It is also possible to evaluate a polynomial for some integer.

```
BigPolynomial s = new BigPolynomial();
s.put(new BigMonomial("5", "6"));
s.put(new BigMonomial("3", "3"));
s.put(new BigMonomial("-8", "2"));
s.put(new BigMonomial("12", "0"));

BigInteger t = new BigInteger("7");

BigInteger result = s.evaluate(t);
```

Which is equivalent to $s(x) = 5x^6 + 3x^3 - 8x^2 + 12$ and $s(7) = 588894$. The GCD of two polynomials can be computed in the expected way:

```
BigPolynomial u = new BigPolynomial();
u.put(...
BigPolynomial v = new BigPolynomial();
v.put(...

BigPolynomial gcd = u.gcd(v);
```

One of the most useful methods in this class is the `mod()` method as it is used in a lot of polynomial based applications. It takes a `BigPolynomial` as an argument and mods another `BigPolynomial` by it. Another useful method along the lines of `mod()` is `modCoeff()` which takes a `BigInteger` as an argument and mods the coefficients of the polynomial by it. An example of how these methods might be used follows.

```
BigPolynomial u = new BigPolynomial();
u.put(new BigMonomial("5", "6"));
u.put(new BigMonomial("13", "4"));
u.put(new BigMonomial("9", "1"));
```

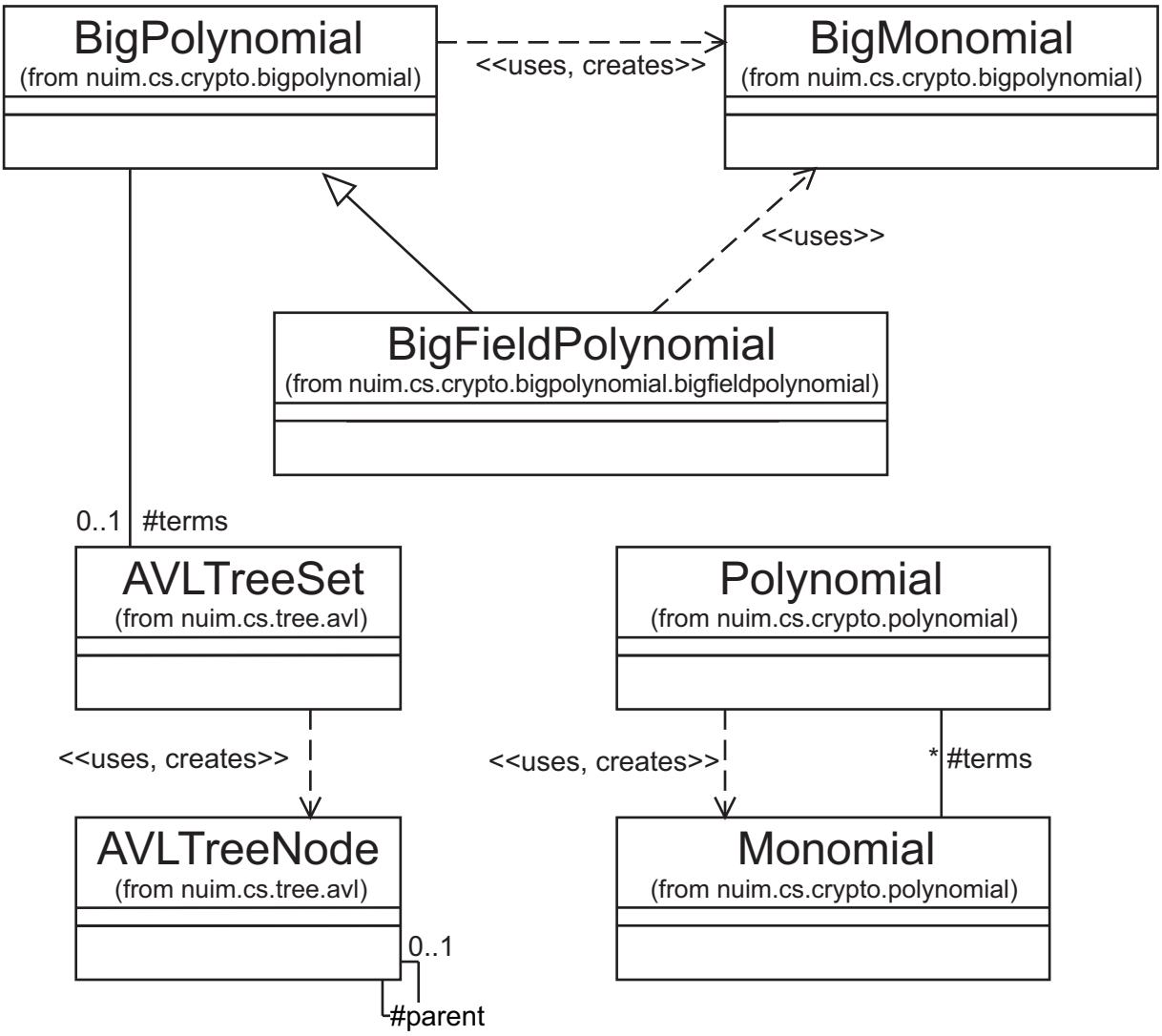


**Figure 4: UML diagram of main API classes**

```
u.put(new BigMonomial("12", "0"));

BigPolynomial v = new BigPolynomial();
v.put(new BigMonomial("1", "4"));
v.put(new BigMonomial("7", "3"));
v.put(new BigMonomial("5", "1"));

BigPolynomial s = u.mod(v);

BigInteger m = new BigInteger("8");

BigPolynomial t = u.modCoeff(m);
```

The above is equivalent to $u(x) = 5x^6+13x^4+9x+12, v(x) = x^4+7x^3+5x$ and $s(x) = -1831x^3+175x^2-1281x+12, t(x) = 5x^6 + 5x^4 + x + 4$

To overcome the limitations in these classes a `BigFieldPolynomial` class was developed. It extends `BigPolynomial` and adds the restriction that the polynomial coefficients be defined over some integer field. Same methods as `BigPolynomial` plus some extra exponentiation methods. The `modPow()` method takes a `BigInteger` as an argument, there is also an overloaded version of the method which takes another argument as a `BigPolynomial`. The arguments of the `modPow()` method are used to reduce the polynomial in between the steps of the power laddering algorithm used to preform the exponentiation. An example of how `modPow()` might be used is shown below.

```
BigFieldPolynomial bp =
    new BigFieldPolynomial(new BigInteger("7"));
bp.put(new BigMonomial("1", "6"));
bp.put(new BigMonomial("6", "4"));
bp.put(new BigMonomial("8", "0"));
BigFieldPolynomial result =
    bp.modPow(new BigInteger("5"));
```

This is equivalent to raising $x^6 + 6x^4 + 8$ to the power of 5 (mod 7), resulting in $x^{30} + 2^{28} + 3x^{26} + 6x^{22} + x^{20} + 4x^{18} + 3x^{16} + 2x^{19} + x^{10} + 3x^8 + 5x^6 + 2x^4 + 1$.

Documentation and source code of the API can be obtained from the authors.

## 5. APPLICATION OF THE API

This section shows how the API can be easily integrated into a real application. The application chosen is a Java implementation of the Agrawal et al. primality test algorithm, see [1]. This implementation requires a substantial amount of polynomial arithmetic.

The main thrust of the algorithm is contained in the following equation

$$(x - a)^n \equiv (x^n - a) \bmod (x^r - 1, n)$$

where $a, r, n \in \mathbb{Z}$. The integer to be tested for primality is $n$ (which must be greater than or equal to $2^{25}$ for the algorithm to be effective). If this condition holds for a range of $a's$ and for a specific $r$, $O(\log^6 n)$, then $n$ is prime ($a$ and $r$ are calculated in another part of the algorithm [1]). The left hand side of the equation requires a polynomial exponentiation. If a standard power ladder was used for this, the size of the $n$'s being used would make it impractical. Since this is a congruence modulo the polynomial $x^r - 1$ and the integer $n$, at each step of the ladder the current result can be modded out. The method `modPow()` in the `BigFieldPolynomial` class is very useful here. This method is overloaded so it is possible to mod out by a polynomial alone or a polynomial and integer together, depending on the arguments given. On the right hand side of the equation it is necessary to mod out by the polynomial $x^r - 1$. This can be achieved by using the `mod()` method, which takes a polynomial as an argument, and also the `modCoeff()` method which mods a polynomial by a `BigInteger` argument. Once both sides have been calculated the `equals()` method can be used to determine if the equation holds. The following code fragment demonstrates this

```
BigInteger n; // prime to be tested
BigInteger a, r;

// x^r - 1
BigFieldPolynomial modulus =
    new BigFieldPolynomial(n);
modulus.put(new BigMonomial(x, r));
modulus.put(new BigMonomial(
    BigInteger.ONE.negate(), BigInteger.ZERO));

// x - a
BigFieldPolynomial u =
    new BigFieldPolynomial(n);
u.put(new BigMonomial(x, BigInteger.ONE));
u.put(new BigMonomial(-a, BigInteger.ZERO));

// x^n - a
BigFieldPolynomial v =
    new BigFieldPolynomial(n);
v.put(new BigMonomial(x, n));
v.put(new BigMonomial(-a, BigInteger.ZERO));

// (x - a)^n (mod x^r - 1, n)
BigFieldPolynomial lhs = u.modPow(n, modulus);

// (x^n - a) (mod x^r - 1, n)
BigFieldPolynomial rhs =
    v.mod(modulus).modCoeff(n);

boolean prime = lhs.equals(rhs);
```

As already mentioned this algorithm only works when $n$ is greater than or equal to $2^{25}$. To show what this means for the equation above we will give a sample $n$ in this range and its associated values. If $n = 788369929 \approx 2^{30}$ then $r = 53700$ and $a \in [1, 13714]$. Considering that the largest known prime is $2^{13466917} - 1$ the sample above is a very small number for use in this algorithm. When testing for realistic primes the polynomials will get extremely large. Our API caters for such unbounded polynomials unlike other polynomial representations based on `array`'s. The API is also easy to use as can be seen from the code examples.

## 6. CONCLUSIONS AND FURTHER WORK

This paper outlined how Java can be used to develop a dedicated API for polynomial arithmetic. It also demonstrates the usefulness of the `BigInteger` class in developing large number mathematical applications.

Polynomial arithmetic is used in many mathematical applications mainly because integers can be easily represented as polynomials. We are currently working on the Agrawal

et al. primality test algorithm which should test the capabilities of the API. In the future we hope to implement more sample applications using this API to see how well it will perform. It is hoped that a Java implementation of an elliptic curve point counting algorithm based on Schoof's algorithm [2] will be one of these. To do this it will be necessary to extend to the API to cater for bivarate polynomials. Both these algorithms require intensive polynomial calculations and it is hoped our API will simplify the implementation and give some performance gain.

## 7. ACKNOWLEDGMENTS

Andrew Burnett and Adam Duffy would like to acknowledge the funding provided by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan. Adam Duffy also acknowledges NUI Maynooth's award of a studentship grant.

## 8. REFERENCES

[1] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. `http://www.cse.iitk.ac.in/news/primality.pdf`, Aug. 2002.

[2] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1st edition, 1999.

[3] R. Crandall and C. Pomerance. *Prime Numbers, A Computational Perspective*. Springer, 1st edition, 2001.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*. Addison Wesley, 1st edition, 1994.

[5] D. Knuth. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addison Wesley, 3rd edition, 1997.

[6] N. Koblitz. *A Course in Number Theory and Cryptography*. Springer, 2nd edition, 1994.