Kronecker substitution for polynomial multiplication

Monday 20th June, 2022 - 21:07

Rafidison Santatra Rakotondrasoa
University of Luxembourg
Email: rafidison.rakotondrasoa.001@student.uni.lu

Volker Müller
University of Luxembourg
Email: volker.muller@uni.lu

***Abstract*—This document presents the Bachelor Semester Project made by Rafidison Santatra Rakotonsrasoa under the direction of his Project Academic Tutor Volker Müller. The aim of this project concerns the Kronecker substitution method for polynomials multiplication. This project seeks to implement the Kronecker's method and we will work around this particular method in order to reduce the computational problem of the polynomial multiplication. The scientific aspect concerns the Kronecker substitution method itself aiming for minimal running time and we present some algorithms on this method. Furthermore, the technical aspect is chiefly the implementation of algorithms.**

1. Introduction

Polynomial multiplication is a wide-used operation. Our project is interested in a specific calculation that is the polynomial multiplication by Kronecker's substitution named after Leopold Kronecker a German mathematician who worked on number theory, algebra and logic. Multiplying polynomials is an algebraic expression that can be computed by several methods. There are different methods to multiply polynomials, for example, the direct brute force approach, the Karatsuba method used in for a small degree problems, the Fast Fourier Transform, the divide and conquer approach, the polynomial multiplication in Chebyshev basis and the Kronecker substitution.

Most don't know but the polynomial multiplication may be used in real life. We use the polynomial multiplication quite often than we imagine; for example, we use it in digital signal processing, image editing, the convolution of vectors used for the blur tool in Photoshop.

We know that there are many methods to multiply polynomials but in this project, we are interested in this specific method that is the Kronecker substitution method. The idea is to encode the coefficients of polynomials into long integers and using efficient integer library routines to do polynomial multiplication. The running time of a method is the features that make it interesting. We will explore in this paper the operation of Kronecker's method and how it is more efficient in some conditions than other methods.

2. Project description

2.1. Domains

The project aims to write a program that multiply polynomials with Kronecker's substitution. This project concerns mainly two domains that are the Kronecker substitution method, and computer programming and optimization of the algorithms. The method owned by Leonard Kronecker should be described in details in order to have an efficient manner to multiply polynomials. In this section, we present the scientific and technical domains of the project.

2.1.1. Scientific. The scientific domain of this project is more focused on Kronecker substitution method itself. The domain is more focused on some mathematical subjects such that the multiplication of polynomials, the binary, the multiplication of large integers.

The scientific work is the study of the Kronecker substitution method for polynomial multiplication. This method is a well-known technique to reduce polynomial multiplication to integer multiplication. In this project, the standard Kronecker idea shall be implemented.

2.1.2. Technical. The technical domain is focused chiefly on the implementation and assessment of Kronecker's algorithms. Thus, the algorithm's development is the technical aspect. We had to implement some tests and improvements of the algorithms with arbitrary large value of coefficients and degrees of each polynomial in order to have a fast running-times.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The aim in the scientific deliverables is on the one hand to understand the Kronecker's algorithms. We need to know how it works and how to optimize it. On the other hand, it is to test and compare function in an algorithm to have the best running time.

2.2.2. Technical deliverables. The aim in the produced technical deliverables is the implementation and the comparison of Kronecker's substitution algorithms. In this part, we also represent those algorithms comparisons into a graph.

3. Pre-requisites

3.1. Scientific pre-requisites

The scientific pre-requisite is mainly to be quite good in the mathematical domain. In the scientific section, we will describe some scientific methods to deal with Kronecker's method. For instance, we will deal with some binary operations like shifting and masking, we will work around polynomial function and other task in polynomial multiplication.

We should have some skills in programming, it is required to be able to understand, read, and reproduce a pseudo-code and programming code. It is also necessary to know how an algorithm works. In this project, we will work around some algorithms then, some knowledge related to algorithm and programming should be known like for example, what is a function, a variable, a class, an object, an array, a type of variable, a loop, a condition, a library and so on. Since we would like to build an efficient algorithm then the concept of algorithm complexity should be familiar.

3.2. Technical pre-requisites

The Technical pre-requisite is to be familiar with the programming itself, especially with C++ programming language. Of course, mastering this language is an advantage otherwise this programming language should be known and learned to implement our algorithm. As things progress, we can learn and be familiar with this language.

Another pre-requisite is to be able to integrate and to work with libraries. In this project, we will work more with an external library.

4. Scientific presentation of the Kronecker substitution method for polynomial multiplication

4.1. Requirements

The Kronecker substitution algorithms has to be looked in depth before implementing any line of code. It is required to understand the Kronecker's algorithms. The scientific dimension of the project deals with the way we would implement the algorithm. In this section, we will explicitly develop the Kronecker substitution method with some explanations and examples. The Kronecker point of view shall be understood.

The aim is to optimize the Kronecker's algorithm. Then, we will test each face of algorithms and looking for some improvement that can make the difference in the algorithms timing. In addition, the Kronecker's algorithm will perform several tests. Some tests on the Kronecker's algorithm shall be done. We will show the running time tests of the algorithm and compare the results.

4.2. Design

There are several manners to do polynomial multiplication. For example, one method to compute it is when we compute it by hand then if we want to multiply $f(x)$ and $g(x)$ then multiply each coefficient of f one by one by all of the coefficients of g, after adding up all the coefficients which have the same degree of x. The complexity of this basic polynomial multiplication method is in quadratic time, $O(n^2)$, which is too much. The problem is how can we efficiently calculate the coefficients of $f(x)$ and $g(x)$.

The method that we use is conceived by Kronecker to multiply polynomials. The concept of substitution in this method is the fact that instead of multiplying in $\mathbb{Z}[x]$ we multiply in $\mathbb{Z}$. Instead of multiplying each coefficient of two polynomials, we pack the coefficients in big integers and compute it in one single time. We can find three faces in Kronecker's algorithms, which are the pack function, the multiplication function and the unpack function. The pack function returns a large number that represents a polynomial, next the multiplication function multiply two large integers of two packed polynomials, and finally, the result is unpacked by the unpack function to obtain the product of the polynomials. The pack is a function which takes the coefficients of a polynomial and returns a large integer, and the other way around the unpack is a function which takes a big integer mostly the product of two big integers and returns a polynomial.

Kronecker substitution is a method to determinate the coefficients of an unknown polynomial by evaluating it at a single value [1]. This method reduce polynomial multiplication to a single multiplication and then ensure faster computing method for polynomial multiplication.

To find the polynomial coefficients h_i of the result of the multiplication of two polynomials, we have to evaluate each polynomial f and g. We evaluate polynomials by packing coefficients together.

If all the integer in the operation is represented in a power of two instead of a power of ten, then the big number is evaluated in binary which can be done by shifting and masking operations. For people, it's much easier to recognise and assign quantitative significance to the decimal number 179 than it's binary representation 10110011. Since computers use the binary code, the compute memory needs more resources to compute in decimal. Furthermore, the power of 2 can be implemented using the shifting operator which is faster than the computation with the power of 10. Power of 10 is a pretty straight-forward computation by hand but since computers work with bits and due to binary representation of the numbers, the computation in power of two seems faster. In this project, we considered that all integers are represented in base two.

Bit shifting operation of a binary number is a bitwise operation which manipulates the bits of a value. It is low cost

and faster for hardware to compute because binary is directly supported by processor and it uses less power because of the reduced use of resources. The shifting operators are represented by $<<$ as the shifting left operator and $>>$ as the shifting right operator.
For example:

$$(0010)_2 << 1 = (0100)_2$$

Or

$$2 << 1 = 4$$

The [figure 1] shows a left shift of a binary number by 1. Each bit is moved to the left and the empty position is filled with a zero.

$$(1011)_2 >> 1 = (0101)_2$$

Or

$$11 >> 1 = 5$$

The [figure 2] shows a right shift of a binary number by 1. Each bit is moved to the left and the empty position is filled by zero.

In one hand, if we want to multiply a number by 2 then we shift left the number by one, and in other hand if we want to divide a number by 2 we shift right the number by one. To multiply a number by 2^2 we shift left the number by two and to divide a number by 2^2 we shift right the number by two, and so on. The result of a Left Shift operation of n bit position is a multiplication by 2^n. The result of a Right Shift operation of n bit position is a division by 2^n. This operation uses less resources than the multiplication operation.
Note: When the first "1" is shifted out of the left edge, the operation has overflowed. The result of the multiplication is larger than the largest possible.

Bit masking is an operation of binary numbers which defines the bits we want to keep and the bits we want to clear. We will use this operation in the unpack function of Kronecker's algorithm. We apply a mask to a value. There exist three different Boolean operators in the bit masking, first the AND operation in order to extract a subset of the bits in the value, after the OR operation in order to set the subset of the bits in the value, and the XOR operation in order to toggle a subset of the bits in the value. We will use the Boolean operator AND to extract subsets of bits of a large integer.

Example of AND operation in binary :

$$Mask : 00001111$$

$$Value : 11011001$$

$$Result : 00001001$$

Or

$$Mask : 2^4 - 1$$

$$Value : 217$$

$$Result : 9$$

Applying the mask to the value by the AND operator means that we want to clear the first 4 bits, and keep the last 4 bits. Thus we have extracted the lower 4 bits.

4.3. Production

Notation: Let f, g, and $h \in Z[x]$ be polynomials respectively have degree m, n, and $m+n-1$:
$f(x) = f_0 + f_1x^1 + f_2x^2 + f_3x^3 + ... + f_{m-1}x^m$
$g(x) = g_0 + g_1x^1 + g_2x^2 + g_3x^3 + ... + g_{n-1}x^n$
$h(x) = h_0 + h_1x^1 + h_2x^2 + h_3x^3 + ... + h_{m+n-2}x^{m+n-1}$
where h is the product of f and g such that $h(x) = f(x).g(x)$

In this section, we will discuss about the way we proceed for multiplying polynomials with the Kronecker's method. We know $f(x)$ and $g(x)$ two polynomials such that $f(x) = \sum_{i=0}^{m} f_i x^i$ and g(x) = $\sum_{i=0}^{n} g_i x^i$. The aim is to obtain the polynomial $h(x) = \sum_{i=0}^{m+n-1} h_i x^i$. There are some algorithms related to the Kronecker substitution, such that:
- The standard Kronecker substitution
- The two-point negated Kronecker substitution
- The two-point reciprocal Kronecker substitution
- The four-point Kronecker substitution

Because of the lack of time, we will develop only the standard Kronecker substitution which evaluate polynomials in a single point.

The standard Kronecker substitution is basically evaluating polynomial in order to map it into an integer in a single point. We illustrate two examples below to facilitate understanding of the method. For the first example, we evaluate polynomials in base ten and for the second in base two. The task is to multiply two polynomials in a single variable with non-negative coefficients. Consider the problem of multiplying f(x) and g(x) and have as result h(x).

Examples : In these examples below, we multiply two polynomials in a single variable with non-negative integer coefficients. Let f, g, and $h \in Z[x]$ be three polynomials respectively have length m, n, and $m+n-1$. We know that *h* is the product of f and g. f_i, g_i and h_i are respectively the coefficients of *f*, *g* and *h*:

$$f(x) = 5x^2 + 4x + 18$$

$$g(x) = 2x^2 + 3x + 7$$

Example 1 : (In base ten)
The aim is to find h_i coefficients of the polynomial *h* for all i from 0 to *m+n-1*. First, we compute the point where the polynomials will be evaluated, we know $0 \leq f_i, g_i \leq 18$. We can bound h_i such that $0 < h_i < 3 \times 18^2$. We choose the smallest *b* such that $10^b > 3 \times 18^2$. We can deduce that

$b = 3$ such that $10^3 > 3 \times 18^2$. We evaluate polynomials to $x = 10^3$. Each polynomials should be evaluated in a single point that should be large enough to contain h_i. We obtain the integers $f(10^3) = 5|004|018$ and $g(10^3) = 2|003|007$. The vertical bars indicate base-10^3 digit boundaries. Our choice of evaluation point ensures that the coefficients do not overlap. When we multiply these integers we have a large integer $h(10^3) = 10|023|083|082|126$. We unpacked $h(10^3)$ to obtain $h(x) = 10x^4 + 23x^3 + 83x^2 + 82x + 126$.

Example 2 : (In base two)
The same as the previous example but in base two. We compute the bound 2^b where the polynomials will be evaluated. We assume that the length of f and the length of g are equal to n and the coefficients of f and g are bounded by 2^c, then the coefficients of h are bounded by $2^{2c}n$ where $h = fg$.

We know that $0 \leq f_i, g_i \leq 2^5$ where $c = 5$, hence, $0 < h_i < 2^b$ where $b = 3 \times 2^{5\times 2}$, we have $b = 12$. We evaluate polynomials to $x = 2^{12}$, we obtain the integers $f(2^{12}) = 83908482$ and $g(x) = 335566727$. When we multiply these integers we obtain a large integer $h(2^{12}) = 2816331707916414$. When we represent this integer in base two then we obtain $h(2^{12}) = (1010|000000010111|000001010011|000001010010|000001 111110)_2$. The vertical bars indicate base-2^{12} digit boundaries. Our choice of evaluation point ensures that the coefficients do not overlap. This sequence of bits represents the coefficients h_i of the polynomial h in base two. From the right to the left, each sub-sequence separated by the bar we convert the binary number to decimal, it is the unpack operation. We obtain $h(x) = 10x^4+23x^3+83x^2+82x+126$.

If we choose a power of two instead of a power of 10, and if the bignum representation is binary or some power of two, then evaluation into an integer can be done by shifting and masking. For the standard Kronecker substitution, we suppose that f and g have non-negative coefficients with c bits, where $c = \{c \in \mathbb{N} / 0 \leq f_i, g_i < 2^c \, for \, all \, i\}$. When we evaluate the polynomial in a specific point, we have to avoid that the coefficients of *h* overlap where the polynomial h is evaluated. So it suffices to evaluate at $x = 2^b$ where the bound b is gives below:

- If the length of f and the length of g are equal to n (i.e. f and g have degree $n - 1$), the coefficients of h are bounded by $2^{2c}n$, the polynomials $f(x)$ and $g(x)$ are evaluated in a single point $x = 2^b$ where $b = 2c + \lceil log_2(n) \rceil$ with $0 \leq h_i \leq 2^b$ for all i.

- If $length(f) = m$ and $length(g) = n$ with $m \neq n$ (i.e. f have degree $m-1$ and g have degree $n-1$) then the polynomials $f(x)$ and $g(x)$ are evaluated in a single point with $x = 2^b$ where $b = 2c + \lceil log_2(min(m, n)) \rceil$.

Note: $log_2(x) = log(x)/log(2)$

The standard Kronecker substitution algorithm is divided into three different faces which are:
- The pack function
- The multiplication
- The unpack function

The pack function is always called two times in the standard Kronecker's substitution of multiplication of two polynomials. The pack is called to evaluate two polynomials in a specific bound $x = 2^b$ where b is supposed to be already computed before and returns a large integer for each polynomial. Suppose that a polynomial f is packed into a large integer X and a polynomial g is packed into a large integer Y. The pack function uses the bit shifting operation to evaluate polynomials.

The multiplication function multiply the two big integers X and Y, and returns Z.

The unpack function of the standard Kronecker's substitution uses the bit shifting operation and the bit masking operation. Actually, this function is the reverse of the pack function. This function is called to return a third polynomial from a large integer Z.

Assuming we have an underlying binary representation, to multiply polynomials with the standard Kronecker substitution algorithm we compute the specific bound where the polynomials will be evaluated, pack, multiply, and unpack. Since the pack and the unpack functions use the bit shifting operation and the bit masking operation, packing and unpacking have linear time complexity.

Algorithm 1: Standard Kronecker Substitution

Result: The product of the two polynomials f and g
m = len(f);
n = len(g);
c = max(max(f),max(g));
$b = 2 * c + \lceil (log(min(m, n))/log(2)) \rceil$;
X = Pack(f,m,b);
Y = Pack(g,n,b);
Z = X*Y;
h = Unpack(Z,m+n-1,b);

4.4. Assessment

For the assessment, we focus on the algorithm faces such as the *pack*, the *multiplication* and the *unpack*. We evaluated its running times with different values of the size (degree plus one) and the coefficients of the polynomials. We performed several tests to analyse the trend of the running times of the three algorithm faces. We computed the average time of some different parameters of the polynomials (degree and coefficients). We will show the general tendency of the running times of the algorithm with compared to the variation of the polynomial coefficients and the polynomial size.

The tests are performed by a laptop with a x64-based processor Intel(R) Core(TM) i5-6300HQ CPU @ 2.30Ghz (4 CPUs) $\sim 2.3Ghz$. And as library, we used NTL version 8.1.2.

The graphs in [figure 1] and [figure 2] represent *pack* in **blue** color, *multiplication* in **red** color and *unpack* in **green** color.

This first graph [Figure 3] give us the running times tend of the pack function, the multiplication function and the unpack function depending on the values of the degrees of the polynomials. Each polynomial size (or the degree plus one) takes the value of the number of elements in a polynomial. Those running times represent the average of several running time tests depending on the value of the degrees of the polynomials. To perform each test, the value of the polynomial size is manually increased.
This graph is measuring the Kronecker's substitution method running times for the polynomial multiplication from 2000 to 30000 degrees. We observe that there are two important parts shown by this graph. From 2000 to 30000, on the one hand, the pack and the unpack running times steadily increased and on the other hand, the multiplication running times did not noticeably change. From 2000 to 30000 degrees of the polynomials, the multiplication function increased from 0.01101107 to 0.96657053 seconds, the pack functions increased from 0.24612013 to 15.97396 second and the unpack function increased from 0.3192569 to 16.12374 seconds.

This second graph [Figure 4] give us the running times tend of the pack function, the multiplication function and the unpack function depending on the values of the coefficients of the polynomials. Those running times represent the average of several running time tests depending on the value of the coefficients of the polynomials. The graph represents the polynomial coefficients as x-axis with the values represented in *bit* and the running times as y-axis with value represented in seconds. To perform these tests, each polynomial coefficient takes a random value of a specific number of bit. For instance, if we assign 15 bits to the coefficients of *f*, then for all i, $2^{14} \leq f_i \leq 2^{15} - 1$.
This graph is measuring the Kronecker's substitution method for polynomial multiplication running times from 2000 to 30000 degrees. We observe that there is two important parts shown by this graph. From 2000 to 30000, on the one hand, the multiplication running times steadily increased and on the other hand, the pack and the unpack running times did not noticeably change. From 2000 to 30000 degrees of the polynomials, the multiplication function increased from 0.13449035 to 9.65813867 seconds, the pack functions increased from 0.05071725 to 0.56136633 seconds and the unpack function increased from 0.03306646 to 0.28201527 seconds.
Note: the polynomial coefficients are represented in bits.

We can conclude that the pack and the unpack running times depend much more on the degrees of the polynomials, whereas the multiplication running times depend much more on the variation of the coefficients of the polynomials.

We plotted graphs for the tests with *Spyder* as Integrated Development Environment and used the programming language *python*. *Spyder* is a powerful scientific python development environment made for data exploration and analysis with interactive execution. We stored the running time's values in some CSV files which we can see in the next section and loaded the data file into a data frame. We used two libraries to represent our datasets in those two graphs, the *pandas* software library to load our CSV file into a data frame and the python two-dimension plotting library *matplotlib* which we used to build the graphs.

5. A Technical Deliverable 1

5.1. Requirements

In this section, the implementation of Kronecker's algorithm should be explained. The implementation is required to be in C++ by our tutor. First, we describe briefly this programming language, afterwards, we will show in this section how we implemented the Kronecker's algorithms.

In fact, the main idea here is to use the Kronecker's technique to provide fast implementation of a polynomial multiplication with less implementation by using efficient software like NTL library. When we use other library programs, we can benefit from their continued refinement as well as adaptation to new and improved hardware.

The aim is also to multiply polynomials such that the coefficients and the degrees are large. We have to include a library which can handle a very large number and improve our algorithms running times. C++ is the programming language that is required by our tutor to implement our algorithms, hence, it also required to use a C++ library such as NTL. We will show explicitly how we use the modules, classes and functions of this library.

The optimization of our implementation is also required. Of course, the code should be well implemented. In this regard, some tests about the algorithm running times are required to be done and at the same time, we targeted to have good running times.

5.2. Design

C++ is a computer programming compiled language that we used to implement the Kronecker's algorithm. We use a compiler to compile code written in C++. Even if we understand the code written in C++, the computer does not understand it. The compiler translates the human-readable code or high-level language to a lower lever language or a language processor which is understood by computers, and that process is called compilation or compiling. In our case,

the compiler is already integrated in our IDE. C++ code source files with a *.cpp*, *.c* or *.cc* extension are not the only files in C++ programs. The other type of file is called a header file. Header files usually have a *.h* or *.hpp* extension or no extension at all. The primary purpose of a header file is to propagate declarations to code files. The header store the constructor, the destructor and the lists of all the functions. These functions are defined in the source file. We call the syntax #$include$ which we can use wherever we want to call headers. Moreover, the default library in C++ is in the namespace std and NTL is in the namespace NTL. A namespace is used to organise the code into logical groups and to prevent name collisions that can occur when our code includes multiple libraries.

Example of code in C++ :

```
#include <iostream>
#include <NTL/ZZ.h>
#include <NTL/ZZX.h>

using namespace std;
using namespace NTL;

int main()
{
    std::cout << "Hello World!";
}
```

These lines of code display " Hello world! " after execution of the C++ program.

Now, we know how to write basic code in C++, after there are different steps which C++ program can be executed. A C++ program gets changed to different files during the execution steps before to be executed. In the beginning, we have a C++ program, for example, a file named *code.cpp* which contains the C++ source code. There are four steps to execute a code in C++ :

1. Preprocessing: the preprocessor takes a C++ source code file with *.cpp*, *.c* or *.cc* extension and includes the headers (#include) and expands the code. The preprocessor produces a single output file that is a C++ file without pre-processor directives (#$include$, #$define$). Preprocessor reads the *code.cpp* file and generates the *code.ii* file which contains the preprocessed code. These files are processed before the compilation.

2. Compilation: the compiler use the pre-processed source code files before to create an object file. The object file is not yet a code that we can run. First, the compiler produces the machine language instructions from the source code file. The compiler translates the source code file into a machine language file with an extension *.o* or *.obj*. The compiler reads the *code.ii* file and converts into assembly code and generates *code.s* and then finally generates object code in *code.o* file. This file can be linked by the linker.

3. Linking: the linker reads *code.o* file and links it with library files or/and other(s) object code which contain definitions of functions and objects we use in our program after it generates an executable file *code.exe*. The linker links all the object files and extracts these required definitions from the library file to build one single executable file with an extension *.exe*. Now the file can be executed.

4. Execution: the executable file is the final version of the program. This program is written in a language which the computer understands. Thus, the loader loads the executable file *code.exe* into the main/primary memory and produces the output and the program run. One more file is created that contains the source code named *code.bak* which is a backup file of the program files.

It is important to understand how the program is executed to understand how it works when some errors occur.

NTL library is an open source C++ library conceived by Professor Victor Shoup for the number of theory in 1882. The 2015 Richard Dimick Jenks Memorial Prize for Excellence in Software Engineering Applied to Computer Algebra was awarded to Professor Victor Shoup on October 30, 2015 for his project.[2] The aim of his project is to design, analyse and implement algorithms for number theory and algebra algorithms. NTL performs arbitrary multiple precision integer arithmetic and provides structures and algorithms for polynomials over the integers and over the finite fields. There are many mathematical algorithms about number theory in NTL that are represented into Modules. For each module foo there is a header file NTL/foo.h which includes another header file NTL/tools.h that have by default the standard headers *cstdlib*, *cmath* and *iostream*.

Examples of NTL Modules:

- zz_p is a class representing modular integers with word-sized modulus.

- ZZ_p is a class representing modular integers with arbitrary sized modulus.

- $GF2$ is a class representing the fields $GF(2)$ integers modulo 2. It means that all numbers are represented in $\mathbb{Z}/2\mathbb{Z}$.

- ZZ is a class representing signed, arbitrary length integers and provides routines for generating small primes, and fast routines for performing modular arithmetic on big numbers.

- ZZX is a class implementing univariate polinomials in $ZZ[x]$.

Since the Kronecker substitution method evaluate polynomials and perform the packing function to pack coefficients of these polynomials into long integers, we used an effi-

cient number theory library routines to multiply polynomials and in order to handle multiple-precision operation. In our project, we implemented with NTL modules ZZ and ZZX. Multiple-precision arithmetic or arbitrary-precision arithmetic is a calculation with very large number which its size is only limited by the host system memory and not based on the length of the variable that allocates a precise value with a limited size.

Microsoft Visual Studio is an Integrated Development Environment (IDE) which contains a compiler, debugger, IntelliSence, code completion with an integrated terminal [4]. IDE refers to an environment in which the source code is written, compiled, linked and executed. We used Visual Studio Express 2012 for Windows Desktop as an Integrated Development Environment. This is the free version of Microsoft Visual Studio 2012. It allows the creation of software in C++, C# or Visual Basic languages and integrates the latest refinements, in particular programming tools as well as optimized compilers. Finally, this IDE offers an excellent learning environment for students.

5.3. Production

Implementation of the Kronecker's standard algorithm

When we initialise the ZZX and ZZ classes to represent polynomials and its coefficients, it calls those constructors that we can see below:

```
const ZZX& ZZX::zero()
{
   NTL_THREAD_LOCAL static ZZX z;
   return z;
}


const ZZ& ZZ::zero()
{
   NTL_THREAD_LOCAL static ZZ z;
   return z;
}
```

We used some functions in the class ZZX and the class ZZ from the NTL library to implement the Kronecker's algorithm :

- The function that the code is represented below returns the degree of the *ZZX*. In our case, *ZZX* represent a polynomial, then the function *deg(f)* returns the degree of a polynomial *f*.

```
long deg(const NTL::ZZX &a)
```

- The function that the code is represented below returns the bit which represents the biggest element in the *ZZX*. In our case, it is used to return the bit of the biggest coefficient in a polynomial, which is what this next program does.

```
long MaxBits(const NTL::ZZX &f)
```

- The function that the code is represented below assign in x the value of the product between a and b. In our case, it is used to multiply to large integers X and Y and set the result in Z.

```
mul(NTL::ZZ &x, const NTL::ZZ &a,
const NTL::ZZ &b)
```

Like we have seen before, we pack polynomial coefficients into a big number. The class ZZX represents polynomials and its coefficients are represented by the class ZZ. Since the coefficients are represented in base two, when we evaluate the polynomial we shift left each coefficient by the degree of x.

```
ZZ Pack(ZZX f, long m, long b)
{
        ZZ X;
        int i;
        for (i=0;i<m;i++)
                X += (f[i]<<(b*i));
        return X;
}
```

This function Pack takes as parameters a polynomial, the length of the polynomial, and b where the polynomial is evaluated in a single point $x = 2^b$. This function returns $f(2^b)$.

One more efficient algorithm for the polynomial packing function is to use the Horner's rule. According to Horner's rule:

$$f(x) = f_0 + f_1x^1 + f_2x^2 + f_3x^3 + ... + f_{m-1}x^{m-1}$$

$$f(x) = f_0 + x(f_1 + x(f_2 + x(f_3 + ... + x(f_m - 1))))$$

For all i from $m-1$ downto 0, instead of shifting left by *(b * i)*, we can only used the shift left by *b*:

$$X = f[i] << (b * i)$$

$$X = f[i] + (X << b)$$

```
ZZ Pack(ZZX f, long m, long b)
{
        ZZ X;
        int i;
        for (i=m-1;i>=0;i--)
                X = (f[i] + (X<<b));
        return X;
}
```

The *unpack* function below extracts h_i from the large integer Z which is the product of the two packed polynomials. h_i represent the coefficients of the polynomial h where $h(x) = f(x)g(x)$. This function uses the AND operator (& in C++) and the left shift operation to extract the coefficients.

For example, we want to unpack this large integer represented in base two $h(2^{12})$ or $(1010|000000010111|000001010011|000001010010|000001111110)_2$. We AND it with $(2^{12}-1)$ or $(111111111111)_2$ and we obtain the last significant coefficient of h. We shift $h(2^{12})$ of 12 positions to the left to drop the last sub-section of $h(2^{12})$, so we obtain $(1010|000000010111|000001010011|000001010010)_2$. We AND it with $(2^{12}-1)$ or $(111111111111)_2$ to obtain one more coefficient of h and shift $h(2^{12})$ of 12 positions to the left.

We repeat this operation until we reach the last coefficient of h.

```
ZZX Unpack(ZZ Z, long l, long b)
{
        ZZX h;
        long x = ((1<<b)-1);
        for(int i=0;i<l;i++){
                SetCoeff(h,i,Z&x);
                Z >>= b;
        }
        return h;
}
```

Implementation for the algorithm's tests

To store the running time of the tests in a file we used the library *fstream* and the library *chrono* which are standard library. With the lines of code below, we include our libraries:

```
#include <fsrteam>
#include <chrono>
```

Fstream library defines several classes that support input/output streams operations with external files. Actually, *fstream* is a data type which represents the file stream generally, and has the capabilities of both *ofstream* and *ifstream*. We can create files, write information to files, and read information from files. For instance, in our case, we saved the running time tests in a *csv* file as *times.csv*.

Chrono library is used to deal with date and time. This library have the name of a header *chrono* and all function are define in *std::chrono namespace*. The asset of this library is that it has a clock with a high resolution of time, the *high_resolution_clock*. This is the clock with the highest time precision in this library, we can see other clocks like *time_point:system_clock* and *steady_clock*.

```
#include <ofstream>
#include <chrono>
using namespace std;
int main()
{
    ofstream myFile;
    myFile.open("times.csv",ios::app);
    myFile << "Pack" << endl;
    auto start = chrono::high_resolution
_clock::now();
    X = Pack(f,m,b);
    auto end = chrono::high_resolution_
clock::now();
    chrono::duration<float> duration =
end - start;
    myFile << duration.count() << endl;
}
```

The parameter *ios:app* indicates that all output to that file are append to the end.

To assign a random value of n bits to the coefficients of the polynomial f with the degree d:

```
for(long i=0;i<=d;i++)
    SetCoeff(f,i,RandomLen_ZZ(n));
```

5.4. Assessment

We run some tests to show the running times of the standard Kronecker substitution method for polynomial multiplication. Each row of the table below represents one test. For each test, two polynomials are created with a random value of 1000-bits for each coefficient in those polynomials with the degree 999. We compile those tests with Visual Studio Express 2012, and we used the NTL library version 8.1.2. Since the performance of a computer is always unstable each row shows different values every time.

The table below represents the running times (in seconds) of the Kronecker substitution method using the NTL version 8.1.2 for the multiplication of two polynomials with 1000-bits assign to each coefficient and the degree of the polynomials is set to 999.

Pack	Multiplication	Unpack	Total
0,88663	1,82612	0,644277	3,357027
0,925526	1,82217	0,653212	3,400908
0,883637	1,83609	0,609371	3,329098
0,8976	1,81914	0,53856	3,2553
0,924493	1,83509	0,543546	3,303129
0,909568	1,82615	0,548547	3,284265
0,916534	1,83008	0,546567	3,293181
0,916521	1,82016	0,550501	3,287182
0,920539	1,82013	0,560501	3,30117
0,911575	1,8191	0,550556	3,281231
0,911563	1,82509	0,552549	3,289202
0,917561	1,83109	0,553492	3,302143
0,919541	1,81717	0,548506	3,285217
0,916546	1,855	0,541552	3,313098
0,914527	1,81817	0,548505	3,281202

The table below represents the running times (in seconds) of the Kronecker substitution method using the NTL version 9.7.0 for the multiplication of two polynomials with 1000-bits assign to each coefficient and the degree of the polynomials is set to 999.

Pack	Multiplication	Unpack	Total
0,88767	24,0128	0,660235	25,560705
0,92554	23,7375	0,656246	25,319286
0,880645	23,6867	0,600354	25,167699
0,877682	23,6677	0,536564	25,081946
0,860715	23,8452	0,545541	25,251456
0,899602	23,7704	0,642283	25,312285
0,897589	23,6956	0,602361	25,19555
0,929516	23,8014	0,574422	25,305338
0,932534	23,6916	0,59046	25,214594
0,930508	23,7235	0,611401	25,265409
0,951459	23,8452	0,591416	25,388075
0,935499	23,7026	0,59542	25,233519
0,937452	23,7515	0,589451	25,278403
0,932479	23,7396	0,60035	25,272429
0,946429	23,6857	0,591418	25,223547

NTL 8.1.2 is faster than the NTL 9.7.0 for the multiplication operation of two *ZZ* objects. The pack and the unpack function are likely the same, whereas the multiplication has a huge difference between those two libraries. Our algorithm is faster using the NTL 8.1.2 library.

Since the numbers are represented in binary, the pack and unpack functions are linear. Kronecker's standard substitution algorithm evaluates polynomials at a single point $x = 2^b$, more efficient algorithms can evaluate polynomials at two points $x = \pm 2b'$ where $b' = b/2$ or four points $x = \pm 2^{\pm b"}$ where $b" = b/4$. Its algorithms improve the complexity of multiplication. The Kronecker substitution algorithm depends on the complexity of the multiplication. To optimize the multiplication, some advanced algorithm such that the *two-point* and the *four-point* Kronecker substitution can be performed. It tends to reduce the complexity of multiplication and the complexity of the algorithm.

Acknowledgment

Firstly, I would like to thank the entire teaching staff of the University of Luxembourg and the professional contributors responsible for the Bachelor in Computer Science (BiCS) training course.

I would like to thank Mr Nicolas GUELFI, BiCS course director, for his help and support in carrying out this report and this project. I would also like to thank my Project Academic Tutor Mr Volker MÜLLER, for this project, for instantiating this project, for his time spent helping me throughout the project and for the advice concerning the missions mentioned in this report.

6. Conclusion

We worked with the Visual Studio Express which is an IDE or Integrated Development Environment and we used the programming language C++ and the NTL library. This library provided us with an arbitrary-length integers and polynomials over the integers and finite field.

We did some analysis, tests and improvements of the Kronecker's algorithm implementation. To represent the running times of the algorithm tests and to understand more about the output data we used Spyder with python programming language with some libraries to manage those tests such that pandas and matplotlib.

In this paper, the aim in this bachelor semester project is to explore and implement the standard Kronecker substitution idea. We developed the standard Kronecker substitution which evaluate polynomials in a single point but there are more enhanced points in Kronecker one could think to implement the multi-point Kronecker substitution.

References

[1] BiCS Bachelor Semester Project Report Template https://github.com/nicolasguelfi/lu.uni.course.bics.global University of Luxembourg, BiCS - Bachelor in Computer Science (2017).

[2] 2015 Richard Dimick Jenks Memorial Prize Award Victor Shoup for NTL http://www.sigsam.org/awards/jenks/awardees/2015/

[3] BiCS Bachelor Semester Project reference document BiCS Semester Projects Reference Document. Technical report, University of Luxembourg (2018)

[4] Microsoft visual studio features https://docs.microsoft.com/en-us/visualstudio/ide/advanced-feature-overview?view=vs-2019#modular-installation Microsoft

[5] Microsoft visual studio documentation https://docs.microsoft.com/en-us/visualstudio/ide/?view=vs-2019 Microsoft

[6] Openclassroom courses https://openclassrooms.com/en/courses/1894236-programmez-avec-le-langage-c Openclassroom

[7] NTL documentation https://www.shoup.net/ntl/doc/tour.html NTL

[8] Matplotlib library documentation https://matplotlib.org/contents.html Matplotlib library

[9] Pandas library documentation https://pandas.pydata.org/pandas-docs/stable/ Pandas library

7. Appendix

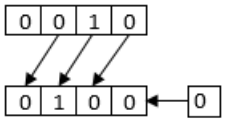

Figure 1. A left shift

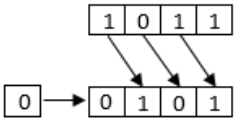

Figure 2. A right shift

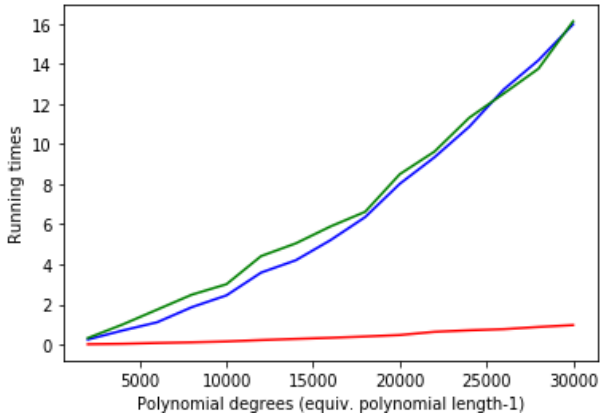

Figure 3. Kronecker's method for multiplication of polynomials with variate degrees and **15-bits** coefficients

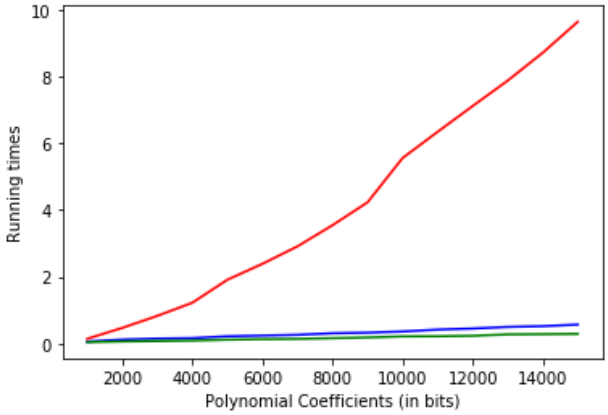

Figure 4. Kronecker's method for multiplication of polynomials with variate coefficients and the **200**