Duale Hochschule Baden-Württemberg Mannheim

Studienarbeit

Arithmetik endlicher Körper (speziell in der Charakteristik 2)

Studiengang Informatik

Studienrichtung Angewandte Informatik

Verfasser(in):	Lars Krickl
Matrikelnummer:	2512317
Kurs:	TINF22 AI2
Studiengangsleiter:	Prof. Dr. Holger D. Hofmann
Wissenschaftliche(r) Betreuer(in):	Prof. Dr. Reinhold Hübl
Bearbeitungszeitraum:	15.10.2024 - 15.04.2025

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit mit dem Titel ”Arithmetik endlicher Körper (speziell in der Charakteristik 2)” selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum: _____ **Unterschrift:** _____

Inhaltsverzeichnis

Abstract

Die vorliegende Arbeit befasst sich mit dem Thema ”Arithmetik endlicher Körper (speziell der Charakteristik zwei)”, welches in den folgenden Schritten bearbeitet wird.

Zu Beginn wird eine theoretische Grundlage geschaffen, indem zunächst auf allgemeine und endliche algebraische Körper eingegangen wird. Im Zuge der endlichen Körper wird von Primzahlkörpern als Grundlage auf Erweiterungskörper geschlossen. Dann wird beschrieben, wie endliche Körper mittels einer Relation eindeutig definiert werden können und wie Polynome über endlichen Körpern beschrieben werden, wobei auf deren Nullstellen eingegangen wird. Danach werden die arithmetischen Operationen der Addition, Multiplikation und Division für Polynome beschrieben. Darauf aufbauend wird der euklidische Algorithmus für Polynome sowie die Bildung von multiplikativ-inversen Elementen in endlichen Körpern erklärt. Dann werden die Irreduzibilitätskriterien von Polynomen aufgezeigt, woraufhin die Konstruktion von endlichen Körpern mittels irreduzibler Polynome erläutert wird. Als Letztes wird im Theorieteil der Cantor-Zassenhaus-Algorithmus beschrieben, welcher ein effizientes Verfahren zur Bestimmung der Irreduzibilität von Polynomen darstellt.

Im Implementierungsteil werden die zuvor aufgeführten theoretischen Grundlagen mittels der Programmiersprache Python praktisch umgesetzt, wobei der Fokus speziell auf der Charakteristik zwei liegt. Dabei wird zunächst eine Basis geschaffen, indem eine effiziente Darstellung von Polynomen mittels binärer Tupel eingeführt wird und zudem verschiedene Hilfsfunktionen implementiert werden, die den Umgang mit Polynomen, bzw. mit Tupeln vereinfachen. Dazu zählen beispielsweise das Wegschneiden von führenden Nullen, die Berechnung des Grades eines Polynoms sowie das Überprüfen auf Nullstellen eines Polynoms über dem Grundkörper $\mathbb{F}_2$. Im Anschluss werden Algorithmen für die zuvor theoretisch beschriebenen arithmetischen Operationen entwickelt und zudem anhand von Beispielrechnungen veranschaulicht. Im weiteren Verlauf wird der euklidische Algorithmus sowie zwei Algorithmen zur Bestimmung der Irreduzibilität von Polynomen implementiert, wozu der Cantor-Zassenhaus-Algorithmus gehört. Für die Polynommultiplikation und für die Bestimmung der Irreduzibilität von Polynomen werden jeweils zwei verschiedene Algorithmen implementiert, weshalb am Ende deren Performance jeweils untereinander verglichen wird.

Der für diese Studienarbeit entwickelte Code befindet sich im Anhang: ’Quellcode’

1 Einleitung

1.1 Motivation

Endliche Körper sind mathematische Strukturen, die eine endliche Menge K an Elementen und zwei Operationen $+$ und $\circ$ enthalten. Diese Operationen sind auf der endlichen Menge K definiert. Zudem müssen zusätzliche Bedingungen wie die Existenz von neutralen bzw. inversen Elementen oder mathematischen Gesetzen wie dem Assoziativgesetz, dem Kommutativgesetz oder dem Distributivgesetz eingehalten werden.

[4]

Diese endlichen Körper bilden ein zentrales Forschungsfeld der Algebra und finden in verschiedenen Bereichen ihre Anwendung. Dazu zählen insbesondere die Kryptologie und die Kodierungstheorie. Dabei haben endliche Körper der Charakteristik zwei, aufgrund ihrer einzigartigen algebraischen Struktur, eine besondere Rolle, da in der Charakteristik zwei die Gleichung $1 + 1 = 0$ gilt. Dadurch bieten sich diese endlichen Körper für binäre Systeme und digitale Anwendungen an, da sie nahtlos mit der Binärarithmetik von Computern funktionieren.

Durch die schnelle Entwicklung der Technologie und der daraus resultierenden Verbesserung der Rechenleistung steigen auch die Sicherheitsanforderungen, die diese endlichen Körper gewährleisten müssen. In der modernen Kryptografie werden komplexere Strukturen benötigt, um Angriffe durch leistungsstarke Computer abwehren zu können. Dazu liefern endliche Körper eine stabile Grundlage, da sie durch ihre mathematische Struktur effiziente Berechnungen ermöglichen, wobei sie gleichzeitig schwer zu berechnen sind.

[5]

Ein verbreitetes Einsatzgebiet von endlichen Körpern der Charakteristik zwei ist die elliptische Kurven-Kryptografie, auch ECC genannt. Diese benötigt im Vergleich zu anderen kryptografischen Systemen wie RSA kleinere Schlüsselgrößen, ist jedoch ebenfalls auf einem hohen Sicherheitsniveau.

Für die elliptische Kurven-Kryptografie gibt es Standards, wie die vom 'National Institute of Standards and Technology' (NIST) spezifizierten Körper, worunter unter anderem die Körper $\mathbb{F}_{2^{163}}, \mathbb{F}_{2^{233}}, \mathbb{F}_{2^{283}}$. Diese Körper bestimmen den Grundstein für

den Einsatz von elliptischen Kurven.

Diese Körper werden durch irreduzible Polynome definiert. Zum Beispiel kann der endliche Körper $\mathbb{F}_{2^{163}}$ durch die Relation $\alpha^{163} = \alpha^7 + \alpha^6 + \alpha^3 + 1$ definiert werden. Relationen, bei denen auf der rechten Seite der Gleichung vergleichsweise niedrige Potenzen stehen, führen zu einer effizienten Berechnung. Diese niedrigen Exponenten führen dazu, dass der Rechenaufwand in diesem Körper gering bleibt, im Vergleich zu höheren Potenzen, die aufwendigere Berechnungen erfordern.

Mit dem Fortschreiten der technologischen Entwicklung zeigt sich, dass bereits Körper wie $\mathbb{F}_{2^{163}}$ an ihre Grenzen stoßen. Um den gestiegenen Sicherheitsanforderungen der Kryptologie zu entsprechen, werden immer größere Körper mit Ordnungen bis zu 2^{400} in Betracht gezogen. Diese größeren Körper bieten eine höhere Sicherheit, da die zugrundeliegenden Berechnungen, mit zunehmender Größe des Körpers, exponentiell schwieriger zu lösen werden. Allerdings steigt mit der Größe des Körpers auch der Rechenaufwand, weshalb ein Gleichgewicht zwischen Sicherheit und Effizienz gefunden werden muss.

[3, 12]

1.2 Struktur der Arbeit

Ziel dieser Arbeit ist es, die Arithmetik endlicher Körper sowohl aus theoretischer als auch aus praktischer Perspektive zu untersuchen. Im ersten Teil werden endliche Körper und deren Arithmetik allgemein, theoretisch betrachtet, wobei auf grundlegende Eigenschaften und Rechenoperationen eingegangen wird. Zu den Rechenoperationen zählen unter anderem die Addition, die Multiplikation und die Division. Zudem wird die Eigenschaft der Irreduzibilität von Polynomen aufgeführt. Dabei wird insbesondere auf die endlichen Körper der Charakteristik zwei eingegangen.

Im praktischen Teil, bzw. in dem Implementierungsteil werden die zuvor theoretisch betrachteten Eigenschaften von endlichen Körpern, speziell für die Charakteristik zwei, in der Programmiersprache Python implementiert. Dabei werden die jeweiligen Algorithmen genau beschrieben und anhand von Beispielen verdeutlicht. Schließlich werden für zwei der Operationen, und zwar der Polynommultiplikation und der Überprüfung von Polynomen auf Irreduzibilität, Performancevergleiche durchgeführt, da für jede der beiden Operationen zwei verschiedene Algorithmen implementiert werden können.

2 Theorie

In diesem Kapitel werden endliche Körper und deren Arithmetik allgemein beschrieben. Dazu werden zunächst allgemeine Körper beschrieben, woraufhin endliche Körper spezifiziert werden. Im Anschluss werden Polynome über endlichen Körpern beschrieben, woraus auf die Arithmetik endlicher Körper geschlossen wird. Schließlich wird auf die Irreduzibilität von Polynomen eingegangen. Dabei werden jeweils Beispielrechnungen aufgeführt, die die jeweiligen Operationen genauer beleuchten sollen.

2.1 Körper allgemein

Zu Beginn betrachten wir die Eigenschaften eines Körpers in der Algebra, wobei wir Bosch [1, S. 35–38] und Waldmann [11, S. 116–118] folgen.

Ein Körper ist eine algebraische Struktur $(K, +, \cdot)$, die zum einen aus einer nichtleeren Menge K mit zwei ausgezeichneten Elementen 0 und 1, und zum anderen aus zwei Verknüpfungen $+$ und $\cdot$ besteht. Dieser Körper setzt sich aus zwei einzelnen abelschen Gruppen $(K, +)$ mit dem neutralen Element 0 und $(K \setminus \{0\}, \cdot)$ mit dem neutralen Element 1 zusammen. Die Voraussetzungen dafür, dass eine Menge G zusammen mit einer Verknüpfung $\circ$ eine abelsche Gruppe bildet, lauten folgendermaßen:

Die erste Bedingung ist die Abgeschlossenheit, das bedeutet, dass das Ergebnis einer Verknüpfung von zwei beliebigen Elementen der Menge G wiederum in der Menge G liegt. Also muss für alle $a, b \in G$ gelten:

$$a \circ b \in G$$

Für die zweite Bedingung muss das Assoziativgesetz gelten. Das heißt, die Reihenfolge, in der die Verknüpfung von drei oder mehr Elementen ausgeführt wird, spielt keine Rolle. Demnach muss für alle $a, b, c \in G$ gelten:

$$(a \circ b) \circ c = a \circ (b \circ c)$$

Die dritte Bedingung ist die Existenz eines neutralen Elements. Ein neutrales Element ist ein Element der Menge G für das gilt, dass ein beliebiges Element durch die Verknüpfung mit dem neutralen Element nicht verändert wird. Somit muss ein $e \in G$ existieren, für das gilt:

$$a \circ e = e \circ a = a$$

Die vierte Bedingung ist die Existenz eines inversen Elements für jedes Element aus der Menge G. Ein inverses Element hat die Eigenschaft, dass, wenn ein Element der Menge K mit seinem zugehörigen inversen Element verknüpft wird, das neutrale Element e als Ergebnis herauskommt. Also muss für jedes $a \in G$ ein inverses Element $a^{-1} \in G$ existieren, sodass gilt:

$$a \circ a^{-1} = a^{-1} \circ a = e$$

Als fünfte und letzte Bedingung für eine abelsche Gruppe muss das Kommutativgesetz gelten. Das bedeutet, die Reihenfolge der Verknüpfung zweier Elemente spielt keine Rolle. Demnach muss für alle $a, b \in G$ gelten:

$$a \circ b = b \circ a$$

Damit durch die beiden abelschen Gruppen $(K, +)$ und $(K \setminus \{0\}, \cdot)$ einen Körper gebildet wird, muss zudem das Distributivgesetz gelten. Für das Distributivgesetz muss für alle $a, b, c \in K$ gelten:

$$a \cdot (b + c) = (a \cdot b) + (a \cdot c)$$

Mit diesen zwei Gleichungen wird sowohl die Links- als auch die Rechtdistributivität abgedeckt.

Ein bekanntes Beispiel für einen Körper lautet $(\mathbb{R}, +, \cdot)$, wobei die reellen Zahlen $\mathbb{R}$ die Menge der Elemente beschreibt, und die beiden Verknüpfungen durch die Addition $(+)$ und die Multiplikation $(\cdot)$ dargestellt werden. Dabei stellen sowohl $(\mathbb{R}, +)$, mit neutralem Element 0, als auch $(\mathbb{R} \setminus \{0\}, \cdot)$, mit neutralem Element 1, jeweils eine abelsche Gruppe dar. Bei der Multiplikation muss die Null jedoch ausgeschlossen werden, da es für die Null kein inverses Element gibt, mit dem man auf das neutrale Element 1 kommt. Ein weiteres Beispiel ist der Körper $(\mathbb{C}, +, \cdot)$ über den komplexen Zahlen $\mathbb{C}$, mit den Verknüpfungen der Addition $(+)$ und der Multiplikation $(\cdot)$.

2.2 Endliche Körper

Endliche Körper sind algebraische Strukturen, die in verschiedenen Bereichen der Mathematik ihre Anwendung finden. Dazu zählen beispielsweise die Themengebiete der Codierungstheorie und der Kryptographie. Endliche Körper erfüllen die gleichen Eigenschaften wie allgemeine Körper, jedoch mit der Besonderheit, dass die Menge K des Körpers $(K, +, \cdot)$ lediglich eine endliche Anzahl an Elementen hat.

2.2.1 Primzahlkörper

Die Eigenschaften der Primzahlkörper wurden basierend auf Bosch [1, S. 43–44], Kunz [6, S. 64–68, 73–75] und Kurzweil [7, S. 11–12] herausgearbeitet.

Primzahlkörper stellen den einfachsten Fall von endlichen Körpern dar, die dabei jeweils genau p Elemente haben, wobei p eine Primzahl ist. In diesen Primzahlkörpern werden alle ganzen Zahlen betrachtet, und diese werden modulo der Primzahl p genommen. Das heißt, durch die Modulo-Rechnung entstehen Restklassen, und jede dieser Restklassen stellt dann ein Element in dem Primzahlkörper $\mathbb{F}_p$. Die Elemente eines Primzahlkörpers $\mathbb{F}_p$ lauten demnach $0, 1, ..., p-1$.

Ein Primzahlkörper $\mathbb{F}_p$ kann auch als Restklassenring der ganzen Zahlen modulo p betrachtet werden und somit durch $\mathbb{Z}/p\mathbb{Z}$ dargestellt werden. Sollte p jedoch keine Primzahl sein, sondern eine beliebige natürliche Zahl n, so stellt $\mathbb{Z}/n\mathbb{Z}$ keinen endlichen Körper dar. Dies liegt daran, dass, wenn n keine Primzahl ist, nicht jedes Element des Restklassenrings ein multiplikatives Inverses hat.

Ein Element a hat genau dann ein multiplikatives Inverses, wenn es ein Element b gibt, sodass gilt:

$$a \cdot b \equiv 1 \quad mod\, n$$

Das heißt, es gibt eine ganze Zahl k, sodass gilt:

$$a \cdot b = 1 + k \cdot n$$

Diese Gleichung ist für b genau dann lösbar, wenn gilt:

$$ggT(a, n) = 1$$

also wenn a und n teilerfremd sind.

Sollte nun n keine Primzahl sein, so gibt es Elemente, die keine Vielfachen von n sind, aber trotzdem nicht teilerfremd zu n sind. Sind beispielsweise $n = 6$ und $a = 2$, so stellt a zwar kein Vielfaches von n dar, jedoch beträgt $ggT(2, 6) = 2$ und somit sind diese nicht teilerfremd. Somit stellt $\mathbb{Z}/n\mathbb{Z}$, sollte n keine Primzahl sein (wie $n = 6$) lediglich einen Ring dar. Nur wenn n eine Primzahl ist, gilt $ggT(a, n) = 1$ für alle a, wobei gelten muss: $0 < a < n$.

Die Charakteristik eines Körpers ist die kleinste natürliche Zahl m, für die gilt:

$$m \cdot 1 = 0$$

das bedeutet, dass 1 genau m-mal mit sich selbst addiert werden muss, um wieder auf 0 zu kommen.

Für Primzahlkörper $\mathbb{F}_p$ ist die Charakteristik genau die Primzahl p, da gilt:

$$p \cdot 1 \equiv 0 \quad mod\, p$$

2.2.2 Allgemeine endliche Körper

Dieser Abschnitt behandelt die Eigenschaften von allgemeinen endlichen Körpern, bzw. von Erweiterungskörpern. Diese Eigenschaften beruhen auf Kunz [6, S. 185–187] und Kurzweil [7, S. 110].

Allgemeine endliche Körper stellen Erweiterungen von Primzahlkörpern dar. Diese endlichen Körper haben dann p^n Elemente, wobei p die Primzahl des zugrundeliegenden Primzahlkörpers darstellt, und n eine beliebige positive ganze Zahl ist. Diese Körper werden in der Form $\mathbb{F}_{p^n}$ dargestellt. Jeder endliche Körper $\mathbb{F}_{p^n}$ kann als algebraische Erweiterung des Primzahlkörpers $\mathbb{F}_p$ aufgefasst werden, wobei die Erweiterung durch ein irreduzibles Polynom vom Grad n über $\mathbb{F}_p$ definiert wird.

Die Existenz und Eindeutigkeit solcher Körper wurde bereits von Évariste Galois gezeigt, weshalb sie auch als Galois-Körper bezeichnet werden.

Die Primzahl p definiert die Charakteristik des endlichen Körpers und die positive ganze Zahl n bestimmt den Grad der Körpererweiterung über $\mathbb{F}_p$. Beispielsweise hat der Körper $\mathbb{F}_{2^3}$ die Charakteristik 2, das heißt, die Menge, auf der der Grundkörper $\mathbb{F}_2$ definiert ist, besteht lediglich aus den beiden Elementen 0 und 1. Der Grad der Körpererweiterung beträgt in diesem Beispiel 3. Demnach hat dieser endliche Körper $2^3 = 8$ Elemente.

Die Charakteristik eines allgemeinen endlichen Körpers $\mathbb{F}_{p^n}$ ist immer genau die Primzahl p des Grundkörpers $\mathbb{F}_p$.

Wie in Satz 5.3 von [10] gezeigt wird, ist die Einheitengruppe $\mathbb{F}_{p^n}^*$ von $\mathbb{F}_{p^n}$ immer zyklisch. Ein Erzeuger der Einheitengruppe von $\mathbb{F}_{p^n}$ heißt primitives Element von

$\mathbb{F}_{p^n}$. Damit schreibt sich jedes $x \in \mathbb{F}_{p^n}^*$ als $x = \alpha^t$ mit $1 \leq t \leq p^n - 1$, wobei $\alpha^{p^n-1} = 1$. Also gilt:

$$\mathbb{F}_{p^n} = \{0, 1, \alpha, \alpha^2, \ldots, \alpha^{p^n-2}\}$$

Das letzte Element hat die Potenz $p^n - 2$, da wir p^n Elemente benötigen und die beiden Elemente 0 und 1 ohne eine α-Potenz dargestellt werden können (wobei sich die 1 auch als p^{n-1}-te Potenz von α schreiben lässt). In dem Beispiel, für den Körper $\mathbb{F}_8$ gilt $n = 3$. Somit wird das letzte Element (das Element mit der höchsten Potenz) zu:

$$\alpha^{2^3-2} = \alpha^{8-2} = \alpha^6$$

Aus dem kleinen Satz von Fermat folgt (und dem offensichtlichen Fall für $\alpha = 0$), dass in einem endlichen Körper $\mathbb{F}_{p^n}$ für jedes $a \in \mathbb{F}_q$ gilt: $a^q = a$, wenn $q = p^n$. Somit können Potenzen, die größer als $q-1$ sind, lediglich modulo $q-1$ gerechnet werden.

2.2.3 Beschreibung durch definierende Relation

In diesem Abschnitt werden, basierend auf Kurzweil [7, S. 141–142] und Werner [13, S. 37–41], die Eigenschaften von definierenden Relationen beschrieben.

Eine definierende Relation von $\mathbb{F}_{p^n}$ ist gegeben durch ein Element $\alpha \in \mathbb{F}_{p^n}$ für das $1, \alpha, \ldots, \alpha^{n-1}$ linear unabhängig über $\mathbb{F}_p$ sind. Diese schaffen damit eine Basis von $\mathbb{F}_{p^n}$ als $\mathbb{F}_p$-Vektorraum. Durch den Fakt der linearen Unabhängigkeit existiert eine eindeutige Beziehung

$$\alpha^n = r_{n-1}\alpha^{n-1} + \cdots + r_1\alpha + r_0 \quad (1)$$

mit $r_i \in \mathbb{F}_p$. Diese wird dann als definierende Relation des Körpers $\mathbb{F}_{p^n}$ bezeichnet. Dazu gehört das Polynom:

$$F(X) = X^n - r_{n-1}X^{n-1} - \cdots - r_1X - r_0 \quad (2)$$

Zudem gilt, dass es eine definierende Relation (1) für $\mathbb{F}_{p^n}$ genau dann gibt, wenn $F(X) \in \mathbb{F}_p[X]$ irreduzibel ist, also wenn $F(X)$ keinen echten Teiler in $\mathbb{F}_p[X]$ besitzt.

Hierbei ist jedoch zu beachten, dass nicht für jede definierende Relation (1) α ein primitives Element von $\mathbb{F}_{p^n}$ ist. Beispielsweise kann $\mathbb{F}_{16}$ durch die Relation

$$\alpha^4 = \alpha^3 + \alpha^2 + \alpha + 1$$

definiert werden, da

$$F(X) = X^4 + X^3 + X^2 + X + 1$$

irreduzibel ist. Jedoch gilt:

$$\alpha^5 = \alpha \cdot \alpha^4 = \alpha \cdot (\alpha^3 + \alpha^2 + \alpha + 1) = \alpha^4 + \alpha^3 + \alpha^2 + \alpha = \alpha^3 + \alpha^2 + \alpha + 1 + \alpha^3 + \alpha^2 + \alpha = 1$$

Also gilt insbesondere $\alpha^5 = 1$. Demnach ist $ord(\alpha) = 5$ in $\mathbb{F}_{16}^*$, woraus folgt, dass nicht jedes $x \in \mathbb{F}_{16}^*$ durch eine α-Potenz dargestellt werden kann.

Ein Beispiel für eine Relation, mit der alle Elemente in Form von α-Potenzen dargestellt werden können, lautet:

$$\alpha^3 = \alpha + 1$$

Mittels dieser Relation kann der Körper $\mathbb{F}_8$eindeutig definiert werden. Aus dieser Relation folgt, dass die Elemente des Körpers folgendermaßen aussehen:

$$\{0, 1, \alpha, \alpha^2, \alpha + 1, \alpha^2 + \alpha, \alpha^2 + \alpha + 1, \alpha^2 + 1\}$$

Damit eine Gleichung wie $\alpha^3 = \alpha + 1$ einen Körper eindeutig identifiziert, muss das Polynom, das aus dieser Relation gebildet werden kann, irreduzibel sein. Dieses Polynom sieht für die gegebene Relation folgendermaßen aus:

$$F(X) = X^3 + X + 1$$

Das heißt, für diese Umwandlung muss lediglich die höchste Potenz auf die andere Seite der Gleichung gezogen werden und die α's in Form von X geschrieben werden. Somit kann die Relation als Polynom in Abhängigkeit von X ausgedrückt werden.

Im weiteren Verlauf werden Polynome in endlichen Körpern genauer erläutert. Zudem wird beschrieben, welche Bedingungen erfüllt werden müssen, damit ein Polynom irreduzibel ist.

2.3 Polynome über endlichen Körpern

In dem folgenden Abschnitt werden Polynome über endlichen Körpern beschrieben. Dabei wird zunächst auf die Form und die Existenz von Nullstellen eingegangen. Dann werden die verschiedenen arithmetischen Operationen von Polynomen erläutert. Dazu zählen die Addition, Multiplikation und Division. Dann wird der euklidische Algorithmus für Polynome dargestellt. Im weiteren Verlauf wird beschrieben,

wie multiplikativ-inverse Elemente in endlichen Körpern bestimmt werden können und was der Begriff Irreduzibilität für Polynome bedeutet. Schließlich wird darauf eingegangen, wie man einen endlichen Körper durch ein irreduzibles Polynom konstruieren kann.

Bei den arithmetischen Operationen, dem euklidischen Algorithmus, sowie bei der Bildung von multiplikativ-inversen Elementen wird zusätzlich eine Beispielrechnung durchgeführt, die das jeweilige Vorgehen veranschaulichen soll.

Für die folgende Definition des Grades eines Polynoms wurde sich an Bosch [1, S. 40] orientiert.

Für endliche Körper der Form $\mathbb{F}_q$, mit $q = p^n$, kann ein Polynomring $\mathbb{F}_p[X]$ über den Grundkörper $\mathbb{F}_p$ betrachtet werden. Dieser besteht aus allen Polynomen, mit der Unbekannten X, wobei die Koeffizienten lediglich Elemente aus $\mathbb{F}_p$ sein können. Nun ist ein Polynom $F(X)$ vom Grad d allgemein gegeben durch:

$$F(X) = \sum_{i=0}^{d} a_i X^i$$

Dabei gilt $F(X) \in \mathbb{F}_p[X]$ und $a_i \in \mathbb{F}_p$. Nun wird der Grad eines Polynoms definiert als:

$$deg(F(X)) = max\{i; a_i \neq 0\}$$

2.3.1 Nullstellen von Polynomen

Nach Bosch [1, S. 79–81] und Kurzweil [7, S. 67–75] lassen sich die Eigenschaften von Polynomen bezüglich Nullstellen folgendermaßen beschreiben.

Aus dem in 2.2.2 aufgeführten kleinen Satz von Fermat folgt, dass jedes Element des Grundkörpers $a \in \mathbb{F}_q$ eine Nullstelle des Polynoms $F_q(X) = X^q - X$ (wobei $F_q(X) \in \mathbb{F}_p[X]$) ist. Da $\mathbb{F}_q$ genau q Elemente hat, und $F_q(X)$ den Grad q hat, gilt:

$$F_q(X) = \prod_{a \in \mathbb{F}_q} (X - a)$$

Daraus folgt, dass $F_q(X)$ keine weiteren Nullstellen außer den Elementen aus $\mathbb{F}_q$ hat.

2.3.2 Addition von Polynomen

Nun wird basierend auf Bosch [1, S. 37–38] die Addition für Polynome beschrieben. Für den Fall, dass $n \geq m$, wird die Addition der beiden Polynome

$$F(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$$

$$G(X) = b_m X^m + b_{m-1} X^{m-1} + \cdots + b_1 X + b_0$$

wobei gilt $F(X), G(X) \in \mathbb{F}_p[X]$ und $a_i, b_i \in \mathbb{F}_p$ folgendermaßen definiert:

$$F(X) + G(X) = a_n X^n + \cdots + a_{m+1} X^{m+1} + (a_m + b_m) X^m + \cdots + (a_1 + b_1) X + a_0 + b_0$$

(Für den Fall, dass $m \geq n$ müssen lediglich die Indizes von m und n vertauscht werden.)

Beispielrechnung

Im Zuge der Beispielrechnung für die Addition von Polynomen werden die beiden Polynome

$$F(X) = X^4 + X^3 + X, G(X) = X^3 + X^2 + X + 1$$

miteinander addiert, wobei $F(X), G(X) \in \mathbb{F}_2[X]$. Da dieser Polynomring in der Charakteristik zwei liegt, gelten für die Addition der Koeffizienten die Rechenregeln der binären Addition. Nach Definition lautet die Summe der beiden Polynome:

$$F(X) + G(X) = X^4 + (1+1)X^3 + X^2 + (1+1)X + 1$$

Da nun laut der binären Addition gilt $1 + 1 = 0$ kann das Ergebnis noch vereinfacht werden:

$$F(X) + G(X) = X^4 + X^2 + 1$$

2.3.3 Multiplikation von Polynomen

Auch die Multiplikation von Polynomen folgt Bosch [1, S. 37–38]. Und zwar ist die Multiplikation der beiden Polynome

$$F(X) = a_n X^n + a_{n-1} X^{n-1} + \cdots + a_1 X + a_0$$

$$G(X) = b_m X^m + b_{m-1} X^{m-1} + \cdots + b_1 X + b_0$$

wobei gilt $F(X), G(X) \in \mathbb{F}_p[X]$ und $a_i, b_i \in \mathbb{F}_p$ folgendermaßen definiert:

$$F(X) \cdot G(X) = \sum_{l=0}^{n+m} c_l X^l$$

mit

$$c_l = \sum_{i=0}^{l} a_i b_{l-i}$$

Dadurch wird $\mathbb{F}_p[X]$ ein Ring, wobei das neutrale Element der Addition das Nullpolynom $N(X) = 0$ ist, und das neutrale Element der Multiplikation das Einspolynom $E(X) = 1$ ist.

Beispielrechnung

Im Zuge der Beispielrechnung für die Multiplikation von Polynomen werden die beiden Polynome

$$F(X) = X^2 + X, G(X) = X + 1$$

miteinander multipliziert, wobei $F(X), G(X) \in \mathbb{F}_2[X]$. Da dieser Polynomring in der Charakteristik zwei liegt, gelten für die Addition und die Multiplikation der Koeffizienten die Rechenregeln der binären Addition sowie die Rechenregeln der binären Multiplikation. Laut Definition lautet das Produkt der beiden Polynome:

$$F(X) \cdot G(X) = c_0 + c_1 X + c_2 X^2 + c_3 X^3 \quad (1)$$

Nun müssen noch die Koeffizienten $c_i \in \mathbb{F}_2$ bestimmt werden. Diese lauten nach Definition folgendermaßen:

$$c_0 = \sum_{i=0}^{0} a_i b_{0-i} = a_0 b_0 = 1 \cdot 0 = 0$$

$$c_1 = \sum_{i=0}^{1} a_i b_{1-i} = a_0 b_1 + a_1 b_0 = 0 \cdot 1 + 1 \cdot 1 = 1$$

$$c_2 = \sum_{i=0}^{2} a_i b_{2-i} = a_0 b_2 + a_1 b_1 + a_2 b_0 = 0 \cdot 0 + 1 \cdot 1 + 1 \cdot 1 = 1 + 1 = 0$$

$$c_3 = \sum_{i=0}^{3} a_i b_{3-i} = a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 = 0 \cdot 0 + 1 \cdot 0 + 1 \cdot 1 + 0 \cdot 1 = 1$$

Setzen wir nun diese Koeffizienten in das Produkt aus (1) ein, erhalten wir:

$$F(X) \cdot G(X) = X^3 + X$$

2.3.4 Polynomdivision

Die Polynomdivision wird im Folgenden anhand Kurzweil [7, S. 26] und Lang [8, S. 41–42] erläutert. Dabei ist die Polynomdivision ein Verfahren, bei dem ein Polynom $F(X)$ durch ein anderes Polynom $G(X)$ geteilt wird. Das Verfahren der Polynomdivision ähnelt der schriftlichen Division für natürliche Zahlen. Dieses Verfahren erfolgt schrittweise.

Dabei muss zuerst der höchste Exponent des Dividenden durch den höchsten Exponenten des Divisors geteilt werden. Dieses Ergebnis wird mit dem gesamten Divisor multipliziert, wobei das Ergebnis dieser Multiplikation wiederum vom Dividenden subtrahiert wird. Diese Differenz stellt den neuen Dividenden dar und dann wird dieser Ablauf so lange wiederholt, bis der Grad des verbleibenden Polynoms kleiner als der Grad des Divisors ist. Dieses verbleibende Polynom stellt dann den Rest dar, der bei dieser Polynomdivision übrig bleibt.

Allgemein kann die Polynomdivision durch folgende Gleichung dargestellt werden

$$F(X) = q(X) \cdot G(X) + r(X)$$

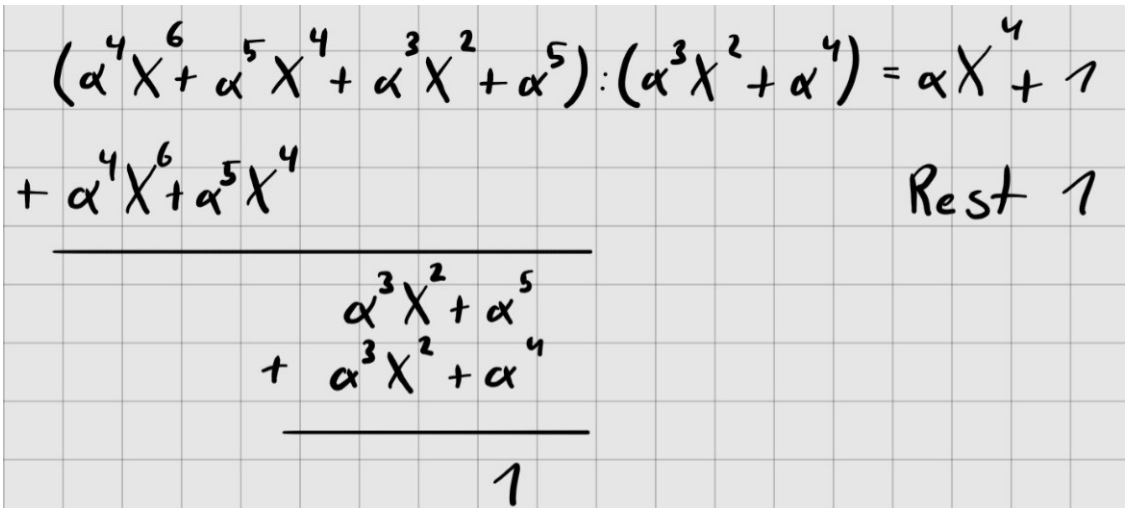
wobei $q(X)$ das Ergebnis der Polynomdivision darstellt und $r(X)$ den übrig bleibenden Rest beschreibt, wobei gilt, dass der Grad von $r(X)$ immer kleiner ist als der Grad von $G(X)$.

Beispielrechnung

Die Beispielrechnung der Polynomdivision erfolgt ebenfalls in dem endlichen Körper $\mathbb{F}_{2^3}$, dessen acht Elemente in 2.2.3 aufgeführt sind. Die definierende Relation für diesen Körper lautet:

$$\alpha^3 = \alpha + 1$$

In dieser Beispielrechnung wird nun die Polynomdivision auf die beiden Polynome $F(X) = \alpha^4 X^6 + \alpha^5 X^4 + \alpha^3 X^2 + \alpha^5$ und $G(X) = \alpha^3 X^2 + \alpha^4$ angewendet.



Aus dieser Berechnung folgt nun:

$$F(X) = (\alpha X^4 + 1) \cdot G(X) + 1$$

Demnach stellt $(\alpha X^4 + 1)$ das Ergebnis der Polynomdivision dar, wobei 1 als Rest übrig bleibt.

2.3.5 Euklidischer Algorithmus

In diesem Abschnitt wird der euklidische Algorithmus für Polynome, basierend auf Kurzweil [7, S. 57–63] und Lang [8, S. 120–123], beschrieben.

Der euklidische Algorithmus wird verwendet, um den größten gemeinsamen Teiler (kurz ggT) zu finden. Sollte dieser ggT gleich eins sein, bedeutet das, dass die beiden Zahlen teilerfremd sind, also haben sie keinen gemeinsamen, nicht-trivialen Teiler.

Der euklidische Algorithmus, für natürliche Zahlen, funktioniert so, dass man die größere der beiden Zahlen durch die kleinere mit Rest r_1 dividiert. Sollte dabei der Rest ungleich null sein, wird im nächsten Schritt die kleinere Zahl durch den Rest r_1 dividiert, wobei r_2 den Rest dieser Division darstellt. Sollte dieser Rest wiederum ungleich null sein, muss nun r_1 durch r_2 mit Rest dividiert werden. Dieser Prozess wird so lange wiederholt, bis die Division aufgeht, das heißt, bis der Rest der Division null ergibt. Dann ist der letzte Divisor der ggT der beiden Ausgangszahlen. Der Algorithmus terminiert immer, da der Rest immer eine nicht-negative ganze Zahl ist und in jedem Schritt kleiner ist als im vorherigen, also notwendig irgendwann mal 0 sein muss.

Anschließend führt man eine Rückrechnung durch, sodass der ggT als Linearkombination aus den beiden ursprünglichen Zahlen geschrieben werden kann. Beispielsweise lauten die Ursprungszahlen a und b, dann kann der ggT mittels $ggT(a,b) = c_a * a + c_b * b$ dargestellt werden. Die Vorfaktoren c_a und c_b der beiden ursprünglichen Zahlen werden Bézout-Koeffizienten genannt.

Der euklidische Algorithmus für Polynome funktioniert analog zu den beschriebenen Verfahren, mit dem Unterschied, dass mittels Polynomdivision gerechnet werden muss, anstatt einfacher Division mit Rest für die natürlichen Zahlen. Der Algorithmus terminiert auch in diesem Fall immer, weil das Restpolynom in jedem Schritt einen Grad hat, der kleiner ist als der des Divisors.

Allgemein kann das Ergebnis des euklidischen Algorithmus folgendermaßen dargestellt werden:

$$ggT(F(X), G(X)) = b_F(X) \cdot F(X) + b_G(X) \cdot G(X)$$

Dabei sind $F(X)$ und $G(X)$ die beiden Polynome, deren ggT berechnet wurde, und $b_F(X)$ und $b_G(X)$ sind die jeweiligen Bézout-Koeffizienten zu den Ausgangspolynomen.

Beispielrechnung

Die Beispielrechnung des euklidischen Algorithmus erfolgt ebenfalls in dem endlichen Körper $\mathbb{F}_{2^3}$, dessen acht Elemente in 2.2.3 aufgeführt sind. Die definierende Relation für diesen Körper lautet:

$$\alpha^3 = \alpha + 1$$

In dieser Beispielrechnung wird nun der ggT der beiden Polynome $F(X) = \alpha^3 X^4 + \alpha^5 X^2 + \alpha^2 X^2 + \alpha^5$ und $G(X) = \alpha^2 X^2 + \alpha^4$ berechnet.

Die erste Polynomdivision für die beiden Polynome $F(X)$ und $G(X)$ wurde bereits in der Beispielaufgabe der Polynomdivision berechnet (siehe 2.3.4). Diese ergab:

$$(\alpha^3 X^5 + \alpha^5 X^2 + \alpha^2 X^2 + \alpha^5) \div (\alpha^2 X^2 + \alpha^4) = (\alpha X^2 + 1) \quad Rest\ 1$$

Im nächsten Schritt wird der Divisor zum nächsten Dividend und der Rest zum neu-

en Divisor. Somit lautet die nächste Polynomdivision:

$$(\alpha^2 X^2 + \alpha^4) \div 1 = (\alpha^2 X^2 + \alpha^4) \quad Rest\ 0$$

Da nun der Rest null ist, ist die Abbruchbedingung für den euklidischen Algorithmus erreicht. Nun stellt der letzte Divisor den ggT der beiden Polynome dar. Demnach gilt nun:

$$ggT(F(X), G(X)) = 1$$

Die Rückrechnung ergibt:

$$1 = (\alpha^3 X^5 + \alpha^5 X^2 + \alpha^2 X^2 + \alpha^5) + (\alpha X^2 + 1) \cdot (\alpha^2 X^2 + \alpha^4)$$

Somit lauten die Bézout-Koeffizienten folgendermaßen: 1 für $F(X)$ und $(\alpha X^2 + 1)$ für $G(X)$.

2.3.6 Bildung von multiplikativ-inversen Elementen

Im folgenden Abschnitt wird, anhand Kurzweil [7, S. 10–11, 57–63], die Bildung von multiplikativ-inversen Elementen über endlichen Körpern behandelt.

Eine Eigenschaft eines Körpers ist es, dass jedes Element ein Inverses, bezüglich der jeweiligen Verknüpfung hat. Bei der Multiplikation muss jedoch die 0 ausgeschlossen werden, da es kein multiplikativ-inverses Element zur 0 gibt. Wird ein beliebiges Element eines Körpers mit seinem inversen Element verknüpft, so ergibt diese Verknüpfung das neutrale Element als Ergebnis.

Um ein multiplikatives Inverses von Elementen in endlichen Körpern $\mathbb{F}_{p^n}$ zu bestimmen, wird der euklidische Algorithmus für Polynome benötigt, der in 2.3.5 beschrieben wird, angewendet.

Gegeben ist ein Erweiterungskörper $\mathbb{F}_{p^n}$ von $\mathbb{F}_p$, beschrieben durch ein Minimalpolynom $G(X)$ (mit zugehörigem α). Für ein $a \in \mathbb{F}_{p^n} \setminus \{0\}$ schreibe

$$a = a_{n-1}\alpha^{n-1} + \cdots + a_1\alpha + a_0$$

mit $a_i \in \mathbb{F}_p$ und setze

$$F(X) = a_{n-1}X^{n-1} + \cdots + a_1 X + a_0$$

Da $G(X)$ irreduzibel (vom Grad n) ist, gilt:

$$ggT(F(X), G(X)) = 1$$

Schreibe

$$1 = b_F(X) \cdot F(X) + b_G(X) \cdot G(X)$$

dann gilt $mod\ G(X)$:

$$1 = b_F(X) \cdot F(X) \quad mod\ G(X)$$

Nach einsetzen von α für X gilt:

$$1 = b_F(\alpha) \cdot F(\alpha)$$

Da $F(\alpha) = a$ gilt nun:

$$1 = b_F(\alpha) \cdot a$$

und daher gilt

$$a^{-1} = b_F(\alpha)$$

Demnach stellt nun $b_F(\alpha)$ das multiplikative Inverse zu dem Element a dar.

Beispielrechnung

Die Beispielrechnung des euklidischen Algorithmus erfolgt ebenfalls in dem endlichen Körper $\mathbb{F}_{2^3}$, dessen acht Elemente in 2.2.3 aufgeführt sind. Die definierende Relation für diesen Körper lautet:

$$\alpha^3 = \alpha + 1$$

In dieser Beispielrechnung wird nun das multiplikative Inverse zu dem Element $a = \alpha^2 + \alpha$ gesucht. Dazu gehört das Polynom $F(X) = X^2 + X$ gesucht. Durch die definierende Relation ist das Minimalpolynom $G(X) = X^3 + X + 1$ gegeben. Wird der euklidische Algorithmus auf die beiden Polynome $F(X)$ und $G(X)$ angewendet, so erhält man:

$$1 = (X + 1) \cdot F(X) + (1) \cdot G(X)$$

Diese Gleichung wird nun noch modulo dem irreduziblen Polynom $G(X)$ gerechnet:

$$1 \equiv (X + 1) \cdot F(X) \quad mod\ G(X)$$

Daraus folgt, dass $b_F(\alpha) = \alpha+1$ das multiplikative Inverse zu dem Element $a = \alpha^2+\alpha$ in dem endlichen Körper $\mathbb{F}_{2^3}$ ist.

2.3.7 Irreduzibilität von Polynomen

Der folgende Abschnitt behandelt die Eigenschaft der Irreduzibilität von Polynomen, wobei wir uns an Bosch [1, S. 37–41, 79–81, 90–92], Kunz [6, S. 56–58] und Kurzweil [7, S. 141–142] orientieren.

Ein Polynom $F(X) \in \mathbb{F}_q[X]$ mit $deg(F) \geq 1$ heißt irreduzibel über $\mathbb{F}_q$, wenn es keine Faktorisierung der Form

$$F(X) = g(X) \cdot h(X)$$

gibt, wobei zum einen $g(X), h(X) \in \mathbb{F}_q[X]$ und zum anderen $1 \leq deg(g), deg(h) \leq deg(F)$ gilt.

Damit ein Polynom $F(X) \in \mathbb{F}_p[X]$ ein irreduzibles Polynom sein kann, muss es folgende Bedingungen erfüllen. Zum einen darf $F(X)$ nicht das Nullpolynom sein und der Grad des Polynoms $F(X)$ muss mindestens 1 sein. Zum anderen darf

$$F(X) = g(X) \cdot h(X)$$

mit $g(X), h(X) \in \mathbb{F}_p[X]$ nur dann gelten, wenn $g(X)$ oder $h(X)$ eine Konstante aus $\mathbb{F}_q$ ist. Sollte eines dieser Polynome keine Konstante und somit ein nicht-triviales Polynom sein, bedeutet das, dass $F(X)$ teilbar durch dieses nicht-triviale Polynom ist, womit es nicht irreduzibel ist.

Wenn

$$F(X) = X^n + r_{n-1} \cdot X^{n-1} + \cdots + r_1 \cdot X + r_0$$

ein irreduzibles Polynom aus $\mathbb{F}_p[X]$ ist, dann ist $F(X)$ ein Minimalpolynom von $\mathbb{F}_q$, mit $q = p^n$. Demnach definiert die Relation

$$\alpha^n = -r_{n-1} \cdot \alpha^{n-1} - \cdots - r_1 \cdot \alpha - r_0$$

den Körper $\mathbb{F}_q$, mit $q = p^n$.

Irreduzible Polynome der Charakteristik zwei berechnen

In diesem Abschnitt werden irreduzible Polynome der Charakteristik zwei bis zum Grad 4 ausgearbeitet. Ein Polynom $F(X)$ ist in der Charakteristik zwei genau dann irreduzibel, wenn gilt:

1. Es keine Nullstellen über dem Grundkörper $\mathbb{F}_2$ hat. Das bedeutet, dass zum

einen der konstante Term $r_0 = 1$ ist und zum anderen die Anzahl der auftretenden Monome ungerade sein muss.

2. Es wird nicht von einem irreduziblem Polynom, vom Grad mindestens 2 und höchstens $\left\lfloor \frac{deg(F)}{2} \right\rfloor$ geteilt.

Angefangen bei Polynomen vom Grad 2. Hierfür kommt nur ein einziges Polynom in Frage, und zwar das Polynom $F(X) = X^2 + X + 1$, da es das einzige Polynom vom Grad 2 ist, das eine ungerade Anzahl Monome hat und bei dem $r_0 = 1$ ist. Da $\left\lfloor \frac{deg(F)}{2} \right\rfloor = 1$ und $1 < 2$ muss auf die zweite Bedingung nicht überprüft werden.

Für Polynome vom Grad 3 kommen, aufgrund der Anzahl der Monome und der Bedingung, dass $r_0 = 1$ sein muss, lediglich die beiden Polynome $G_1(X) = X^3 + X^2 + 1$ und $G_2(X) = X^3 + X + 1$ in Frage. Da diese beiden Polynome nicht ohne Rest durch F(X) teilbar sind, bedeutet das, dass auch diese beiden Polynome irreduzibel sind. Auch hierbei gilt $\left\lfloor \frac{deg(F)}{2} \right\rfloor = 1$.

Für Polynome vom Grad 4 kommen, aufgrund der Anzahl der Monome und der Bedingung, dass $r_0 = 1$ sein muss, folgende Polynome in Frage:

$$H_1(X) = X^4 + X^3 + X^2 + X + 1, H_2(X) = X^4 + X^3 + 1$$

$$H_3(X) = X^4 + X^2 + 1, H_4(X) = X^4 + X + 1$$

Da $H_3(X) = X^4 + X^2 + 1 = (X^2 + X + 1)^2$ ist $H_3(X)$ also durch $F(X)$ ohne Rest teilbar. Daher ist $H_3(X)$ nicht irreduzibel. Jedoch sind die anderen drei Polynome irreduzibel, da sie jeweils nicht durch das Polynom $F(X)$ ohne Rest teilbar sind.

Aus diesen Berechnungen für Polynome bis zum Grad 4 wird das generelle Vorgehen klar, wie ein Polynom $F(X)$ auf Irreduzibilität überprüft werden kann. Als Erstes wird das Polynom auf ungerade Monom-Anzahl und darauf, dass $r_0 = 1$ ist, geprüft. Und dann muss geprüft werden, ob es durch ein irreduzibles Polynom vom Grad 2 bis zum Grad $\left\lfloor \frac{deg(F)}{2} \right\rfloor$ ohne Rest teilbar ist.

2.3.8 Konstruktion von endlichen Körpern

Auf Grundlage von Bosch [1, S. 171–173] und Kunz [6, S. 185–186] wird im Folgenden die Konstruktion von endlichen Körpern mittels eines irreduziblen Polynoms beschrieben.

Mittels der Notation

$$\mathbb{F}_q = \mathbb{F}_p[X]/F(X)$$

mit $q = p^n$ können endliche Körper $\mathbb{F}_q$ mit $q = p^n$ Elementen konstruiert werden. Dabei ist p eine Primzahl und n eine beliebige positive ganze Zahl. Zudem ist $\mathbb{F}_p[X]$ ein Polynomring über dem Grundkörper $\mathbb{F}_p$ und $F(X)$ ein irreduzibles Polynom vom Grad n.

Entscheidend für die Konstruktion des endlichen Körpers $\mathbb{F}_q$ ist, dass das Polynom $F(X)$ irreduzibel, bzw. ein Minimalpolynom ist. Dabei hat $F(X)$ den Grad n. Denn nur wenn $F(X)$ irreduzibel ist, stellt $\mathbb{F}_p[X]/F(X)$ einen endlichen Körper dar.

2.4 Cantor-Zassenhaus-Algorithmus

In diesem Abschnitt wird der Cantor-Zassenhaus-Algorithmus beschrieben, basierend auf der Arbeit der gleichnamigen Autoren [2]. Anschließend folgt eine Beispielrechnung, um diesen Algorithmus zu veranschaulichen.

Der Cantor-Zassenhaus-Algorithmus dient dazu, ein Polynom vom Grad n der Form

$$F(X) = X^n + r_{n-1}X^{n-1} + \cdots + r_1X + r_0 \quad \in \mathbb{F}_p[X]$$

auf Irreduzibilität zu überprüfen.

Sollte $F(X)$ nicht irreduzibel sein, bedeutet das, dass $F(X)$ einen irreduziblen, nichttrivialen Teiler $G(X)$ vom Grad m hat, wobei gilt: $0 < m \leq \frac{n}{2}$. Das Polynom $G(X)$ hat dann folgende Form:

$$G(X) = X^m + s_{m-1}X^{m-1} + \cdots + s_1X + s_0$$

Da das Polynom $G(X)$ nun irreduzibel ist, definiert dieses Polynom den endlichen Körper $\mathbb{F}_{p^m}$ mit der zugehörigen definierenden Relation:

$$\alpha^m = -s_{m-1}\alpha^{m-1} - \cdots - s_1\alpha - s_0$$

Dann gilt, dass α eine Nullstelle von $G(X)$ ist. (alle Nullstellen von $G(X)$ definieren $\mathbb{F}_{p^m}$) Da alle Elemente von $\mathbb{F}_{p^m}$ Nullstellen des Polynoms

$$F_{p^m}(X) = X^{p^m} - X$$

sind, gilt:

$$G(X) \mid (X^{p^m} - X)$$

Also teilt das Polynom $G(X)$ das Polynom $F_{p^m}(X)$. Daraus folgt:

$$G(X) \mid ggT(F(X), X^{p^m} - X)$$

Das bedeutet, dass $G(X)$ auch den ggT von $F(X)$ und $F_{p^m}(X)$ teilt. Damit gilt, dass $F(X)$ nicht irreduzibel ist.

Daraus folgt, dass ein beliebiges Polynom $F(X)$, vom Grad n, genau dann nicht irreduzibel ist, wenn Folgendes gilt:

$$\exists m \in \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor\} : (ggT(F(X), X^{p^m} - X) \neq 1)$$

Das heißt, sobald ein Polynom der Form $X^{p^m} - X$ gefunden wird, dessen ggT mit $F(X)$ ungleich eins ist, so ist $F(X)$ nicht irreduzibel. m kann dabei eine Zahl von 1 bis $\left\lfloor \frac{n}{2} \right\rfloor$ sein.

Andersrum ist ein beliebiges Polynom $F(X)$, vom Grad n, genau dann irreduzibel, wenn Folgendes gilt:

$$\forall m \in \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor\} : (ggT(F(X), X^{p^m} - X) = 1$$

Das heißt, wenn der ggT von $F(X)$ mit allen Polynomen der Form $X^{p^m} - X$ gleich 1 ist, dann ist $F(X)$ irreduzibel. Auch hierbei kann m eine Zahl von 1 bis $\left\lfloor \frac{n}{2} \right\rfloor$ sein.

Beispielrechnung

In der Beispielrechnung des Cantor-Zassenhaus-Algorithmus wird geprüft, ob das Polynom $F(X) = X^4 + X + 1 \in \mathbb{F}_2[X]$ irreduzibel ist. Dabei beträgt nun $p = 2$ und $n = 4$. Demnach muss nun der ggT von $F(X)$ und Polynomen der Form $X^{2^m} + X$ berechnet werden, wobei $m \in \{1, 2\}$. Da diese Beispielrechnung in der Charakteristik zwei stattfindet, (da $p = 2$) ist die Addition und die Subtraktion dasselbe, weswegen in dem Polynom das $-$ durch ein $+$ ersetzt werden kann.

$$ggT(F(X), X^{2^1} + X) = ggT(F(X), X^2 + X) = 1$$

$$ggT(F(X), X^{2^2} + X) = ggT(F(X), X^4 + X) = 1$$

Da diese beiden ggT's gleich 1 sind, gilt nun:

$$\forall m \in \{1, \ldots, \left\lfloor \frac{n}{2} \right\rfloor\} : (ggT(F(X), X^{p^m} - X) = 1$$

Daraus folgt, dass das Polynom $F(X) = X^4 + X + 1$ irreduzibel ist.

3 Implementierung

In diesem Kapitel wird die Implementierung der Arithmetik endlicher Körper der Charakteristik zwei in Python erläutert. Der für diese Studienarbeit entwickelte Code befindet sich im Anhang: 'Quellcode'. Dazu zählt die Polynomdarstellung, die Implementierung der verschiedenen Rechenoperationen sowie weitere Algorithmen für den Umgang mit Polynomen.

Für die Funktionen der arithmetischen Operationen sowie für die weiteren Algorithmen für Polynome werden zusätzlich Beispielaufgaben aufgeführt. Das soll dazu dienen, dass nach der Erklärung der Implementierung des jeweiligen Algorithmus dessen Funktionsweise anhand eines Beispiels genauer erläutert wird.

3.1 Polynomdarstellung

Um in der Programmiersprache Python Polynome darzustellen, wird die Datenstruktur Tupel verwendet. In einem Tupel können mehrere Inhalte unveränderlich gespeichert werden. Da in der Charakteristik zwei lediglich Zahlen aus dem Binärsystem Vorfaktoren der verschiedenen Monome sein können, werden die Tupel nur mit Nullen und Einsen befüllt. Dadurch wird bestimmt, ob ein bestimmtes Monom in dem Polynom vorkommt (in diesem Fall mit einer 1 an der entsprechenden Stelle dargestellt) oder ob das Monom nicht in dem Polynom vertreten ist (durch eine 0 dargestellt).

In diesen Tupeln steht das Monom mit dem höchsten Exponenten an der ersten Stelle (entspricht der Stelle ganz links) und die Exponenten nehmen dann schrittweise nach rechts ab, sodass die letzte Stelle (ganz rechts) dem Monom mit dem Exponenten 0 entspricht. Diese Darstellung durch Tupel vereinfacht den Umgang mit Polynomen, da die ausführliche mathematische Schreibweise zusätzliche, unnötige Informationen wie die Variable X oder das $+$-Zeichen enthält. Für den Umgang mit Polynomen ist jedoch nur wichtig, welcher Vorfaktor an welcher Stelle des Polynoms steht, und diese beiden Informationen sind in der Schreibweise als binäre Tupel enthalten, wodurch sich diese für das Arbeiten mit Polynomen anbietet.

```python
def tuple_to_polynom(F):

    if all(elem == 0 for elem in F):
        return 0

    cases = {0: "1", 1: "X"}

    L = [
        cases.get(i, f"X^{i}")
        for i, coeff in enumerate(reversed(F))
        if coeff == 1
    ]

    return "(" + " + ".join(reversed(L)) + ")"
```

Um solch ein binäres Tupel in Python wieder in die mathematische Schreibweise eines Polynoms mit Vorfaktor, Variable X und Exponent für jedes Monom darzustellen, wurde die Funktion *"tuple_to_polynom"* erstellt. Wie der Name schon sagt, bekommt die Funktion ein Tupel als Funktionsparameter und gibt einen String zurück, der das vollständige Polynom umfasst.

Sollte das Tupel lediglich aus Nullen bestehen, gibt die Funktion 0 zurück, da das leere Tupel der Null in der mathematischen Schreibweise entspricht.

Ansonsten iteriert die Funktion über die einzelnen Elemente des Tupels und bestimmt daraus einen Teilstring. Bei der Erstellung dieser Teilstrings gibt es zwei verschiedene Fälle. Bei dem letzten Monom eines Tupels, also dem Monom ganz rechts, wird der Teilstring lediglich eine 1, da das letzte Monom den Exponenten null hat, und alles hoch null gleich eins ergibt. Für alle anderen Fälle, bei denen der Exponent größer als null ist, wird der Teilstring zu X^n, wobei n dem Exponenten des zugehörigen Monoms entspricht. Im Zuge dieser Funktion wird das Tupel vor der Erstellung des Lösungsstrings einmal umgekehrt, sodass nun das letzte Monom an erster Stelle steht, das vorletzte an zweiter Stelle steht, und so weiter. Dadurch gewinnt man den Vorteil, dass jetzt die Position des Monoms in dem Tupel gleich dem Exponenten des Monoms entspricht. Da bei Tupeln, ähnlich wie bei Arrays, das Zählen bei null anfängt, steht nach der Umkehrung das erste Monom (was ursprünglich das Letzte war) an der Position null in dem Tupel, was genau dem Exponenten dieses Monoms entspricht. All diese einzelnen Teilstrings werden dann aneinandergereiht, wobei zwischendrin immer ein $+$ beigefügt wird. So erhält man schließlich einen String, der das vollständige Polynom enthält, wie man es auch in der mathematischen Schreibweise aufschreiben würde.

Anschließend werden um den Ergebnisstring noch Klammern gesetzt. Dies dient

lediglich der Lesbarkeit der Tupel.

Diese Funktion, die aus einem Tupel wieder ein Polynom macht, erfüllt keinen direkten Zweck, der für weitere Operationen wichtig ist, sondern diese Funktion dient lediglich dazu, aus einem Tupel eine für den Anwender lesbare Darstellung zu bieten, da die Tupel, vor allem bei steigender Länge, unübersichtlich werden können.

3.2 Hilfsfunktionen

Während der Umsetzung werden für einige der implementierten Operationen Hilfsfunktionen benötigt. Im Folgenden werden diese Funktionen genauer erläutert.

3.2.1 Wegschneiden von Nullen

Um führende Nullen von Polynomen, bzw. von binären Tupeln wegzuschneiden, wurde die Funktion *"cut_zeros_left"* implementiert, die dafür sorgt, dass das gegebene Tupel mit der ersten Eins, die in dem Tupel vorhanden ist, beginnt und alle vorherigen Stellen abgeschnitten werden.

```
1 def cut_zeros_left(F):
2
3     while F and F[0] == 0:
4         F = F[1:]
5
6     if F == ():
7         return (0,)
8
9     return F
```

Um diese führenden Nullen abzuschneiden, wird über das binäre Tupel iteriert und überprüft, ob die erste Stelle eine Null ist. Sollte diese Bedingung wahr sein, wird die erste Stelle abgeschnitten. Dieser Vorgang wird so lange wiederholt, bis die erste Stelle eine Eins ist.

Für den Sonderfall, dass ein leeres Tupel in die Funktion gegeben wird, muss eine zusätzliche Bedingung eingeführt werden, da in einem leeren Tupel keine Eins gefunden werden kann, und somit eine Endlosschleife entstehen würde. Dieser Fall kann einfach abgefangen werden, indem in der Schleifenbedingung geprüft wird, ob das Tupel leer ist, und die Schleife nur durchlaufen wird, sollte das Tupel nicht leer sein.

3.2.2 Erste Einser-Stelle finden

Die Funktion *"find_pos_of_first_one"* dient dazu, die erste Stelle eines Polynoms, von links aus, zu finden, die mit einer Eins befüllt ist. Diese Funktion wird im späteren Verlauf benötigt, um den Grad einer Funktion ermitteln zu können.

```
1 def find_pos_of_first_one(F):
2
3     for i in range(len(F)):
4         if F[i] == 1:
5             return i
6
7     return None
```

Um diese Stelle bestimmen zu können, wird über die Stellen des gegebenen Tupels iteriert. Sobald eine Eins gefunden wurde, wird der Index zurückgegeben, bei dem sich die Iteration zu dem Zeitpunkt befindet. Dieser Index stellt dann die Position der ersten Eins des Polynoms dar.

Sollte keine Eins gefunden werden, wird von der Funktion 0 zurückgegeben. Demnach ist der Grad des Polynoms gleich Null, da es lediglich aus Nullen besteht.

3.2.3 Grad eines Polynoms bestimmen

```
1 def find_degree(F):
2
3     FirstOne = find_pos_of_first_one(F)
4
5     return (len(F)-1) - FirstOne if FirstOne != None else 0
```

Der Grad eines Polynoms wird bestimmt durch den höchsten Exponenten, der in dem jeweiligen Polynom zu finden ist. Um nun, mittels der Funktion "find_degree", den Grad eines Polynoms bestimmen zu können, muss zunächst die erste Stelle des Polynoms ermittelt werden, die eine Eins enthält. Dies ist durch die eben aufgeführte Hilfsfunktion *"find_pos_of_first_one"* möglich.

Anhand dieser Stelle, welche das Monom mit dem höchsten Exponenten darstellt, kann jetzt der Grad des Polynoms ermittelt werden. Um den Grad ermitteln zu können, muss von der Gesamtlänge des Polynoms die Zahl, der Index der ersten Eins, abgezogen werden. Dadurch erhält man die Länge des Polynoms, wenn man von der ersten Eins aus anfängt zu zählen. Von dieser Länge muss nun noch eins

abgezogen werden, da das letzte Monom den Exponenten null hat.

Als Ergebnis erhält man den höchsten Exponenten, der in dem Polynom zu finden ist, und dies entspricht dem Grad des Polynoms. Sollte das Polynom jedoch nur aus Nullen bestehen, so beträgt der Grad null.

3.2.4 Polynom auf Nullstellen prüfen

Mittels der Funktion *"has_root"* ist es möglich, ein Polynom darauf zu prüfen, ob es eine Nullstelle im Grundkörper $\mathbb{F}_2$ hat. Ein Polynom hat genau dann eine Nullstelle, wenn man in dem Polynom für die Variable X ein Element aus dem Grundkörper $\mathbb{F}_2$ einsetzen kann, wodurch das Polynom zu Null wird. In der Charakteristik zwei, das heißt in Körpern, die auf dem Grundkörper $\mathbb{F}_2$ basieren, gibt es hierbei lediglich die beiden Elemente Null und Eins, die in das Polynom eingesetzt werden können.

Die Null stellt genau dann eine Nullstelle dar, wenn in dem Polynom ein konstanter Faktor auftritt. Denn in allen anderen Monomen des Polynoms wird eine Null eingesetzt, wodurch sich all diese zu Null addieren. Dadurch muss nur das niedrigste Monom betrachtet werden, da dies durch den Exponenten von Null nicht von der für die Variable eingesetzten Zahl abhängt. Demnach gilt: Wenn die letzte Stelle des Tupels eine Eins ist, stellt die Null keine Nullstelle dar, und wenn die letzte Stelle des Tupels eine Null ist, ist die Null eine Nullstelle des Polynoms.

Für die Eins hingegen müssen andere Bedingungen überprüft werden. Hierfür ist insbesondere die folgende Rechenregel wichtig, die aufgrund des Binärsystems gilt: $1 + 1 = 0$. Daraus ergibt sich, dass, wenn die Summe der Einser gerade ist, das Ergebnis Null wird, da sich immer zwei Einser gegenseitig auslöschen. Dadurch gilt, dass die Eins eine Nullstelle ist, sollte die Anzahl der Einser im binären Tupel gerade sein. Somit ist bei ungerader Anzahl der Einser im Tupel die Eins keine Nullstelle.

```python
def has_root(F):

    if F[-1] == 0:
        return True

    counter = F.count(1)

    return (counter % 2) == 0
```

In der Implementierung wurden die eben aufgeführten Bedingungen für Nullstellen, bezüglich der Elemente des Grundkörpers $\mathbb{F}_2$, folgendermaßen umgesetzt. Da für

die Null lediglich die letzte Stelle entscheidend ist, ob sie eine Nullstelle für das Polynom ist, wird überprüft, ob die letzte Stelle eine Null ist. Sollte diese Bedingung stimmen, gibt die Funktion *'True'* zurück, da in diesem Fall eine Nullstelle vorliegt.

Um die Eins zu überprüfen, wird die Anzahl der vorkommenden Einsen gezählt. Sollte diese Anzahl gerade sein, gibt es eine Nullstelle und somit gibt die Funktion *'True'* zurück. Sollten diese beiden aufgeführten Bedingungen nicht eingetreten sein, hat das Polynom keine Nullstelle und die Funktion gibt *'False'* zurück.

3.2.5 Tupel nach Stellen aufteilen

Die Hilfsfunktion *"create_splitted_tuples"* bewirkt, dass ein Polynom in verschiedene Polynome aufgeteilt werden kann, sodass die Polynome nach der Aufspaltung lediglich eine Null besitzen. D.h. das ursprüngliche Tupel wird in so viele Tupel aufgespalten, wie die Anzahl Einsen des Tupels beträgt. Dabei behalten die aufgespaltenen Tupel die Information, wo die Einsen des ursprünglichen Tupels standen.

Beispielsweise wird das Tupel $(1, 0, 1, 1)$ in drei verschiedene Tupel aufgeteilt, da in dem ursprünglichen Tupel drei Einsen vorhanden sind. Sodass nun die Information über die Stellen, an denen sich die Einsen befinden, nicht verloren geht, werden die Einsen in dem neuen Tupel genau die gleiche Stelle haben, wie im ursprünglichen Tupel. Nun werden aus dem ursprünglichen Tupel $(1, 0, 1, 1)$ drei Tupel, mit jeweils einer Eins, an ihrer entsprechenden Stelle, erstellt. Diese sehen dann folgendermaßen aus: $(1, 0, 0, 0)$, $(0, 0, 1, 0)$ und $(0, 0, 0, 1)$. Dabei ist wichtig, dass alle Tupel die gleiche Länge wie das ursprüngliche Tupel haben.

```
1 def create_splitted_tuples(indizes, length):
2
3     return [tuple(1 if i == index else 0
4         for i in range(length))
5         for index in indizes]
```

Im Zuge der Implementierung werden dieser Funktion zwei Parameter mitgegeben. Zum einen eine Liste *'indizes'*, die bestimmt, an welchen Stellen später die Einsen stehen müssen, und zum anderen die Länge *'length'*, die bestimmt, wie viele Stellen die einzelnen Tupel haben müssen.

In der Funktion wird über den Funktionsparameter *'indizes'* iteriert und für jeden Index ein Tupel erstellt, das nur an der Stelle des Index eine Eins hat, und der Rest wird mit Nullen gefüllt. Diese Tupel haben die Länge des gegebenen Funktionspa-

rameters *'length'*. Die von der Funktion zurückgegebene Tupel-Liste enthält nun die aufgespaltenen Tupel, der Länge *'length'*, die jeweils nur eine Eins an der zugewiesenen Stelle haben.

Diese Hilfsfunktion wird für die Implementierung eines Algorithmus für die Multiplikation von Polynomen benötigt. Dieser Algorithmus basiert darauf, dass eines der Polynome in mehrere Polynome aufgespalten wird, sodass das Distributivgesetz der Multiplikation einfacher umgesetzt werden kann.

3.2.6 Exponent zu einer Stelle finden

Die Hilfsfunktion *"find_exponent"* dient dazu, in einem Tupel zu einer angegebenen Stelle den entsprechenden Exponenten zu finden.

```
1 def find_exponent(F, pos):
2
3     return len(F) - pos - 1
```

Das heißt, die Funktion bekommt ein Tupel sowie eine Zahl als Parameter. Die Zahl bestimmt dabei, zu welcher Stelle im Tupel der entsprechende Exponent gefunden werden soll.

Der gesuchte Exponent wird bestimmt, indem von der Länge des Tupels die Stelle des gesuchten Exponenten innerhalb des Tupels abgezogen wird. Davon muss zusätzlich noch eins abgezogen werden, da die Zählweise der Stellen null-basiert ist. Würde man beim Zählen der Stellen bei eins anfangen, müsste man das Abziehen der zusätzlichen Eins weglassen.

3.2.7 Ableitung bestimmen

Mithilfe der Funktion *"find_derivative"* kann die Ableitung eines Polynoms bestimmt werden. Die Ableitung wird im späteren Verlauf für den Algorithmus von Cantor-Zassenhaus benötigt.

```
1 def find_derivative(F):
2
3     L = [find_exponent(F, i) % 2 if F[i] == 1 else 0
4                                  for i in range(len(F))]
5
6     return cut_zeros_left(tuple(L[:-1]))
```

Zu Beginn wird über alle Stellen des Polynoms F, welches abgeleitet werden soll, iteriert und für jede Stelle einzeln die zugehörige Ableitung bestimmt. Diese Stellen werden dann jeweils in die Liste L geschrieben, wobei diese Liste am Ende die Ableitung darstellt.

Eine Stelle kann in der Ableitung nur dann eins sein, wenn der Vorfaktor im ursprünglichen Polynom eine Eins ist. Der Vorfaktor muss eins sein, da, wenn er null wäre, die Ableitung eines einzelnen Monoms folgendermaßen aussehen würde: $r \cdot 0 \cdot X^{r-1}$ (wenn das ursprüngliche Monom $0 \cdot X^r$ ist). Und aufgrund der Multiplikation mit Null im Vorfaktor fällt das Monom somit in der Ableitung weg, und an die Lösungsliste L wird eine 0 für diese Stelle angefügt.

Wenn der Vorfaktor eins ist, muss zudem noch der Exponent ungerade sein, damit das Monom in der Ableitung ungleich Null ist. Sollte der Exponent nämlich gerade sein, sieht die Ableitung eines Monoms der Form X^r (wobei r gerade ist) so aus: $r \cdot X^{r-1}$. Da jedoch r gerade ist, ist es in der Charakteristik zwei gleich Null, da jede gerade Zahl modulo zwei gleich Null ist, wodurch erneut mit Null multipliziert wird. Dadurch wird ein Monom mit geradem Exponenten ebenfalls zu 0 in der Ableitung.

Demnach wird eine Stelle in der Ableitung nur dann eins, wenn die folgenden zwei Bedingungen gelten. Zum einen muss der Vorfaktor des Monoms eins sein, und zum anderen muss der Exponent ungerade sein, sodass dieser Exponent modulo zwei gleich eins ergibt.

Nachdem nun einzeln über die Stellen des Ausgangspolynoms F iteriert wurde, und für jedes Monom die Ableitung gebildet wurde, muss nun die letzte Stelle der Lösungsliste abgeschnitten werden. Das liegt daran, dass bei der Ableitung die Exponenten immer um je eins geringer werden. Durch das Entfernen der letzten Stelle wird die Länge der Liste um eins kleiner, und somit verringern sich alle Exponenten um eins. Zudem fallen Konstanten, und somit die letzte Stelle, bei der Ableitung weg, sodass bei diesem Abschneiden der letzten Stelle keine wichtige Information verloren geht.

Schließlich wird die Liste umgewandelt, als Tupel von der Funktion zurückgegeben, wobei noch alle führenden Nullen von links, mittels der Hilfsfunktion *"cut_zeros_left"*, abgeschnitten werden.

3.3 Arithmetische Operationen

Da die Charakteristik Zwei auf dem Binärsystem basiert, verhalten sich die arithmetischen Operationen innerhalb von Körpern der Charakteristik Zwei analog zu den Berechnungen im Binärsystem. Im Folgenden wird die Implementierung der arithmetischen Rechenoperationen für binäre Tupel, welche Polynome der Charakteristik Zwei darstellen, genauer erläutert.

3.3.1 Addition von Polynomen

Bei der Addition von Polynomen in Körpern der Charakteristik zwei gelten die Rechenregeln wie bei der Addition im Binärsystem. Diese lauten folgendermaßen. $0+0=0; 0+1=1; 1+0=1; 1+1=0$.

```
1  def add_polynoms(F, G):
2
3      max_len = max(len(F), len(G))
4
5      F = (0,) * (max_len - len(F)) + F
6      G = (0,) * (max_len - len(G)) + G
7
8      return tuple([(F[i] + G[i]) % 2
9                    for i in range(max_len)])
```

Um nun diese Addition von Polynomen in Python zu implementieren, werden der Funktion *"add_polynoms"* zunächst zwei Polynome übergeben, die in Form von binären Tupeln dargestellt werden. Um diese beiden binären Tupel zu addieren, wird über die Stellen der Tupel iteriert, und jede Stelle einzeln, nach den oben gegebenen Rechenregeln, binär addiert. Die Ergebnisse werden dann schrittweise dem Lösungspolynom hinzugefügt.

Bevor die iterative Addition der Stellen der Polynome durchgeführt werden kann, muss jedoch sichergestellt werden, dass auch jeweils die passenden Stellen miteinander addiert werden. Als Voraussetzung dafür, dass immer die passenden Stellen miteinander addiert werden, gilt, dass die beiden Polynome, beziehungsweise die beiden Tupel, gleich lang sind. Um sicherzustellen, dass die beiden Tupel gleich viele Stellen haben, wird, vor der iterativen Addition, bestimmt, welches Tupel das längere der beiden ist, und diese Länge wird zwischengespeichert. Im Anschluss werden beide Tupel mit führenden Nullen von links aufgefüllt, bis beide die gespeicherte maximale Länge haben. Diese Nullen dienen lediglich zur richtigen Zuordnung der Stellen bei der Addition. Zudem ändern sie nichts an den Rechnungen, da die Zahl Null das neutrale Element in der Addition darstellt. Demnach wird durch

diese Auffüllung mit Nullen das Ergebnis der Addition nicht beeinflusst.

Schließlich wird am Ende der Funktion das Lösungspolynom in Form eines binären Tupels zurückgegeben.

Beispielrechnung:

In diesem Beispiel sind $F(X) = X^3+X$ und $G(X) = X^3+X^2$ die beiden Summanden der Addition. Diese können als binäre Tupel $F = (1,0,1,0)$ und $G(X) = (1,1,0,0)$ geschrieben werden. Um diese Tupel nun zu addieren, werden sie stellenweise addiert, wobei die Addition im Grundkörper $\mathbb{F}_2$ der *'XOR'*-Verknüpfung entspricht.

Zur Veranschaulichung werden die beiden Tupel untereinander geschrieben, sodass die Addition wie die schriftliche Addition dargestellt wird.

$$\begin{array}{r} 1010 \\ +1100 \\ \hline 0110 \end{array}$$

Bei dieser Addition sind alle vier Fälle abgedeckt, die bei der Addition von Elementen des Binärsystems auftreten können.

3.3.2 Multiplikation von Polynomen

Das Multiplizieren von zwei Polynomen funktioniert so, dass man die beiden Polynome mithilfe des Distributivgesetzes ausmultipliziert. Das bedeutet, dass jedes Monom des ersten Polynoms mit jedem Monom des zweiten Polynoms multipliziert wird.

Bei diesem Ausmultiplizieren der einzelnen Monome kann es dazu kommen, dass mehrere Monome die gleiche Potenz haben. Diese müssen dann noch miteinander addiert werden. Da diese Polynome in Körpern der Charakteristik zwei sind, muss hierbei lediglich geprüft werden, ob die Anzahl der Monome mit der gleichen Potenz gerade oder ungerade ist.

```python
def multiply_polynoms(F, G):

    L = [0] * (len(F) + len(G) - 1)

    for i, coef1 in enumerate(F):
        for j, coef2 in enumerate(G):

                L[i + j] ^= coef1 * coef2

    return cut_zeros_left(tuple(L))
```

Die Multiplikation von zwei Polynomen wird durch die Funktion *"multiply_polynoms"* ermöglicht. Hierbei wird zu Beginn eine Lösungsliste L definiert und alle Stellen mit Null initialisiert. Die Länge dieser Liste wird von den Längen der beiden Faktoren F und G bestimmt. Und zwar bekommt die Liste die Länge der Summe der beiden Faktoren. Davon muss jedoch noch eins abgezogen werden, da die Multiplikation mit dem letzten Monom, welches lediglich 1 oder 0 sein kann, keine zusätzliche Stelle für die Lösung liefert.

Wie bereits erwähnt, muss jedes Monom des einen Polynoms mit jedem Monom des anderen Polynoms multipliziert werden. Daraus folgt, dass für die Implementierung der Multiplikation von Polynomen zwei ineinander geschachtelte Schleifen benötigt werden, die jeweils über die Länge der beiden Polynome iterieren.

Innerhalb dieser beiden Schleifen wird dann die Multiplikation der einzelnen Monome sowie die Addition von Monomen gleicher Potenzen durchgeführt. Dabei wird anhand der Schleifenindizes die aktuelle Stelle des Lösungspolynoms ermittelt, und der Inhalt dieser aktuellen Stelle wird mit dem Produkt der Multiplikation der einzelnen Monome mittels einer *'XOR'*-Anweisung verknüpft. Die *'XOR'*-Verknüpfung entspricht in der Charakteristik zwei der Addition, wodurch die Multiplikation und die Addition zusammen in einer Anweisung durchgeführt werden können.

Im Anschluss wird die Lösungsliste L noch zu einem Tupel umgewandelt. Für die Zwischenspeicherung der Lösung wurde eine Liste benutzt, da Tupel in Python *'immutable'* sind, das heißt, man kann den Inhalt nicht mehr ändern. Von diesem Tupel werden anschließend noch, mittels der Hilfsfunktion *"cut_zeros_left"*, die führenden Nullen von links abgeschnitten, da dies überflüssige Stellen sind, und dieses Tupel wird dann von der Funktion zurückgegeben.

Beispielrechnung:

Für das Beispiel der Multiplikation wurden die beiden Tupel $F(X) = X^2 + 1$ und

$G(X) = X^4 + X + 1$ gewählt. Die daraus entstehenden Tupel $F = (1, 0, 1)$ und $G = (1, 0, 0, 1, 1)$ werden nun anhand des eben aufgeführten Algorithmus zur Multiplikation von Polynomen miteinander multipliziert.

Zu Beginn wird die Lösungsliste L initial mit Nullen gefüllt, bis sie die Länge $len(F) + len(G) - 1$ hat. Das entspricht in diesem Beispiel $3 + 5 - 1 = 7$. Somit ist $L = [0, 0, 0, 0, 0, 0, 0,]$ und wird im weiteren Verlauf stellenweise überschrieben.

Als nächstes wird über die drei Stellen des Tupels F iteriert, und für jede dieser Stellen wird über die fünf Stellen des Tupels G iteriert. Dabei wird für jeden Koeffizienten des Tupels F die Lösungsliste mit den Koeffizienten des Tupels G mittels der *'XOR'*-Anweisung verknüpft. Dabei beginnt man für jeden Iterationsschritt über F eine Stelle weiter rechts im Lösungstupel L. Durch diese Verschiebung wird erreicht, dass die Potenzen bei der Multiplikation passend sind.

Für die erste Stelle von F wird die Lösungsliste komplett mit G, mittels *'XOR'* verknüpft, da der aktuelle Koeffizient von F eine Eins ist. Demnach wird die Liste L folgendermaßen neu berechnet.

$$L = [0, 0, 0, 0, 0, 0, 0] \oplus [1, 0, 0, 1, 1] = [1, 0, 0, 1, 1, 0, 0]$$

Für die nächste Stelle von F muss die Lösungsliste nicht angepasst werden, da der Koeffizient der zweiten Stelle von F eine Null ist. Jedoch ist der Koeffizient der dritten Stelle von F eine Eins, weshalb L erneut mit G mittels *'XOR'* verknüpft werden muss, jedoch beginnend bei der dritten Stelle von L, da wir in der dritten Iteration über das Tupel F sind. Hierbei wird die Lösungsliste zu:

$$L = [1, 0, 0, 1, 1, 0, 0] \oplus [0, 0, 1, 0, 0, 1, 1] = [1, 0, 1, 1, 1, 1, 1]$$

Dabei wurde G um zwei Nullen links ergänzt, da die Lösungsliste in dieser Iterationsstufe erst ab der dritten Stelle mit G verknüpft werden muss. Deshalb wurde G um zwei Nullen links ergänzt, da die Null das neutrale Element in der *'XOR'*-Verknüpfung ist, und sich demnach die ersten beiden Stellen nicht ändern.

Nun wurde über alle Stellen von F iteriert, wobei die Lösungsliste folgendermaßen aussieht.

$$L = [1, 0, 1, 1, 1, 1, 1]$$

In Polynomschreibweise beträgt das Ergebnis der Multiplikation also:

$$L(X) = X^6 + X^4 + X^3 + X^2 + X + 1$$

3.3.3 Reduktion eines Produkts

Wenn man sich in einem Körper, hier der Charakteristik zwei, befindet, gibt es eine definierende Relation, die diesen Körper definiert. Dafür gilt, dass alle Elemente dieses Körpers in ihrer Darstellung bezüglich der Basis $1, \alpha, \ldots, \alpha^{d-1}$ (d der Grad des Minimalpolynoms), einen geringeren Grad in α haben müssen, als das Minimalpolynom, bzw. als die definierende Relation.

Da es bei dem eben aufgeführten Verfahren der Multiplikation von Polynomen dazu kommen kann, dass die Stellenanzahl, und damit der Grad des Lösungspolynoms nach der Multiplikation größer wird als der Grad des Minimalpolynoms zu der zugehörigen definierenden Relation, muss nach der Berechnung der Multiplikation noch eine Reduktion des Produkts mithilfe der definierenden Relation des Körpers durchgeführt werden.

```
1  def reduce_product(F, M):
2
3      Gf, Gm = find_degree(F), find_degree(M)
4
5      if Gf < Gm:
6          return (0,) * (Gm - len(F)) + F
7
8      F, M = cut_zeros_left(F)[::-1], cut_zeros_left(M)[::-1]
9
10     while len(F) >= len(M):
11
12         remaining_exponent = len(F) - 1 - Gm
13
14         temp_M = (0,) * remaining_exponent + M[:-1]
15
16         F = cut_zeros_left(add_polynoms(F[:-1], temp_M)[::-1])[::-1]
17
18     return (F + (0,) * (Gm - len(F)))[::-1]
```

Diese Reduktion funktioniert folgendermaßen. Zunächst werden der Funktion *"reduce_product"* zwei Polynome übergeben. Zum einen das Polynom F, das reduziert werden soll, und zum anderen das Minimalpolynom M, das für die definierende Relation steht.

Als erstes werden die Grade beider Polynome mittels der Hilfsfunktion *"find_degree"* ermittelt. Sollte der Grad des Polynoms F kleiner als der Grad des Polynoms M sein,

bedeutet das, dass das Polynom F nicht reduziert werden muss. Demnach kann dieses Polynom unverändert zurückgegeben werden, wobei es noch um führende Nullen ergänzt wird, damit es so viele Stellen hat, wie in dem Körper, der durch die Relation definiert wird, zugelassen sind.

Sollte der Grad des Polynoms F jedoch gleich oder höher als der Grad des Polynoms M sein, bedeutet das für das Polynom F, dass es reduziert werden muss. Dazu werden zuerst alle führenden Nullen von beiden Polynomen weggeschnitten und im Anschluss werden die beiden Polynome umgedreht. Durch diese Umkehrung der Polynome entsprechen nun die Stellen der Polynome den Exponenten der jeweiligen Monome.

Nun wird das Polynom F stellenweise mit der Relation reduziert, wobei pro Stelle, die weggeschnitten wird, der Grad des Polynoms um eins verringert wird. Dieser Vorgang wird so lange wiederholt, bis der Grad von F kleiner als der Grad des Minimalpolynoms M ist. Im Zuge dieser stellenweisen Reduktion wird zuerst der höchste Exponent des Polynoms ermittelt, indem von der Länge des Polynoms eins abgezogen wird. Von diesem höchsten Exponenten des Polynoms wird nun der Grad des Minimalpolynoms M abgezogen. Dieser Wert beschreibt nun, welcher Faktor nach dem Schritt der Vereinfachung übrig bleibt. Beispielsweise ist der Grad der Relation acht und der höchste Exponent des Polynoms F ist 13. Dann wird dieser Exponent 13 mittels der Potenzregel für gleiche Basen aufgeteilt in $X^8 * X^5$. Hierbei kann nun das Monom mit Exponent acht durch die Relation ersetzt werden.

Im weiteren Verlauf der Funktion wird nun die letzte Stelle, entsprechend dem Monom mit dem höchsten Exponenten, des Polynoms F abgeschnitten, da diese Stelle in dem aktuellen Schritt der Vereinfachung wegfällt. Anschließend wird ein temporäres Polynom erstellt, welches beschreibt, was zu F addiert werden muss. Dieses temporäre Polynom stellt den aktuellen Schritt der Vereinfachung dar, der eben beispielsweise anhand des Exponenten 13 erklärt wurde. Dazu werden dem temporären Polynom von links so viele Nullen angefügt, wie hoch der Faktor nach dem Schritt der Vereinfachung übrig bleibt. Anschließend wird das Minimalpolynom rechts an dem temporären Polynom angefügt, wobei die letzte Stelle weggelassen wird. Um zurück zu dem eben aufgeführten Beispiel zurückzukommen, entsprechen die Nullen dem Monom mit Exponent fünf, und das Minimalpolynom ohne den höchsten Exponenten stellt die Relation dar.

Anschließend wird das Polynom F mit dem eben aufgeführten temporären Polynom addiert, und somit wird ein Schritt der Vereinfachung durchgeführt. Dann werden

noch alle führenden Nullen von rechts weggeschnitten. Dazu wird das Tupel umgedreht, alle führenden Nullen von links weggeschnitten, und anschließend wird das Tupel erneut umgedreht. Diese Schritte der Vereinfachung werden so lange wiederholt, bis der Grad von F kleiner als der Grad des Minimalpolynoms M ist.

Ist diese Bedingung nun erreicht, ist die Reduktion des Polynoms F mittels der definierenden Relation des Körpers, bzw. mittels des Minimalpolynoms, vollendet. Das Ergebnis wird nun noch um führende Nullen ergänzt, bis es die maximale Länge hat, die in dem jeweiligen Körper möglich ist. Außerdem wird das Ergebnis noch umgekehrt, sodass nun das Monom mit dem höchsten Exponenten wieder an erster Stelle steht.

Beispielrechnung:

Für das Beispiel der Produkt-Reduzierung wird das Ergebnis der Multiplikation aus dem Rechenbeispiel zur Polynom-Multiplikation genommen. Dieses Polynom lautet

$$F(X) = X^6 + X^4 + X^3 + X^2 + X + 1$$

und soll nun mittels der definierenden Relation $X^5 = X^2 + 1$ reduziert werden. Aus dieser Relation ergibt sich das Minimalpolynom $M(X) = X^5 + X^2 + 1$.

Somit lauten die beiden Tupel, die für die Produkt-Reduzierung benötigt werden $F = (1, 0, 1, 1, 1, 1, 1)$ und $M = (1, 0, 0, 1, 0, 1)$. Zunächst werden die Grade der beiden Polynome, bzw. Tupel bestimmt. Dabei beträgt der Grad von F gleich 6 und der Grad von M gleich 5. Da $6 \geq 5$, ist der Grad von M nicht größer als der Grad von F und demnach muss F reduziert werden.

Als nächstes werden nun von den Tupeln F und M jeweils die führenden Nullen weggeschnitten, und anschließend werden beide Tupel umgedreht, damit die Stellen der Tupel nun den Exponenten entsprechen. Somit lauten die beiden Tupel nun: $F = (1, 1, 1, 1, 1, 0, 1)$ und $M = (1, 0, 1, 0, 0, 1)$.

Nun wird so lange iteriert, bis das Tupel F kürzer ist, als das Tupel M ist. Die Variable *"remaining_exponent"* beträgt $len(F) - 1 - Gm = 7 - 1 - 5 = 1$. Das bedeutet, dass sich das Zwischenspeicher-Tupel *'temp_M'* folgendermaßen zusammensetzt. Es beginnt mit einer Null (da *'remaining_exponent'* = 1), und daran wird das Minimalpolynom, ohne die letzte Stelle, gehängt. Da die letzte Stelle den höchsten Exponenten darstellt, bedeutet das, dass an die eine Null die Relation gehängt wurde. In

diesem Beispiel beträgt das Zwischenspeicher-Tupel nun: *temp_M* $= (0, 1, 0, 1, 0, 0)$.

Nun wird das Tupel $F = (1, 1, 1, 1, 1, 0, 1)$ angepasst, indem zu diesem Tupel, wobei die letzte Stelle weggelassen wird, das Zwischenspeicher-Tupel hinzuaddiert wird.

$$F = (1, 1, 1, 1, 1, 0) + (0, 1, 0, 1, 0, 0) = (1, 0, 1, 0, 1, 0)$$

Von diesem Tupel werden nun noch die führenden Nullen von rechts weggeschnitten. Somit gilt nun

$$F = (1, 0, 1, 0, 1)$$

Da nun die Länge von F gleich 5 ist und somit dieses Tupel kürzer ist als das Tupel M, so endet die Schleife. Damit lautet das Ergebnis der Reduktion, in umgedrehter Reihenfolge, damit der höchste Exponent wieder links steht: $F = (1, 0, 1, 0, 1)$. In Polynomschreibweise beträgt das Ergebnis der Reduktion also:

$$F(X) = X^4 + X^2 + 1$$

3.3.4 Alternativer Algorithmus für die Multiplikation

Der bereits aufgeführte Algorithmus, um Polynome zu multiplizieren, basiert darauf, dass die beiden Polynome stellenweise miteinander ausmultipliziert werden. Und anschließend wird das Lösungspolynom in einem separaten Schritt mittels der definierenden Relation bzw. dem zugehörigen Minimalpolynom reduziert, sodass das Lösungspolynom auch ein Element des zu betrachtenden Körpers (der Charakteristik zwei) ist.

Alternativ gibt es noch einen anderen Algorithmus, mit dem diese Multiplikation möglich ist. Bei diesem Algorithmus passieren die beiden Schritte des vorherigen Algorithmus, also zum einen die reine Multiplikation der beiden Polynome und zum anderen die Reduktion mittels des Minimalpolynoms, zusammen in einem Schritt.

Bei diesem Algorithmus wird einer der Polynomfaktoren schrittweise aufgeteilt, sodass die beiden Polynome nicht miteinander nach dem Distributivgesetz ausmultipliziert werden, sondern dass das eine Polynom schrittweise mit den Potenzen des anderen Polynoms multipliziert. Sollte dabei der Grad des Polynoms so groß werden, wie der Grad der Relation, wird der höchste Exponent direkt mit der Relation ersetzt, also wird die Relation zum aktuellen Stand des Lösungspolynoms addiert. Dieser Vorgang wird so lange wiederholt, bis über alle Stellen des einen Polynoms iteriert worden ist.

```python
def alternative_multiplication(F, G, M):

    if find_degree(F) >= find_degree(M) or find_degree(G) >= find_degree(M):
        return False

    G = (0,) * (len(M) - (len(G) + 1)) + G

    R = cut_zeros_left(M)[1:]

    pos_of_ones = [index for index, value in enumerate(F) if value == 1]

    tuple_list = create_splitted_tuples(pos_of_ones, len(F))

    L = ()

    for i in range(len(tuple_list)):

        if tuple_list[i][-1] == 1:
            L = add_polynoms(L,G)
        else:
            distance = len(F) - 2 - tuple_list[i].index(1)

            iteration_count = distance + 1

            temp = G

            for j in range(iteration_count):

                temp = add_polynoms(temp[1:] + (0,), temp[0] * R)

            L = add_polynoms(L, temp)

    return L
```

In der Implementierung werden der Funktion *"alternative_multiplication"* zunächst drei Funktionsparameter übergeben. Dazu zählen die beiden Polynomfaktoren F und G sowie das Minimalpolynom M, welches die definierende Relation des Körpers darstellt.

Im ersten Schritt überprüft die Funktion, ob die zwei Polynomfaktoren überhaupt zulässig sind. Dazu müssen die Grade der beiden Polynome kleiner als der Grad des Minimalpolynoms sein. Wenn der Grad von einem der beiden Polynome nicht kleiner als der Grad des Minimalpolynoms ist, bedeutet das, dass dieses Polynom dann kein Element des Körpers ist, welcher durch das Minimalpolynom, bzw. die darauf basierende definierende Relation, bestimmt wird. Dazu wird das Tupel G um führende Nullen ergänzt, bis es die Länge $len(M) - 1$ hat. Dies ist für den weiteren Algorithmus von Bedeutung, da gegeben sein muss, dass der Exponent der ersten Stelle von G dem Exponenten der zweiten Stelle von M entspricht.

Als nächstes wird ein neues Tupel R erstellt, welches die definierende Relation darstellt. Die definierende Relation kann aus dem Minimalpolynom M bestimmt werden,

in dem lediglich die erste Stelle, also das Monom mit dem höchsten Exponenten, weggelassen wird. Beispielsweise kann das Minimalpolynom $M(X) = X^3 + X + 1$ (in Tupelschreibweise: *(1,0,1,1)*) als definierende Relation der Form $X^3 = X + 1$ (in Tupelschreibweise: *(0,1,1)*) geschrieben werden. Dabei geht zwar die Information verloren, welches die Stelle mit dem höchsten Exponenten ist, doch diese Information kann weiterhin über den Grad des Minimalpolynoms ermittelt werden. Da in diesem Algorithmus mit der definierenden Relation gerechnet werden muss, ist es sinnvoll, diese in dem Tupel R zu speichern.

Anschließend werden alle Stellen des Tupels F ermittelt, an denen eine Eins steht. Dies wird mittels einer Schleife, die über die Stellen von F iteriert und in eine Liste schreibt, welche Stellen eine Eins beinhalten, ermöglicht.

Da wir nun in einer Liste genau die Stellen haben, an denen F eine Eins hat, können wir mittels der Hilfsfunktion *"create_splitted_tuples"* eine Liste an Tupeln erstellen, die jeweils genau eine Eins haben, wobei diese an genau den Stellen stehen, an denen sie auch in dem ursprünglichen Tupel F stehen. Diese Tupel haben alle jeweils die Länge des ursprünglichen Tupels, da diese Länge als zweiter Funktionsparameter, neben den Stellen, an denen sich in dem Ausgangstupel die Einsen befinden, mitgegeben wurde.

Mittels dieser Liste an Tupeln wird im weiteren Verlauf die Multiplikation funktionieren, indem der zweite Tupel-Faktor G schrittweise mit den einzelnen Tupeln der eben erstellten Tupel-Liste multipliziert wird. Sollte in dieser Berechnung der Grad des Produkt-Polynoms auf den Grad des Minimalpolynoms anwachsen, so wird die höchste Stelle durch die definierende Relation ersetzt, und diese wird dann zu dem Produkt-Polynom hinzuaddiert.

Nun wird das Lösungspolynom L definiert, in dem im weiteren Verlauf schrittweise die Lösung der Multiplikation eingefügt wird.

Anschließend wird in einer Schleife über die Anzahl der Elemente in der erstellten Tupel-Liste iteriert. Nun wird geprüft, ob im aktuellen Tupel der Iteration die letzte Stelle eine Eins ist. Da die letzte Stelle den Exponenten Null darstellt, bedeutet das, sollte die letzte Stelle eine Eins sein, dass das zweite Polynom G lediglich mit Eins multipliziert werden muss. Das kann über die Addition des Polynoms G zum Lösungspolynom L dargestellt werden.

Sollte die letzte Stelle des aktuellen Tupels keine Eins sein, muss ermittelt werden,

an welcher Stelle die Eins in dem Polynom steht, um damit die richtige Potenz bestimmen zu können, die für den nächsten Schritt der Multiplikation benötigt wird. Im nächsten Schritt wird die Distanz von der Stelle, an der sich die Eins des Tupels befindet, zur vorletzten Stelle berechnet. Denn für den Algorithmus ist diese Distanz entscheidend, um zu wissen, wie oft die schrittweise Multiplikation des Polynoms G mit dem Monom X durchgeführt werden muss. Beispielsweise bedeutet die Distanz 2, dass der Exponent zu der zugehörigen Einser-Stelle 3 beträgt. Dieses Monom wiederum kann dann zu dreimal dem Monom X aufgespalten werden (jeweils durch eine Multiplikation getrennt). Denn laut Potenzregel für die Multiplikation gilt: $X^3 = X * X * X$. Dann muss der Algorithmus dreimal das Monom X mit dem Polynom G multiplizieren.

Diese Anzahl, die bestimmt, wie oft das Monom X mit dem Polynom G multipliziert werden muss, ergibt sich aus der eben errechneten Distanz (von der Stelle der Eins zur vorletzten Stelle), indem sie um Eins erhöht. Wie man am Beispiel gesehen hat, ergibt eine Distanz von 2, dass der Algorithmus dreimal durchlaufen werden muss. Diese Information wird als *'iteration_count'* gespeichert, da sie im weiteren Verlauf des Algorithmus benötigt wird.

Nun wird ein neues Tupel *temp* erstellt, und mit dem Inhalt von G befüllt. Dieses Tupel dient als Zwischenspeicher, der in den Iterationsstufen des Algorithmus dazu dient, zu speichern, was in jedem Schritt zum Lösungspolynom L hinzuaddiert werden muss. Ein neues Tupel zur Zwischenspeicherung muss erstellt werden, da der Inhalt des Tupels G unverändert erhalten bleiben muss.

Im Anschluss folgt eine Schleife, wobei die Durchlaufzahl von dem eben ermittelten *'iteration_count'* bestimmt wird. Innerhalb der Schleife wird der Inhalt des Tupels *'temp'* erneuert. Und zwar dadurch, dass das Tupel *'temp'* einmal nach links geshiftet wird. Dies wird umgesetzt, indem die erste Stelle abgeschnitten wird und am rechten Ende eine zusätzliche 0 hinzugefügt wird. Somit bleibt die Länge des Tupels erhalten. Zu diesem verschobenen Tupel wird nun die Relation (mittels des Tupels R) hinzuaddiert. Jedoch findet diese Addition nur dann statt, wenn die erste Stelle, die von dem Tupel *temp* beim Shiften abgeschnitten wurde, eine Eins war. Dies wird sichergestellt, indem das Tupel für die Relation R mit der weggeschnittenen Stelle von *'temp'* multipliziert wird.

Anschließend wird das Tupel für die Zwischenspeicherung *'temp'*, für jeden Schleifendurchlauf der aufgespaltenen Tupel, zum Lösungspolynom L hinzuaddiert.

Am Ende der Funktion wird das Lösungstupel L zurückgegeben. In diesem Tupel steht nun das Produkt der Multiplikation der beiden Ausgangstupel und ist bereits mittels des Minimalpolynoms bzw. mittels der definierenden Relation reduziert, sodass das Lösungstupel auch ein Element des Körpers ist, der durch die Relation definiert wird.

Beispielrechnung:

Für die Beispielrechnung des alternativen Algorithmus für die Multiplikation wurden die gleichen Polynome wie für die Beispielrechnung des anderen Multiplikationsalgorithmus verwendet. Zudem wird auch das gleiche Minimalpolynom benutzt.

Die beiden Polynome, die miteinander multipliziert werden, lauten $F(X) = X^2 + 1$ und $G(X) = X^4 + X + 1$. Diese Polynome werden mittels des Minimalpolynoms $M(X) = X^5 + X^2 + 1$ reduziert. Diese drei Polynome werden durch die Tupel $F = (1, 0, 1)$, $G = (1, 0, 0, 1, 1)$ und $M = (1, 0, 0, 1, 0, 1)$ dargestellt.

Zu Beginn wird geprüft, ob eins der beiden Polynome F oder G einen Grad hat, der mindestens so groß ist wie der des Minimalpolynoms M, denn das ist die Bedingung dafür, dass ein Element kein Teil des Körpers ist. Sollte dies der Fall sein, so gibt es einen Fehler, da einer der Faktoren kein Element des Körpers ist, der durch das Minimalpolynom M, bzw. die Relation, definiert wird. Dieser Fall ist in dem gegebenen Beispiel jedoch nicht gegeben, da der Grad von F gleich 2, der Grad von G gleich 4 und der Grad von M gleich 5 ist. Das Tupel G muss hierbei nicht um führende Nullen ergänzt werden, da bereits gilt: $len(G) = 5 = 6 - 1 = len(M) - 1$

Im Anschluss wird aus dem Tupel M die definierende Relation gebildet, indem die erste Stelle, also der höchste Exponent, weggeschnitten wird. Hier lautet $R = (0, 0, 1, 0, 1)$. Dann werden die Stellen bestimmt, bei denen das Tupel F eine Eins hat, und all diese Stellen werden in einer Liste gespeichert: *pos_of_ones* $= [0, 2]$. Aus dieser Liste wird eine Liste mittels der Hilfsfunktion *"create_splitted_tuples"* erstellt, die Polynome der Länge des Tupels F erstellt, die jeweils nur eine Eins haben, und diese Einsen stehen an den Stellen, die durch *'pos_of_ones'* bestimmt werden. In diesem Beispiel sieht diese Liste folgendermaßen aus:

$$\textit{tuple\_list} = [(1, 0, 0), (0, 0, 1)]$$

Als nächstes wird ein leeres Tupel L definiert, in das das Ergebnis geschrieben wird.

Anschließend folgt eine Schleife, die über die Elemente der Liste *'tuple_list'* iteriert. Für das erste Tupel wird geprüft, ob die letzte Stelle eine Eins ist. Da dies jedoch nicht der Fall ist, wird die Distanz von der Stelle, an der die Eins des aktuellen Tupels steht, zur vorletzten Stelle berechnet. Das aktuelle Tupel lautet $(1, 0, 0)$, demnach gilt: *distance* $= 3 - 2 - 0 = 1$, da das Tupel die Länge 3 hat, und die Eins am Index 0 steht. Für den *'iteration_count'* muss zur Distanz noch eins hinzuaddiert werden, sodass dieser hier 2 beträgt.

Anschließend wird das Tupel G als *'temp'* zwischengespeichert, somit gilt:

$$\textit{temp} = (1, 0, 0, 1, 1)$$

Im Anschluss wird eine Schleife durchgeführt, die so oft läuft, wie eben in *'iteration_count'* berechnet wurde. Innerhalb dieser Schleife wird der Inhalt des Tupels *'temp'* verändert. Und zwar um die folgende Gleichung:

$$\textit{temp} = (\textit{temp}[1:] + (0)) + \textit{temp}[0] \cdot R$$

Das bedeutet, der neue Inhalt von *'temp'* besteht aus zwei Summanden. Der eine Summand ist der vorherige Wert des Zwischenspeicher-Tupels, wobei dies um eins nach links verschoben wurde und um eine Null nach rechts aufgefüllt wurde. Der andere Summand ist das Produkt aus der weggeschnittenen ersten Stelle und der Relation R.

Im ersten Schleifendurchlauf wird das Zwischenspeicher-Tupel zu:

$$\textit{temp} = (0, 0, 1, 1, 0) + 1 \cdot (0, 0, 1, 0, 1) = (0, 0, 0, 1, 1)$$

Im zweiten Durchlauf wird *'temp'* zum folgenden Tupel:

$$\textit{temp} = (0, 0, 1, 1, 0) + 0 \cdot (0, 0, 1, 0, 1) = (0, 0, 1, 1, 0)$$

Nun ist die Schleife zu Ende und das Tupel für die Zwischenspeicherung wird zum Lösungstupel L hinzuaddiert. Da L bisher noch leer war, beträgt dieses Tupel jetzt: $L = (0, 0, 1, 1, 0)$.

Dann geht die Iteration über die Elemente der Liste *'tuple_list'* zum zweiten Tupel. Da bei dem zweiten Tupel $(0, 0, 1)$ die letzte Stelle eine Eins ist, muss zum Lösungstupel

L lediglich das Polynom G hinzuaddiert werden.

$$L = (0, 0, 1, 1, 0) + (1, 0, 0, 1, 1) = (1, 0, 1, 0, 1)$$

Da jetzt über alle Tupel aus der Liste *'tuple_list'* iteriert wurde, stellt $L = (1, 0, 1, 0, 1)$ das Ergebnis der Multiplikation mittels des alternativen Algorithmus dar, wobei dieses Ergebnis direkt durch das Minimalpolynom M reduziert wurde. In Polynomschreibweise beträgt das Ergebnis der Multiplikation also:

$$F(X) = X^4 + X^2 + 1$$

Vergleicht man nun die Ergebnisse der beiden Beispielaufgaben für die verschiedenen Multiplikations-Algorithmen, stellt man fest, dass diese das gleiche Ergebnis liefern. Das ergibt Sinn, da für beide Multiplikationen dieselben Faktoren sowie das gleiche Minimalpolynom gewählt wurden.

3.3.5 Polynomdivision

Die Division von Polynomen funktioniert so, dass man die beiden Polynome mit Rest teilt. Dazu schaut man, wie oft der Nenner in den Zähler passt, und das, was dabei übrig bleibt, wird der Rest. Das Vorgehen dabei ist wie beim schriftlichen Dividieren von reellen Zahlen, nur dass hierbei mit Monomen, d.h. mit Potenzen zur Variable X gerechnet wird. Der Fakt, dass wir in der Charakteristik zwei sind, erleichtert diese Polynomdivision in gleich zwei verschiedenen Aspekten. Zum einen gibt es als Vorfaktor zu den jeweiligen Monomen lediglich die beiden Möglichkeiten null und eins, das heißt, der Vorfaktor bestimmt nur, ob das Monom vorhanden ist oder nicht. Zum anderen gelten die üblichen Rechenregeln der Charakteristik zwei, das bedeutet, dass es keine Subtraktion gibt, da diese im Binärsystem gleichbedeutend mit der Addition ist. Dazu vereinfacht die Rechenregel $1+1 = 0$ die Polynomdivision um einiges, da, wenn in einer Addition zwei Monome mit dem gleichen Exponenten addiert werden müssen, streichen sich die beiden Monome aufgrund dieser Rechenregel weg.

Der genaue Vorgang der Polynomdivision läuft folgendermaßen ab. Zu Beginn wird geguckt, wie oft der Nenner in den Zähler passt. Dies kann über deren Exponenten bestimmt werden, und zwar passt der Nenner so oft in den Zähler, wie die Differenz der beiden höchsten Exponenten von Zähler und Nenner ist. Dann wird zum Ergebnis ein Monom mit genau diesem Exponenten geschrieben. Als nächstes folgt die Rückrechnung, das heißt, das Monom, welches eben zum Ergebnis hinzugefügt wurde, wird mit dem Nenner multipliziert. Dieses Ergebnis wird anschließend mit

dem ursprünglichen Zähler addiert. Die Summe aus dieser Rechnung stellt nun den neuen Zähler dar. Also wird der beschriebene Vorgang im nächsten Schritt mit diesem neuen Zähler und dem ursprünglichen Nenner durchgeführt. Dieser Prozess wird so lange wiederholt, bis beim Schritt der Addition die Summe entweder null oder ein Polynom mit kleinerem Grad als dem Grad des Nenners ergibt.

Sollte diese Summe null ergeben, bedeutet das, dass die Polynomdivision ohne Rest aufgegangen ist. In diesem Fall ist somit der Nenner ein echter Teiler des Zählers, also kann man den Zähler in die zwei Faktoren zerlegen. Zum einen in den Nenner und zum anderen in das Polynom, welches als Ergebnis der Polynomdivision herauskommt. Sollte jedoch das Ergebnis der Summe ein Polynom ergeben, dessen Grad kleiner ist als der Grad des Nennerpolynoms, bedeutet das, dass dieses Polynom nicht mehr in den Nenner passt, und somit stellt dieses Polynom den Rest dar, der bei der Polynomdivision übrig bleibt. In diesem Fall führt die durchgeführte Polynomdivision zu dem Ergebnis-Polynom, wobei der eben aufgeführte Rest zusätzlich übrig bleibt.

```python
def polynom_division(F, G):

    if G == (1,):
        return F, (0,)

    L = []

    F = list(F)

    while len(F) >= len(G):

        L.append(F[0])

        if F[0] == 1:
            for i in range(len(G)):
                F[i] ^= G[i]

        F.pop(0)

    return tuple(L), tuple(F)
```

In der Implementierung werden der Funktion *"polynom_division"* die beiden Polynome als Parameter gegeben. Die Funktion gibt zwei verschiedene Werte zurück, zum einen das Ergebnis der Polynomdivision und zum anderen den übrigbleibenden Rest. Als erstes wird geprüft, ob das Nenner-Polynom gleich 1 ist. Sollte dies der Fall sein, wird als Ergebnis das Zähler-Polynom und 0 als Rest zurückgegeben.

Sollte das Nenner-Polynom ein nicht-triviales Polynom sein, wird zunächst eine Liste

L definiert, in die die Lösung geschrieben wird. Zudem wird das Tupel F in eine Liste umgewandelt, sodass dessen Inhalt bearbeitet werden kann.

Nun wird über die Länge des Zähler-Polynoms F iteriert, und diese Schleife bricht ab, sobald dieses Polynom, welches nun in Form einer Liste vorliegt, kürzer als das Nenner-Polynom G ist. Das ist gleichbedeutend damit, dass der Grad des Zählers kleiner als der Grad des Nenners ist.

In jedem Iterationsschritt wird zur Lösungsliste L die aktuelle Stelle der Iteration von F hinzugefügt. Wenn diese Stelle nun eine Null war, bedeutet das, dass keine Rückrechnung erforderlich ist, womit lediglich die Potenzen im Lösungstupel um eins erhöht wurden. Sollte diese Stelle jedoch eine Eins sein, muss eine Rückrechnung durchgeführt werden. Für diese Rückrechnung müssen zwei Polynome addiert werden. Zum einen das Zähler-Polynom F beginnend bei dem aktuellen Index der Schleifeniteration. Und zum anderen das Nenner-Polynom G. Diese Addition wird mittels dem *'XOR'*-Operator umgesetzt, da dieser gleichbedeutend mit der Addition in der Charakteristik zwei ist. Im Zuge dieser Iteration wird über die Stellen des Nenner-Polynoms G iteriert, und für jede Stelle einzeln die *'XOR'*-Anweisung durchgeführt. Somit werden die beiden Polynome, stellenweise addiert, und das Ergebnis wird in der Liste des Zählertupels F abgelegt. Anschließend kann die erste Stelle der Liste des Zählertupels entfernt werden, da diese Stelle bereits verarbeitet wurde.

Dieser Vorgang wird so lange wiederholt, bis die Abbruchbedingung erreicht ist, also sobald die Liste F kürzer ist als das Nenner-Polynom G . Dann wird die Lösungsliste L, in Form eines Tupels, als Ergebnis zurückgegeben. Zudem wird die Liste F, in Form eines Tupels, als Rest zurückgegeben, da das, was noch in diesem Tupel übrig bleibt, den Rest der Polynomdivision darstellt.

Beispielrechnung:

Für das Beispiel der Polynomdivision werden die beiden Polynome $F(X) = X^3 + X^2 + X + 1$ und $G(X) = X^2 + 1$ verwendet. In Tupelschreibweise lauten diese Tupel $F = (1, 1, 1, 1)$ und $G = (1, 0, 1)$.

Im ersten Schritt wird geprüft, ob das Tupel G gleich 1 ist. Sollte dies der Fall sein, wäre F die Lösung mit Rest 0. Da jedoch $G \neq (1)$ ist diese Abbruchbedingung nicht gegeben. Dann wird eine leere Liste L definiert, in die später die Lösung geschrieben wird. Zudem wird das Tupel F in eine Liste umgewandelt, da Tupel in Python *'immutable'* sind, jedoch wird F in dieser Funktion bearbeitet.

Als nächstes folgt eine Schleife, die so lange iteriert, bis F kürzer als G ist. Da F die Länge 4 und G die Länge 3 hat, ist diese Abbruchbedingung noch nicht erreicht. Dann wird an die Lösungsliste L das erste Element von F gehängt. Somit lautet $L = [1]$. Da nun die erste Stelle von F eine 1 ist, wird über das Tupel G iteriert, wobei F und G, mittels der *'XOR'*-Verknüpfung, stellenweise miteinander addiert werden. Somit wird aus F:

$$F = [1, 1, 1, 1] \oplus [1, 0, 1] = [0, 1, 0, 1]$$

Dann wird noch die erste Stelle von F weggeschnitten. Damit lautet $F = [1, 0, 1]$.

Im nächsten Schleifendurchlauf ist die Länge von F nun 3, somit ist die Abbruchbedingung noch nicht erreicht, da G ebenfalls die Länge 3 hat. Dann wird erneut die erste Stelle von F an die Liste L gehängt, somit lautet diese Liste nun: $L = [1, 1]$. Da diese Stelle von F erneut eine Eins war, muss F wieder angepasst werden.

$$F = [1, 0, 1] \oplus [1, 0, 1] = [0, 0, 0]$$

Dann wird wieder die erste Stelle von F abgeschnitten, sodass diese Liste nur wie folgt aussieht: $F = [0, 0]$.

Nun ist die Länge von F nun 2 und somit ist diese Liste kürzer als G, weshalb die Abbruchbedingung der Schleife erreicht ist. Damit lautet das Ergebnis der Polynomdivision $L = [1, 1]$ mit Rest $F = [0, 0]$. In Polynomschreibweise lautet das Ergebnis $X + 1$ Rest 0.

3.4 Polynom-Algorithmen

In diesem Abschnitt werden weitere Algorithmen beschrieben, die für den Umgang mit Polynomen geeignet sind. Zum einen der euklidische Algorithmus zur Bestimmung des ggT von zwei Polynomen. Und zum anderen zwei Algorithmen, die ein Polynom darauf prüfen, ob es ein Minimalpolynom ist, bzw. ob das Polynom irreduzibel ist.

3.4.1 Euklidischer Algorithmus

In der Implementierung muss der euklidische Algorithmus für Polynome etwas angepasst werden. Dabei wird die Rückwärtsrechnung durch Vorwärtsrechnen ersetzt, da man sich für die Rückwärtsrechnung alle Zwischenschritte merken muss, während man sich beim Vorwärtsrechnen lediglich den letzten Schritt merken muss.

Daher ist dieses Verfahren für die Implementierung geeigneter.

```python
def euclid(m, n):

    r0, r1 = m, n

    a0, b1 = (1,), (1,)
    a1, b0 = (0,), (0,)

    while True:

        q, r = polynom_division(r0, r1)

        if cut_zeros_left(r) == (0,):
            return r1, a1, b1

        a0, a1 = a1, add_polynoms(a0, multiply_polynoms(q, a1))
        b0, b1 = b1, add_polynoms(b0, multiply_polynoms(q, b1))

        r0, r1 = r1, cut_zeros_left(r)
```

Für den Algorithmus werden die folgenden Variablen benötigt, $a_0, a_1, b_0, b_1, r_0, r_1$. Dabei stehen a und b jeweils für die beiden Vorfaktoren der Polynome, die später zu den Bézout-Koeffizienten werden. Die Variablen mit r stehen für die beiden Polynome, auf denen der euklidische Algorithmus angewandt wird. Die Indizes 0 und 1 stehen dabei für die alten, bzw. neuen Werte der jeweiligen Polynome, da diese in jedem Iterationsschritt des Algorithmus angepasst werden müssen.

Der Funktion werden die beiden Polynome m und n als Parameter übergeben. Zu Beginn werden die Variablen r_0 und r_1 initial auf die beiden Funktionsparameter gesetzt, und zwar so, dass $deg(r_1) \leq deg(r_0)$. Während des Algorithmus müssen die beiden folgenden Gleichungen immer gelten:

$$r_0 = a_0 \cdot m + b_0 \cdot n$$

$$r_1 = a_1 \cdot m + b_1 \cdot n$$

Damit diese Gleichungen für die initiale Befüllung gelten, müssen die Variablen folgendermaßen befüllt werden. Für die erste Gleichung: $a_0 = 1$, $b_0 = 0$ und für die zweite Gleichung: $a_1 = 0$, $b_1 = 0$.

Anschließend folgt eine Schleife, die solange läuft, bis der ggT der beiden Zahlen gefunden wurde, wobei neben dem ggT auch die beiden Bézout-Koeffizienten zu den beiden Polynomen zurückgegeben werden.

Zu Beginn der Schleife wird auf die beiden Tupel r_0 und r_1 mithilfe der bereits defi-

nierten Funktion *"polynom_division"* eine Polynomdivision durchgeführt. Dabei wird das Ergebnis in die Variable q und der Rest in r gespeichert. Als Abbruchbedingung des euklidischen Algorithmus gilt, dass der Rest bei der Polynomdivision gleich null sein muss. Wenn also bei der Polynomdivision der Rest r null ist, so ist der ggT gefunden und steht in $r1$. Dieser ggT wird zusammen mit den beiden Bézout-Koeffizienten zurückgegeben.

Sollte in dieser Schleifeniteration der ggT noch nicht gefunden worden sein, so werden alle Variablen für den nächsten Iterationsschritt angepasst. Dafür werden a_0 und b_0 mittels a_1 und b_1 erneuert. Anschließend werden die beiden Werte a_1 und b_1 wiederum durch die folgenden Gleichungen angepasst:

$$a_1 = a_{0,old} + q \cdot a_1$$

$$b_1 = b_{0,old} + q \cdot b_1$$

Dann wird noch das Tupel r_0 durch r_1 neu befüllt und r_1 bekommt den Inhalt von r, wobei von r führende Nullen weggeschnitten werden.

Mit diesen neuen Variableninhalten folgt nun der nächste Iterationsschritt der Schleife. Diese Schritte werden so lange wiederholt, bis die Abbruchbedingung, dass der Rest der Polynomdivision gleich null ist, erreicht ist, und somit der ggT gefunden wurde.

Beispielrechnung:

Für die Beispielrechnung des euklidischen Algorithmus wurden die beiden folgenden Tupel gewählt: $m(X) = X^3 + X + 1$ und $n(X) = X^2 + 1$. In Tupelschreibweise lauten diese: $m = (1, 0, 1, 1)$ und $n = (1, 0, 1)$ Zu Beginn werden diese Tupel in die Variablen r_0 und r_1 geschrieben. Dann werden die anderen Variablen folgendermaßen initial gefüllt: $a_0 = (1)$, $b_0 = (0)$, $a_1 = (0)$, $b_1 = (1)$

Dann folgt eine Schleife, die so lange läuft, bis der ggT der beiden Polynome gefunden wurde. Innerhalb dieser Schleife wird auf die beiden Tupel r_0 und r_1 eine Polynomdivision durchgeführt, wobei das Ergebnis in q und der Rest in r gespeichert werden. In diesem Beispiel für die Polynomdivision zu $q = (1, 0)$ und $r = (1)$. Da $r \neq (0)$ wurde der ggT noch nicht gefunden, und die Variablen müssen angepasst werden. Dabei werden a_1 und b_1 anhand der folgenden Gleichungen aktualisiert.

$$a_1 = a_0 + q \cdot a_1 = (1) + (1, 0) \cdot (0) = (1)$$

$$b_1 = b_0 + q \cdot b_1 = (0) + (1,0) \cdot (1) = (1,0)$$

Die Variablen a_0 und b_0 bekommen jeweils die alten Werte von a_1 und b_1: $a_0 = (0)$, $b_0 = 1$. Zudem wird in r_0 der Inhalt von r_1, und in r_1 der Rest der Polynomdivision r geschrieben. Somit gilt: $r_0 = (1,0,1)$ und $r_1 = (1)$.

Im nächsten Schleifendurchlauf wird eine Polynomdivision auf die neuen Werte von r_0 und r_1 durchgeführt. Diese liefert das Ergebnis $q = (1,0,1)$ mit Rest $r = (0)$. Da nun der Rest null ist, ist die Abbruchbedingung der Schleife erreicht, und somit stellt $r_1 = (1)$ den ggT dar. Die Variablen $a_1 = (1)$ und $b_1 = (1,0)$ stellen die Bézout-Koeffizienten zu dieser Beispielrechnung dar.

In Polynomschreibweise lautet das Ergebnis:

$$1 = 1 \cdot m + X \cdot n$$

wobei die 1 auf der linken Seite den ggT darstellt, während m und n die Ausgangs-Polynome, bzw. Tupel sind.

3.4.2 Minimalpolynome

Minimalpolynome sind Polynome, die keinen nicht-trivialen Teiler haben, das heißt, sie sind durch kein nicht-triviales Polynom ohne Rest teilbar. Minimalpolynome haben die Eigenschaft, dass aus ihnen eine Relation geschaffen werden kann, die Körper der Form $\mathbb{F}_q$ mit $q = p^l$ definieren.

Ist ein Polynom

$$F(X) = X^l + r_{l-1} \cdot X^{l-1} + ... + r_1 \cdot X + r_0$$

ein Minimalpolynom des Körpers $\mathbb{F}_q$, so gibt es ein $\alpha \, \epsilon \, \mathbb{F}_q$, sodass die Relation

$$\alpha^l = -r_{l-1} \cdot \alpha^{l-1} - ... - r_1 \cdot \alpha - r_0$$

den Körper $\mathbb{F}_q$ definiert.

Da in der Charakteristik die Subtraktion analog zur Addition ist, kann die definierende Relation in der Form

$$\alpha^l = r_{l-1} \cdot \alpha^{l-1} + ... + r_1 \cdot \alpha + r_0$$

geschrieben werden.

Damit ein Polynom ein Minimalpolynom sein kann, darf es zum einen keine Nullstelle über dem Grundkörper $\mathbb{F}_2$ haben. Und zum anderen darf es nicht in zwei nicht-triviale Faktoren aufgespalten werden können. Das bedeutet, es darf keinen nicht-trivialen Teiler haben, der das Polynom ohne Rest teilt.

Um bestimmen zu können, ob ein Polynom einen nicht-trivialen Teiler hat, benötigt man zunächst alle Minimalpolynome bis zum Grad $\lfloor \frac{d}{2} \rfloor$ (wenn d der Grad des Polynoms ist, das auf ein Minimalpolynom überprüft wird, ist). Das liegt daran, dass ein Polynom nur in mindestens zwei Faktoren zerfallen kann, und wenn es einen nicht-trivialen Teiler hat, dessen Grad größer als $\lfloor \frac{d}{2} \rfloor$ ist, so hat es auch einen nicht-trivialen Teiler, dessen Grad kleiner, bzw. gleich $\lfloor \frac{d}{2} \rfloor$ ist. Deshalb reicht die Überprüfung auf nicht-triviale Teiler bis zum Grad $\lfloor \frac{d}{2} \rfloor$.

Beispielsweise muss ein Polynom des Grades 5 auf echte Teiler bis zum Grad $\lfloor \frac{5}{2} \rfloor = 2$ überprüft werden. Dafür muss das Polynom zuerst auf Nullstellen im Grundkörper $\mathbb{F}_2$ überprüft werden, wodurch auf nicht-triviale Teiler des Grades 1 überprüft wird. Zudem muss es noch auf nicht-triviale Teiler des Grades 2 überprüft werden. Da es für den Grad 2 lediglich das Minimalpolynom $X^2 + X + 1$ gibt, reicht es, das Polynom des Grades 5 durch dieses Minimalpolynom zu teilen. Sollte bei dieser Polynomdivision ein Rest übrig bleiben, so ist $X^2 + X + 1$ kein Teiler des Polynoms vom Grad 5. Somit hat das Polynom keine echten Teiler vom Grad 1 oder 2. Daraus folgt, dass es auch keine echten Teiler vom Grad 3 oder 4 hat, da, wenn es einen Teiler vom Grad 3 (bzw. vom Grad 4) hätte, wäre der andere Faktor, in den das Polynom zerfällt, vom Grad 2 (bzw. vom Grad 1). Jedoch wurden bereits echte Teiler vom Grad 1 und 2 ausgeschlossen, wodurch auch die Grade 3 und 4 ausgeschlossen werden können.

```python
def is_minpol(F):

    if has_root(F):
        return False

    max_div_degree = find_degree(F) // 2

    list_of_minpols = get_all_minpols(max_div_degree)

    for G in list_of_minpols:

        L, R = polynom_division(F, G)

        if tuple_to_polynom(R) == 0:
            return False

    return True
```

Die Funktion *"is_minpol"* bekommt ein Polynom, in Form eines binären Tupels, als Parameter und gibt zurück, ob das Polynom ein Minimalpolynom ist oder nicht. Da-

bei wird zunächst mittels der Hilfsfunktion *"has_root"* überprüft, ob das Polynom, bzw. das binäre Tupel eine Nullstelle im Grundkörper $\mathbb{F}_2$ hat. Wenn das Polynom eine Nullstelle hat, bedeutet das, dass das Polynom kein Minimalpolynom sein kann, weshalb die Funktion in diesem Fall *'False'* zurückgibt.

Als nächstes wird der maximale Teilergrad bestimmt, in den das Polynom, vom Grad d zerfallen kann. Dieser maximale Teilergrad beträgt $\lfloor \frac{d}{2} \rfloor$. Daraufhin wird eine Liste an Minimalpolynomen bestimmt, deren Grad von 2 bis $\lfloor \frac{d}{2} \rfloor$ reicht. Dazu wurde die Funktion *"get_all_minpols"* implementiert.

```
1 def get_all_minpols(max_degree):
2
3     return [polynom for degree in range(2, max_degree + 1)
4                     for polynom in product([0,1], repeat = degree + 1)
5                     if polynom[0] and is_minpol(polynom)]
```

Diese Funktion bekommt als Parameter den maximalen Teilergrad, bis zu dem die Minimalpolynome bestimmt werden sollen. Diese von der Funktion *"get_all_minpols"* generierten Polynome haben einen Grad von 2 bis zum gegebenen maximalen Grad. Mittels der Funktion *'product'* von der Python-Bibliothek *'itertools'* werden alle möglichen Tupelkombinationen erstellt, deren Länge jeweils zu dem Grad des Tupels passt. Anschließend wird noch überprüft, dass die erste Stelle des Tupels eine 1 ist, da ansonsten der Grad nicht mehr zu dem Tupel passt, und es wird mittels der Funktion *"is_minpol"* nur die Tupel zurückgegeben, die auch ein Minimalpolynom sind. Eine vollständige Liste an Minimalpolynomen befinden sich in Kapitel 5: 'Liste an Minimalpolynomen'.

Dabei fällt auf, dass die beiden Funktionen *"is_minpol"* und *"get_all_minpols"* sich gegenseitig aufrufen, das heißt, es werden schrittweise alle Minimalpolynome bis zu dem bestimmten Teilergrad generiert, und dabei muss in jedem Schritt überprüft werden, ob die möglichen Teilerpolynome auch selbst bereits Minimalpolynome sind.

Das bedeutet, dass diese Implementierung für höhere Grade der zu überprüfenden Polynome immer ineffizienter wird, da sich die beiden Funktionen immer häufiger gegenseitig aufrufen müssen. Für Grade bis ca. 10 funktioniert diese Implementierung noch, jedoch steigt die Ausführungszeit der Funktion exponentiell. Im späteren Verlauf wird der Algorithmus von Cantor-Zassenhaus implementiert, der Polynome auf Irreduzibilität, und damit auch auf Minimalpolynome, überprüft, der deutlich effizienter ist. Mit diesem Algorithmus können dann auch Polynome mit Graden größer

als zehn auf Minimalpolynome überprüft werden.

Nachdem die Funktion *"is_minpol"* die Liste aller Minimalpolynome bis zum Grad $\lfloor \frac{d}{2} \rfloor$ bestimmt hat, wird über diese Liste iteriert, und für jedes Element dieser Liste wird eine Polynomdivision, mittels der Funktion *"polynom_division"* mit dem zu überprüfenden Polynom F durchgeführt. Sollte eine dieser Polynomdivisionen ohne Rest aufgehen, bedeutet das, dass das Polynom F einen nicht-trivialen Teiler hat, und demnach kein Minimalpolynom sein kann. In diesem Fall wird von der Funktion *'False'* zurückgegeben.

Sollte wiederum keine Polynomdivision, von F mit den Minimalpolynomen bis zum Grad $\lfloor \frac{d}{2} \rfloor$, ohne Rest aufgehen, so ist F ein Minimalpolynom, und somit gibt die Funktion *'True'* zurück.

Beispielrechnung:

Für die Beispielrechnung wird das Polynom $F(X) = X^6 + X + 1$ darauf überprüft, ob es ein Minimalpolynom ist. Dafür muss zu Beginn geprüft werden, ob das Tupel $F = (1, 0, 0, 0, 0, 1, 1)$, welches das Polynom $F(X)$ darstellt, eine Nullstelle im Grundkörper $\mathbb{F}_2$ hat. Dies kann mittels der Hilfsfunktion *"has_root"* überprüft werden. Da die letzte Stelle von F eine Eins ist und die Anzahl der Einsen ungerade ist, hat F im Grundkörper $\mathbb{F}_2$ keine Nullstelle und somit kommt F dafür in Frage, ein Minimalpolynom zu sein.

Als nächstes wird der maximale Teilergrad bestimmt, den ein Polynom haben kann, um das Polynom F zu teilen. Dieser liegt bei $\left\lfloor \frac{deg(F)}{2} \right\rfloor = \lfloor \frac{6}{2} \rfloor = 3$. Dann werden mittels der Funktion *"get_all_minpols"* alle Minimalpolynome bis zu diesem maximalen Teilergrad bestimmt und anschließend in einer Liste gespeichert. In diesem Beispiel lautet diese Liste: $[(1, 1, 1), (1, 0, 1, 1), (1, 1, 0, 1)]$.

Anschließend wird über diese Liste iteriert, und für jedes Minimalpolynom überprüft, ob es das Polynom F ohne Rest teilt. Für das erste Minimalpolynom $(1, 1, 1)$ ergibt die Polynomdivision $L = (1, 1, 0, 1, 1)$ mit Rest $R = (1, 0)$. Da der Rest ungleich null ist, wird in der nächsten Iteration geprüft, ob das Minimalpolynom $(1, 0, 1, 1)$ F ohne Rest teilt. Hierbei ist $L = (1, 0, 1, 1)$ das Ergebnis mit dem Rest $R = (1, 1, 0)$. Da auch hier der Rest ungleich null ist, muss nur noch das letzte Minimalpolynom der Liste überprüft werden. Die Polynomdivision von F mit $(1, 1, 0, 1)$ ergibt $L = (1, 1, 1, 0)$ mit dem Rest $R = (1, 0, 1)$.

Da nun keine der Polynomdivisionen von F mit den einzelnen Minimalpolynomen bis zum Grad $\left\lfloor \frac{deg(F)}{2} \right\rfloor = 3$ ohne Rest aufgeht, bedeutet das, dass das Polynom $F(X) = X^6 + X + 1$ ein Minimalpolynom ist.

3.4.3 Cantor-Zassenhaus-Algorithmus

Der Cantor-Zassenhaus-Algorithmus ist ein Algorithmus, der für ein gegebenes Polynom vom Grad d direkt überprüft, ob es irreduzibel ist, ohne dafür alle irreduziblen Polynome vom Grad höchstens $\lfloor \frac{d}{2} \rfloor$ zu benötigen. Das heißt, ob man ein Polynom in zwei oder mehrere Polynomfaktoren aufspalten kann. Sollte dies nicht der Fall sein, so ist das Polynom irreduzibel, bzw. so ist das Polynom ein Minimalpolynom.

```python
def cantor_zassenhaus(F):

    if has_root(F):
        return False

    for M in [(1,1,1,), (1,0,1,1), (1,1,0,1)]:
        ggT, _, _ = euclid(F,M)
        if find_degree(ggT) >= 1 and find_degree(F) > 3:
            return False

    if find_derivative(F) == (0,):
        return False

    ggT_dF, _, _ = euclid(F,find_derivative(F))

    if find_degree(ggT_dF) >=1:
        return False

    for l in range(4, int(find_degree(F)/2) + 1):

        h = add_polynoms((1,) + (0,) * (2**l), (1,0))

        ggT_h, _, _ = euclid(F,h) if find_degree(F) >= find_degree(h) else euclid(h,F)

        if find_degree(ggT_h) >= 1:
            return False

    return True
```

In dem Algorithmus wird zunächst geprüft, ob das Polynom F eine Nullstelle im Grundkörper $\mathbb{F}_2$, einen quadratischen oder kubischen Teiler hat, indem mittels der Hilfsfunktion *"has_root"* auf Nullstellen geprüft wird und dann der ggT des Polynoms F mit den einzelnen Minimalpolynomen der Grade 2 und 3 bestimmt wird, wobei das Polynom F reduzierbar ist, sollte einer der berechneten ggT größer als 1 sein. Die Überprüfung auf einen quadratischen, bzw. kubischen Teiler muss erst für Polynome, deren Grad höher als 3 ist, durchgeführt werden, da erst für Polynome ab Grad 4 gilt: $\left\lfloor \frac{deg(F)}{2} \right\rfloor \geq 2$. Wenn eines dieser Kriterien zutrifft, so kann das Poly-

nom kein Minimalpolynom sein, und somit ist es auch nicht irreduzibel. Diese drei Überprüfungen auf Nullstellen, quadratische und kubische Teiler sind in dem Cantor-Zassenhaus-Algorithmus nicht enthalten, jedoch wurden diese aus Performance-Gründen hinzugefügt, da die meisten nicht irreduziblen Polynome bereits anhand dieser Tests als nicht irreduzibel identifiziert werden können.

Im nächsten Schritt wird geprüft, ob die Ableitung des Polynoms F gleich Null ist. Sollte dies der Fall sein, bedeutet das, dass das Polynom F nicht irreduzibel ist. Wenn die Ableitung jedoch ungleich Null ist, wird der ggT des Polynoms F und dessen Ableitung bestimmt. Sollte dieser ggT größer als 1 sein, also wenn die beiden Polynome nicht teilerfremd sind, so ist das Polynom F nicht irreduzibel.

Als nächstes wird in einer Schleife die Variable l von 4 bis $\left\lfloor \frac{deg(F)}{2} \right\rfloor$ iteriert. Normalerweise müsste diese Schleife bei $l = 1$ anfangen, durch die Vorabüberprüfung auf Nullstellen, sowie quadratische und kubische Teiler, würden die Schleifendurchläufe für $l = 1$ bis 3 jedoch dieselben Ergebnisse wie die Vorabprüfungen selbst liefern, weshalb bei $l = 4$ angefangen werden kann. Dabei wird in jedem Schleifendurchlauf das Polynom $X^{q^l} - X$ gebildet. Da wir uns in der Charakteristik zwei befinden, beträgt $q = 2$. Um diese Polynome als binäre Tupel zu generieren, werden diese in zwei Teilen erstellt. Der erste Teil, also X^{2^l} kann dadurch erstellt werden, indem an eine 1, welche die erste Stelle darstellt, 2^l Nullen angehängt werden. Der zweite Teil des Polynoms, also $+X$ (in der Charakteristik zwei entspricht das $-X$) wird so erstellt, dass an das eben erstellte Tupel für den ersten Teil X^{2^l} das Tupel $(1, 0)$, also X hinzuaddiert wird.

Nun wird der ggT des Polynoms F und dem erstellten Polynom $X^{2^l} + X$ ermittelt. Dabei muss beachtet werden, dass die Funktion *"euclid"*, welche den ggT von zwei Polynomen, bzw. binären Tupeln berechnet, so implementiert wurde, dass das Polynom mit dem höheren Grad als erster Funktionsparameter übergeben werden muss. Deshalb erfolgt vor dem Funktionsaufruf von *"euclid"* eine Überprüfung, welches der beiden Polynome den höheren Grad hat, sodass die beiden Polynome der Funktion in der richtigen Reihenfolge übergeben werden. Sollte dieser berechnete ggT nun größer als 1 sein, also sollten die beiden Polynome nicht teilerfremd sein, bedeutet das, dass das Polynom F einen Teiler vom Grad l hat und damit nicht irreduzibel ist.

Der ggT von F und $X^{2^l} + X$ wird nun in der Schleife für alle l von 4 bis $\left\lfloor \frac{deg(F)}{2} \right\rfloor$ berechnet. Sollte einer dieser ggT größer als 1 sein, so ist F nicht irreduzibel und der Algorithmus wird beendet. Sollte jedoch F mit all diesen Polynomen der Form $X^{2^l} + X$ teilerfremd sein, so ist F irreduzibel.

Beispielrechnung:

In der Beispielrechnung des Cantor-Zassenhaus-Algorithmus wird überprüft, ob das Polynom $F(X) = X^8 + X^4 + X^3 + X^2 + 1$, woraus das Tupel $F = (1, 0, 0, 0, 1, 1, 1, 0, 1)$ entsteht, irreduzibel ist. Dazu werden vorab die Checks auf Nullstellen, quadratische und kubische Teiler durchgeführt, die dem Algorithmus aus Performance-Gründen hinzugefügt wurden.

Die Hilfsfunktion *"has_root"* ergibt, dass F keine Nullstellen hat, da die letzte Stelle eine Eins ist und die Anzahl der Einsen im Tupel ungerade ist. Zudem ergeben die ggT's von F mit den Minimalpolynomen vom Grad 2 und 3, (diese lauten: $(1, 1, 1), (1, 0, 1, 1), (1, 1, 0, 1)$) alle gleich 1. Somit hat F keine Nullstellen und ist auch nicht durch einen irreduziblen Faktor vom Grad 2 oder 3 teilbar, wodurch nun der eigentliche Algorithmus fortgesetzt werden kann.

Dazu wird zunächst mittels der Hilfsfunktion *"find_derivative"* überprüft, ob die Ableitung von F gleich null ist. Da diese Ableitung $F' = (1, 0, 0)$ ist, ist dieser Fall nicht gegeben. Dann wird im nächsten Schritt der ggT von F und F' berechnet. Da dieser ggT gleich 1 ist, ist gegeben, dass die Ableitung F' kein echter Teiler des Polynoms F ist.

Als nächstes folgt eine Schleife, in der die Variable l von 4 bis $\left\lfloor \frac{deg(F)}{2} \right\rfloor = \left\lfloor \frac{8}{2} \right\rfloor = 4$ iteriert wird, also wird die Schleife lediglich einmal durchlaufen, wobei $l = 4$ ist . Innerhalb dieser Schleife werden nun Polynome der Form $X^{2^l} + X$ gebildet. Für $l = 4$ lautet dieses Polynom in Tupelschreibweise:

$$h = (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0)$$

Dieses Tupel stellt nun das Polynom $h(X) = X^{2^4} + X = X^{16} + X$ dar. Dann wird der ggT von F und h ermittelt. Da der Grad von h der höhere der beiden ist, wird es als erster Funktionsparameter in die *"euclid"* Funktion überliefert. Diese Berechnung ergibt:

$$ggT(F, h) = ggT((1, 0, 0, 0, 1, 1, 1, 0, 1), (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0)) = 1$$

Da dieser berechnete ggT gleich 1 ist, folgt, dass das Polynom F mit dem Polynom $X^{2^4} + X = X^{16} + X$ teilerfremd ist. Zudem wurde bereits bestimmt, dass F keine Nullstelle im Grundkörper $\mathbb{F}_2$ hat, dass es keinen quadratischen und keinen kubischen Teiler hat, dass $F'(X) \neq 0$ und dass $ggT(F(X), F'(X)) = 1$. Dadurch ist das Polynom $F(X) = X^8 + X^4 + X^3 + X^2 + 1$ nun laut dem Cantor-Zassenhaus-Algorithmus

irreduzibel.

3.5 Performance-Vergleiche

In diesem Abschnitt werden zum einen die beiden Algorithmen für die Multiplikation von Polynomen verglichen und zum anderen wird der Algorithmus *"MinimalPolynoms"* mit dem Cantor-Zassenhaus-Algorithmus verglichen.

3.5.1 Vergleich der beiden Multiplikations-Algorithmen

Da für die Multiplikation von Polynomen zwei unterschiedliche Algorithmen implementiert wurden, werden diese nun miteinander verglichen. Dabei wird zum einen überprüft, ob die beiden Algorithmen zu den gleichen Ergebnissen führen, und zum anderen wird die Performance der beiden Funktionen verglichen, indem die Zeit verglichen wird, die die beiden Algorithmen jeweils für eine gleiche Anzahl an Multiplikationen benötigen.

```
MinimalPolynoms = [
    (1,0,1,1), # Grad 3
    (1,0,0,1,1),
    (1,0,0,1,0,1),
    (1,0,0,0,0,1,1),
    (1,0,0,0,0,0,1,1),
    (1,0,0,0,1,1,0,1,1),
    (1,0,0,0,0,1,0,0,0,1),
    (1,0,0,0,0,0,0,1,0,0,1),
    (1,0,0,0,0,0,0,0,0,1,0,1),
    (1,0,0,0,0,0,1,0,1,0,0,1,1),
    (1,0,0,0,0,0,1,0,0,0,1,0,1,1),
    (1,0,0,0,0,0,0,0,0,1,0,1,0,1,1),
    (1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1), # Grad 15
]
```

Zu Beginn wird eine Liste an Minimalpolynomen, die vom Grad 3 bis zum Grad 15 reicht, definiert. Aus dieser Liste kann man dann im weiteren Verlauf ein Minimalpolynom auswählen, und somit kann entschieden werden, in welchem Körper die Multiplikationen durchgeführt werden sollen.

```
degree = 8
M = MinimalPolynoms[degree-3]
print(f"Minimalpolynom: M = {tuple_to_polynom(M)}")

print(f"M ist ein Minimalpolynom: {is_minpol(M)}")

n = len(M) - 1

List_of_all_tupels = list(product([0, 1], repeat=n))
```

```
Minimalpolynom: M = (X^8 + X^4 + X^3 + X + 1)
M ist ein Minimalpolynom: True
```

In nächsten Schritt kann man nun den Grad $degree$ von 3 bis 15 frei bestimmen. Anhand dieser Angabe wird dann in das Tupel M das Minimalpolynom aus der eben definierten Liste gespeichert, welches den Grad des angegebenen Wertes für $degree$ hat. Beispielsweise wird für $degree = 4$ das Tupel $M = 1, 0, 0, 1, 1$, da dies das Minimalpolynom aus der Liste mit dem Grad 4 ist.

Anschließend wird dieses Tupel M, welches das Minimalpolynom darstellt, einmal in Tupelschreibweise ausgegeben und überprüft, ob es auch wirklich ein Minimalpolynom ist. All die 13 Tupel, die in der Liste stehen, sind bereits Minimalpolynome, daher ist diese Überprüfung für diese Tupel irrelevant. Sollte jedoch ein neues Tupel als M definiert werden, so muss erst geprüft werden, ob es auch wirklich ein Minimalpolynom ist.

Dann wird in n die maximale Länge bestimmt, die ein Element in dem Körper, der durch M definiert wird, höchstens haben kann. Für diese Länge muss von der Länge von M lediglich eins abgezogen werden. Basierend auf dieser Länge n werden dann alle Tupel erstellt, die ein Element dieses Körpers sind. Dafür wird die *"product"*-Funktion aus der *"math"*-Bibliothek verwendet. Diese berechnet das kartesische Produkt, wobei die Stellen lediglich auf 0 und 1 beschränkt wurden. Und all diese Elemente sollen die Länge n haben. Somit stehen nun in *'List_of_all_tuples'* alle 2^l Elemente des Körpers $\mathbb{F}_{2^l}$, wobei l der Grad des Minimalpolynoms ist. Beispielsweise hat der Körper $\mathbb{F}_{2^3} = \mathbb{F}_8$ genau 8 Elemente, wobei der Grad des Minimalpolynoms $M(X) = X^3 + X + 1$, welches diesen Körper definiert, gleich 3 ist.

```python
correct_counter, mistake_counter = 0, 0

for F in List_of_all_tupels:
    for G in List_of_all_tupels:

        A = reduce_product(multiply_polynoms(F, G), M)
        B = alternative_multiplication(F, G, M)

        if cut_zeros_left(A) == cut_zeros_left(B):
            correct_counter = correct_counter + 1
        else:
            mistake_counter = mistake_counter + 1

print(f"Falsch: {mistake_counter}")
print(f"Korrekt: {correct_counter}")
```

```
Falsch: 0
Korrekt: 65536
```

Im nächsten Schritt werden die Ergebnisse der beiden Multiplikationen miteinander verglichen. Dazu werden zunächst zwei Zähler definiert, die initial mit 0 befüllt werden. Diese beiden Zähler zählen zum einen, wie oft die Ergebnisse der beiden Algorithmen übereinstimmen, und zum anderen, wie oft diese nicht übereinstimmen.

Nun wird ein methodischer Test durchgeführt, indem alle Elemente des Körpers $\mathbb{F}_{2^l}$ miteinander multipliziert werden. Somit erhält man $2^l \cdot 2^l = 2^{2l}$ Multiplikationen. Wenn beispielsweise der Grad des gewählten Minimalpolynoms 8 beträgt, so werden in diesem Test $2^{2 \cdot 8} = 65.536$ Multiplikationen, für jeden der beiden Algorithmen, berechnet.

Um jedes Element des Körpers miteinander zu multiplizieren, werden zwei Schleifen ineinander geschachtelt, die jeweils über alle Elemente des Körpers iterieren. Innerhalb dieser Schleifen wird dann der jeweilige Multiplikations-Algorithmus aufgerufen. Während die beiden Multiplikations-Algorithmen ihre 2^{2l} Multiplikationen durchführen, wird jeweils mittels der *"time"*-Bibliothek die Zeit gestoppt, die die beiden Algorithmen benötigen.

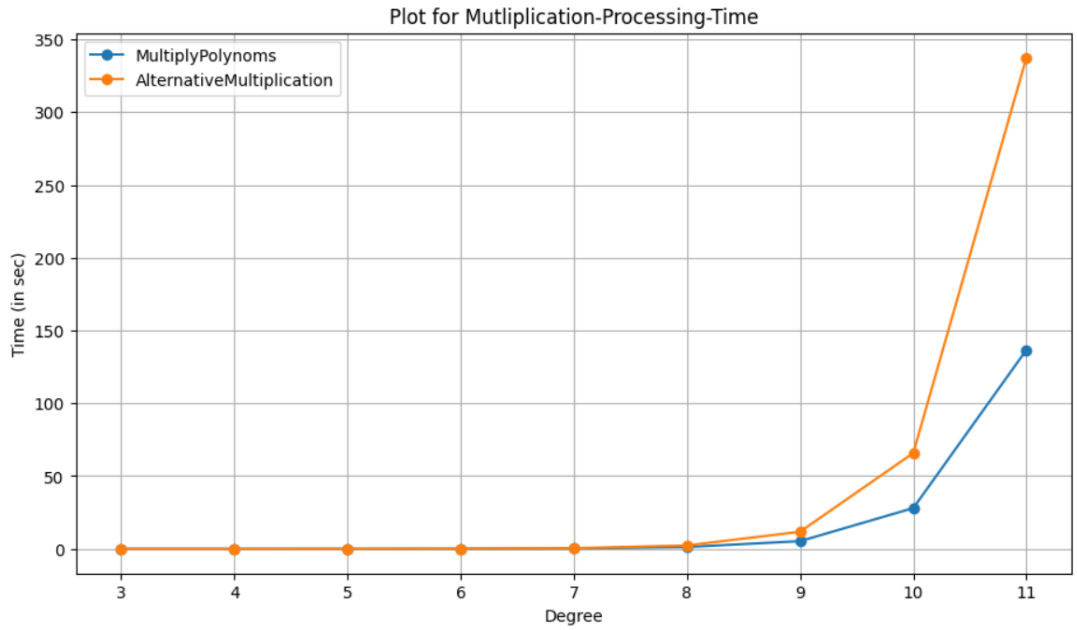
Für das Beispiel, dass der Grad des Minimalpolynoms 8 beträgt, ergibt dieser Test, dass alle 65.536 Ergebnisse der beiden Algorithmen übereinstimmen. Dieser Test kann auch für Minimalpolynome mit höheren Graden durchgeführt werden. Sobald der *'mistake_counter'* bei 0 bleibt und der *'correct_counter'* bei 2^{2l} liegt, bedeutet das, dass beide Multiplikations-Algorithmen übereinstimmen.

```python
start1 = t.time()

for F in List_of_all_tupels:
    for G in List_of_all_tupels:
        result1 = reduce_product(multiply_polynoms(F, G), M)

end1 = t.time()

start2 = t.time()

for F in List_of_all_tupels:
    for G in List_of_all_tupels:
        result2 = alternative_multiplication(F, G, M)

end2 = t.time()

print(f"MultiplyPolynoms:          {(end1 - start1):.5f} Sekunden")
print(f"AlternativeMultiplication: {(end2 - start2):.5f} Sekunden")
```

```
MultiplyPolynoms:          1.39839 Sekunden
AlternativeMultiplication: 2.78688 Sekunden
```

Anschließend werden die beiden Algorithmen bezüglich ihrer Ausführungszeit verglichen. Dazu werden die 2^{2l} Multiplikationen für beide Multiplikations-Algorithmen einzeln durchgeführt, wobei jeweils ein Zeitpunkt vor und ein Zeitpunkt nach der Berechnung festgehalten wird. Dann wird die Differenz aus diesen Zeitpunkten bestimmt. Diese Differenz stellt dann die Zeit dar, die der jeweilige Algorithmus für die Berechnung der 2^{2l} Multiplikationen benötigt hat. Für das Beispiel, dass das Minimalpolynom den Grad 8 hat, benötigt der erste Algorithmus gerundet 1.4 Sekunden, wobei der andere Algorithmus gerundet 2.8 Sekunden braucht.

Anhand dieser Messwerte kann man vermuten, dass der Algorithmus *"MultiplyPolynoms"* effizienter ist, als der Algorithmus *"AlternativeMultiplication"*. Aufgrund dieser Vermutung wurden die Zeiten der beiden Algorithmen für Grade von 3 bis 11 gemessen. Diese Messwerte sind in der folgenden Darstellung grafisch dargestellt.



Anhand dieser Grafik erkennt man, dass der Algorithmus *"AlternativeMultiplication"* (für Grade von 9 bis 11) um einen Faktor von ungefähr zwei langsamer ist. Da die Graphen jedoch exponentiell steigen, wird der Algorithmus *"MultiplyPolynoms"* mit steigenden Graden immer effizienter im Vergleich zu dem Algorithmus *"Alternative-Multiplication"*.

3.5.2 Vergleich Minimalpolynom- mit Cantor-Zassenhaus-Algorithmus

Da Minimalpolynome gleichbedeutend sind mit irreduziblen Polynomen, überprüfen beide Algorithmen *"MinimalPolynoms"* und der Cantor-Zassenhaus-Algorithmus Polynome auf dieselbe Eigenschaft. Dabei bietet sich an, die Performance der beiden Algorithmen zu vergleichen.

```
length = 9

list_of_all_tupels = list(product([0, 1], repeat=length))[4:]

correct_counter, mistake_counter = 0, 0

for F in list_of_all_tupels:

    if is_minpol(F) == cantor_zassenhaus(F):
        correct_counter = correct_counter + 1

    else:
        mistake_counter = mistake_counter + 1

print(f"Falsch: {mistake_counter}")
print(f"Korrekt: {correct_counter}")
```

```
Falsch: 0
Korrekt: 508
```

Zu Beginn wird getestet, ob die beiden Algorithmen auch die gleichen Ergebnisse beim Überprüfen der Polynome auf Irreduzibilität liefern. Dabei werden zuerst alle Polynome, dargestellt als Tupel, bis zur Länge *'length'* erstellt und in der Liste *'list_of_all_tuples'* gespeichert. Von dieser Liste werden nun die ersten vier Elemente weggeschnitten, da der Grad dieser Elemente unter 2 ist und es kein Minimalpolynom in Körpern der Charakteristik zwei gibt, dessen Grad kleiner als zwei ist. Genauer gesagt sind diese vier Elemente die Polynome 0, 1, X und $X + 1$.

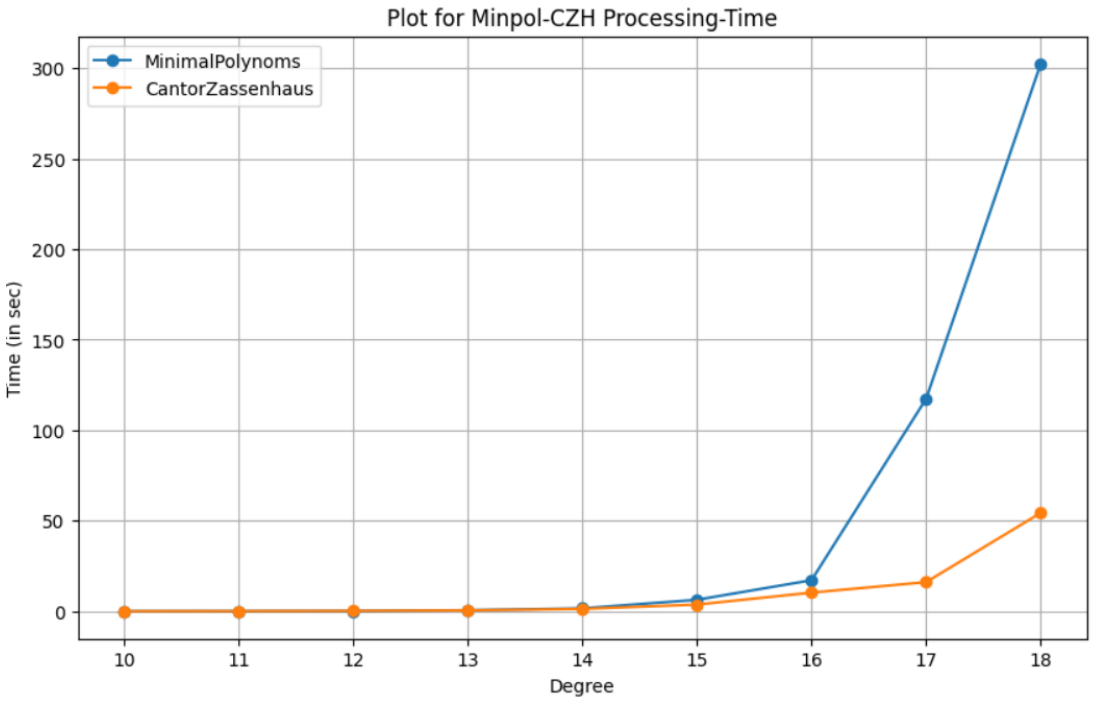
Nun werden zwei Zähler definiert, die jeweils mit dem Wert 0 initialisiert werden. Diese Zähler dienen dazu, mitzuzählen, wie viele Ergebnisse der beiden Algorithmen übereinstimmen, bzw. wie viele Ergebnisse nicht übereinstimmen.

Als nächstes werden die beiden Algorithmen auf jedes der Elemente aus der Liste *'list_of_all_tuples'* angewendet. Sollten die beiden Ergebnisse gleich sein, wird der *'correct_counter'* eins hochgezählt. Sollten die Ergebnisse jedoch nicht gleich sein, so wird der *'mistake_counter'* eins hochgezählt. Dann werden die Werte der beiden Zähler ausgegeben. Das Beispiel $length = 9$ ergibt, dass alle der 508 Ergebnisse der beiden Algorithmen gleich sind, da der Zähler für Fehler bei 0 geblieben ist.

```python
start1 = t.time()

for F in list_of_all_tupels:
        A = is_minpol(F)

end1 = t.time()

start2 = t.time()

for F in list_of_all_tupels:
        B = cantor_zassenhaus(F)

end2 = t.time()

print(f"MinimalPolynoms:  {(end1 - start1):.5f} Sekunden")
print(f"CantorZassenhaus: {(end2 - start2):.5f} Sekunden")
```

```
MinimalPolynoms:  0.00936 Sekunden
CantorZassenhaus: 0.02351 Sekunden
```

Als nächstes wird, analog zur Zeitmessung der beiden Multiplikationen, die Zeit gemessen, die die beiden Algorithmen jeweils benötigen, um die 508 Polynome auf Irreduzibilität zu überprüfen. Da erkennt man, dass der Cantor-Zassenhaus für diese Anzahl und Länge an Polynomen länger braucht als der Algorithmus *"MinimalPolynoms"*. Dies liegt jedoch an den zusätzlichen Checks, die zu Beginn des Cantor-Zassenhaus-Algorithmus gemacht wurden. Im Folgenden werden Zeitmessungen für Polynome mit höheren Graden durchgeführt.



In diesem Graph sieht man, wie lange die jeweiligen Algorithmen brauchen, um alle Tupelkombinationen einer bestimmten Länge auf Irreduzibilität zu überprüfen. Dabei erkennt man, dass der Cantor-Zassenhaus-Algorithmus für steigende Längen der Tupel deutlich effizienter ist, als der Algorithmus *"MinimalPolynoms"*. Dies liegt hauptsächlich daran, dass bei dem Algorithmus *"MinimalPolynoms"* alle Minimalpolynome bis zum Grad $\left\lfloor \frac{deg(F)}{2} \right\rfloor$ berechnet werden müssen, um ein Polynom auf Irreduzibilität zu überprüfen, während der Cantor-Zassenhaus-Algorithmus ohne solch einen aufwendigen Zwischenschritt auskommt.

4 Fazit

4.1 Zusammenfassung

Im Rahmen dieser Studienarbeit wurde ein ausführlicher Einblick in die Arithmetik endlicher Körper gegeben, wobei der Fokus besonders auf endlichen Körpern der Charakteristik zwei lag. Ausgehend von den theoretischen Grundlagen allgemeiner und endlicher Körper wurden die Rechenregeln mittels Polynomen beschrieben. Aufgrund der Eigenschaft, dass bei endlichen Körpern der Charakteristik zwei $1+1=0$ gilt und da diese Eigenschaft praxisnah an der Funktionsweise von Computern ist, wurde auf diese Körper speziell eingegangen und die Rechenoperationen anhand von Beispielen verdeutlicht.

Darauf aufbauend wurden diese theoretischen Betrachtungen mittels Python praktisch umgesetzt. Dabei wurden für die jeweiligen theoretischen Rechenregeln Algorithmen erstellt, um die jeweiligen Operationen mithilfe von Python berechnen zu können. Schließlich wurde die Performance von Algorithmen, die dieselbe Operation abbilden, miteinander verglichen, um herauszufinden, welcher Algorithmus effizienter ist.

Zusammenfassend kann man sagen, dass diese Arbeit die Arithmetik endlicher Körper sowohl auf theoretischer als auch auf praktischer Ebene genau beleuchtet hat und aufgezeigt hat, wie sich endliche Körper, speziell der Charakteristik zwei, verhalten, da sie eine Basis für bedeutende kryptografische Verfahren darstellen.

4.2 Ausblick

Als Ausblick können zwei verschiedene Aspekte aufgeführt werden, die in Zukunft noch behandelt werden können. Zum einen ist es eine Überlegung, den Cantor-Zassenhaus-Algorithmus zu verbessern, um die Bestimmung von irreduziblen Polynomen weiter zu optimieren. Dieser Algorithmus berechnet den ggT von dem zu überprüfenden Polynom $F(X)$ und Polynomen der Form $X^{p^l} - X$ (wobei in der Charakteristik zwei gilt: $X^{2^l} + X$). Dabei ist l abhängig von dem Grad des Polynoms $F(X)$, denn der ggT der beiden Polynome muss für l von 1 bis $\left\lfloor \frac{n}{2} \right\rfloor$ bestimmt werden, wobei n der Grad des Polynoms $F(X)$ ist.

Potenzial für Optimierung liegt hier genau in dem Polynom $X^{2^l} + X$, denn für steigende l ändert sich an dem Polynom in Tupeldarstellung lediglich die Anzahl an Nullen, zwischen den beiden Monomen mit Vorfaktor 1. Mit zunehmender Größe

von l wird der Cantor-Zassenhaus-Algorithmus immer länger brauchen, obwohl das Polynom $X^{2^l} + X$ in Tupeldarstellung nur um Informationen angereichert wird, die nicht unbedingt benötigt werden. Daher wäre eine Möglichkeit der Optimierung des Cantor-Zassenhaus-Algorithmus, das Polynom $X^{2^l} + X$ nicht wie bisher in der aufgeführten Tupeldarstellung darzustellen, sondern auf eine andere Darstellungsform auszuweichen, die lediglich beinhalten würde, an welchen Stellen sich die Einsen in der Tupeldarstellung befinden.

Beispielsweise kann das Polynom $G(X) = (1, 0, 0, 0, 0, 0, 0, 1, 0)$, für $(X^{2^4} + X)$ in Form von $(9, 1, 8)$ dargestellt werden, wobei die erste Stelle des Tupels die Länge des ursprünglichen Tupels darstellt und die beiden darauffolgenden Stellen die Positionen der beiden Einsen des ursprünglichen Tupels signalisieren. (In diesem Beispiel würde beim Zählen der Stellen des Tupels bei 1 angefangen, man könnte auch bei 0 mit dem Zählen anfangen.)

Mit dieser Schreibweise könnte man jedes Polynom der Form $X^{2^l} + X$ lediglich durch ein Tupel mit 3 Elementen darstellen, egal wie lange es in der ursprünglichen Tupeldarstellung wäre, da Polynome der Form $X^{2^l} + X$ immer nur an genau zwei Stellen eine 1 haben. Sobald diese Darstellung eingeführt wurde, müssen die Funktionen, die zur Berechnung des ggT benötigt werden, so angepasst werden, dass sie mit Tupeln der neuen Darstellungsform arbeiten können. Dadurch würde der Cantor-Zassenhaus-Algorithmus deutlich effizienter werden, da nicht bei jeder ggT-Berechnung und damit bei jeder Polynomdivision durch alle Stellen eines Tupels der Form $X^{2^l} + X$ iteriert werden, wobei an den meisten Stellen Nullen stehen und man so unnötige Wiederholungen vermeiden kann.

Der zweite Aspekt des Ausblicks ist eine Weiterführung des Themas, und zwar geht es dabei um quadratische Gleichungen in endlichen Körpern. (Auch hier kann der Fokus auf endliche Körper der Charakteristik zwei gelegt werden.) In solchen Körpern nehmen quadratische Gleichungen der Form $x^2 + ax + b = 0$ eine besondere Form an, da zum einen der Vorzeichenwechsel entfällt und zum anderen die Gleichung $1 + 1 = 0$ in der Charakteristik zwei gilt. Dadurch ist die klassische Mitternachtsformel hierbei nicht anwendbar, und stattdessen müssen spezielle Verfahren wie die Trace-Funktion (vgl. [9]) benutzt werden.

5 Liste an Minimalpolynomen

Grad	Minimalpolynom	Grad	Minimalpolynom
2	X^2+X+1	7	$X^7+X^6+X^5+X^3+X^2+X+1$
3	X^3+X+1		$X^7+X^6+X^5+X^4+1$
	X^3+X^2+1		$X^7+X^6+X^5+X^4+X^2+X+1$
4	X^4+X+1		$X^7+X^6+X^5+X^4+X^3+X^2+1$
	X^4+X^3+1	8	$X^8+X^4+X^3+X+1$
	$X^4+X^3+X^2+X+1$		$X^8+X^4+X^3+X^2+1$
5	X^5+X^2+1		$X^8+X^5+X^3+X+1$
	X^5+X^3+1		$X^8+X^5+X^3+X^2+1$
	$X^5+X^3+X^2+X+1$		$X^8+X^5+X^4+X^3+1$
	$X^5+X^4+X^2+X+1$		$X^8+X^5+X^4+X^3+X^2+X+1$
	$X^5+X^4+X^3+X+1$		$X^8+X^6+X^3+X^2+1$
	$X^5+X^4+X^3+X^2+1$		$X^8+X^6+X^4+X^3+X^2+X+1$
6	X^6+X+1		$X^8+X^6+X^5+X+1$
	X^6+X^3+1		$X^8+X^6+X^5+X^2+1$
	$X^6+X^4+X^2+X+1$		$X^8+X^6+X^5+X^3+1$
	$X^6+X^4+X^3+X+1$		$X^8+X^6+X^5+X^4+1$
	X^6+X^5+1		$X^8+X^6+X^5+X^4+X^2+X+1$
	$X^6+X^5+X^2+X+1$		$X^8+X^6+X^5+X^4+X^3+X+1$
	$X^6+X^5+X^3+X^2+1$		$X^8+X^7+X^2+X+1$
	$X^6+X^5+X^4+X+1$		$X^8+X^7+X^3+X+1$
	$X^6+X^5+X^4+X^2+1$		$X^8+X^7+X^3+X^2+1$
7	X^7+X+1		$X^8+X^7+X^4+X^3+X^2+X+1$
	X^7+X^3+1		$X^8+X^7+X^5+X+1$
	$X^7+X^3+X^2+X+1$		$X^8+X^7+X^5+X^3+1$
	X^7+X^4+1		$X^8+X^7+X^5+X^4+1$
	$X^7+X^4+X^3+X^2+1$		$X^8+X^7+X^5+X^4+X^3+X^2+1$
	$X^7+X^5+X^2+X+1$		$X^8+X^7+X^6+X+1$
	$X^7+X^5+X^3+X+1$		$X^8+X^7+X^6+X^3+X^2+X+1$
	$X^7+X^5+X^4+X^3+1$		$X^8+X^7+X^6+X^4+X^2+X+1$
	$X^7+X^5+X^4+X^3+X^2+X+1$		$X^8+X^7+X^6+X^4+X^3+X^2+1$
	X^7+X^6+1		$X^8+X^7+X^6+X^5+X^2+X+1$
	$X^7+X^6+X^3+X+1$		$X^8+X^7+X^6+X^5+X^4+X+1$
	$X^7+X^6+X^4+X+1$		$X^8+X^7+X^6+X^5+X^4+X^2+1$
	$X^7+X^6+X^4+X^2+1$		$X^8+X^7+X^6+X^5+X^4+X^3+1$
	$X^7+X^6+X^5+X^2+1$		

Tabelle mit Minimalpolynomen bis zum Grad 8

Literatur

[1] Siegfried Bosch. *Algebra*. 10. Aufl. 2023. Berlin, Heidelberg: Springer Berlin Heidelberg, 2023. 1 S. ISBN: 978-3-662-67464-2. DOI: 10.1007/978-3-662-67464-2.

[2] David G. Cantor und Hans Zassenhaus. „A new algorithm for factoring polynomials over finite fields“. In: *Mathematics of Computation* 36.154 (1981), S. 587–592. ISSN: 0025-5718, 1088-6842. DOI: 10.1090/S0025-5718-1981-0606517-5. URL: https://www.ams.org/mcom/1981-36-154/S0025-5718-1981-0606517-5/ (besucht am 05. 03. 2025).

[3] *Elliptic-curve cryptography*. In: *Wikipedia*. Page Version ID: 1282815573. 28. März 2025. URL: https://en.wikipedia.org/w/index.php?title=Elliptic-curve_cryptography&oldid=1282815573 (besucht am 07. 04. 2025).

[4] *Endliche Körper: Definition, Anwendung*. StudySmarter. URL: https://www.studysmarter.de/studium/mathematik-studium/algebra-studium/endliche-koerper/ (besucht am 07. 04. 2025).

[5] *Endlicher Körper*. In: *Wikipedia*. Page Version ID: 252551010. 23. Jan. 2025. URL: https://de.wikipedia.org/w/index.php?title=Endlicher_K%C3%B6rper&oldid=252551010 (besucht am 07. 04. 2025).

[6] Ernst Kunz. *Algebra*. Vieweg studium 43. Braunschweig: Vieweg, 1991. ISBN: 978-3-528-07243-8.

[7] Hans Kurzweil. *Endliche Körper: Verstehen, Rechnen, Anwenden*. 2. Aufl. 2008. Springer-Lehrbuch. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. 1 S. ISBN: 978-3-540-79598-8. DOI: 10.1007/978-3-540-79598-8.

[8] Serge Lang. *Algebra*. Rev. print. Addison-Wesley series in mathematics. Reading, Mass.: Addison-Wesley, 1971. 526 S. ISBN: 978-0-201-04177-4.

[9] *Spur (Mathematik)*. In: *Wikipedia*. Page Version ID: 250251451. 11. Nov. 2024. URL: https://de.wikipedia.org/w/index.php?title=Spur_(Mathematik)&oldid=250251451 (besucht am 07. 04. 2025).

[10] Unbekannt. *Die Einheitengruppe von Z/nZ und Primitivwurzeln modulo*. Online verfügbar unter: https://agag-lassueur.math.rptu.de/~lassueur/en/teaching/EZTSS17/EZTSS17/Kap5.pdf. 2017.

[11] Stefan Waldmann. *Lineare Algebra 1: Grundlagen für Studierende der Mathematik und Physik*. 2. Aufl. 2021. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021. 1 S. ISBN: 978-3-662-63263-5. DOI: 10.1007/978-3-662-63263-5.

[12] *Was ist Elliptische-Kurven-Kryptografie (ECC)? - Definition von Computer Weekly*. ComputerWeekly.de. URL: https://www.computerweekly.com/de/definition/

Elliptische-Kurven-Kryptografie-Elliptic-Curve-Cryptography-ECC (besucht am 07. 04. 2025).

[13] Dirk Werner. *Lineare Algebra*. Grundstudium Mathematik. Cham, Switzerland: Birkhäuser, 2022. 317 S. ISBN: 978-3-030-91107-2.

Quellcode

Für diese Arbeit wurde ein praktischer Teil entwickelt, der die Funktionen, die im Implementierungsteil beschrieben werden, enthält. Diese Funktionen wurden mit der Programmiersprache Python mithilfe von Jupyter Notebooks implementiert. Dazu ist der vollständige Quellcode im folgenden GitHub-Repository verfügbar:

Link: https://github.com/Larsk7/ArithmetikEndlicherKoerperF2

Beschreibung: In dem Ordner 'Jupyter_Notebook_Files' befinden sich die Jupyter-Notebook-Dateien. Darin befinden sich drei Notebooks, wobei eines ('TestNotebook.ipynb') zum Testen der Funktionen ist und zwei weitere ('CompareMultiplications.ipynb' und 'CompareMinPol-CZH.ipynb'), die für die Performancevergleiche der Multiplikationsalgorithmen bzw. der Überprüfung auf Irreduzibilität von Polynomen sind. Zudem befindet sich dort der Ordner 'Functions', in dem zum einen die Hilfsfunktionen ('AuxiliaryFunctions.ipynb') und zum anderen die in der Ausarbeitung beschriebenen Funktionen einzeln implementiert sind.

Zugang: Das Repository ist öffentlich zugänglich (für Lesezugriffe).

Letzte Aktualisierung (des Ordners: 'Jupyter_Notebook_Files'): 21.02.2025
(CommitID: 6a8fe2cf4b28acda2525c207505bdb539bef6922)