Florian HEIGL

Digit expansions in cryptography and bounds for minimal weight expansions

DIPLOMARBEIT

zur Erlangung des akademischen Grades eines Diplom-Ingenieurs

Diplomstudium Technische Mathematik

Technische Universität Graz

Betreuer:
Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Clemens HEUBERGER

Institut für Optimierung und Diskrete Mathematik

Graz, im Dezember 2011

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am . . . .

(Unterschrift)

Ich möchte mich bei allen Personen bedanken, die mich bei und während der Erstellung dieser Arbeit unterstützt haben. Allen voran danke ich meinen Eltern für ihre Unterstützung während meines gesamten Studiums. Dank gilt auch meinem Betreuer, Prof. Heuberger, für seine Geduld und seinen Einsatz.

Graz, am 13.12.2011

Contents

1 Introduction

The exchange of information between two or more entities is one of the cornerstones of our world. From an anthropological point of view it is the basis of all human interaction and happens in many forms that are not restricted to written or spoken language. But not only humans communicate. Machines such as computers, cell phones and (more recently) even cars use various strictly defined protocols for the exchange of information with each other, connected peripherals or their users.

One important consideration in this context is the level of confidentiality of the information exchanged. Two people talking about some shared secret they want to keep confidential, can achieve this by avoiding the presence of a third person within earshot. In a more general setting the same is not possible and we have to assume that the information is exchanged via a potentially insecure channel such as a telephone line, the postal service, radio communication, a wide area computer network such as the Internet or a wireless local network.

The field of cryptography addresses precisely this problem. Whilst having been developed and used mostly in a military context for hundreds of years, the technological advances of the 20th century introduced cryptographic applications into the everyday life of most people. Whenever we communicate with our cellphones, send secure e-mails, log on to on-line platforms for eg. electronic banking or use the chip on a smartcard (such as an ATM-card, a modern passport or the Austrian E-card), we make use of a broad range of cryptographic algorithms and protocols that protect our privacy.

1.1 Historic remarks and outlook

Before the dawn of the digital age, historic cryptosystems usually operated on texts of human language. Cryptography was then more related to linguistics and encoding a given text was based on substitutions and transformations of letters of the alphabet.

The 20th century saw a big change in cryptographic applications due to the inception of computing devices. These were at first more mechanical than electrical machines that were used both in the encryption and decryption of information, but also used to break certain cryptosystems[1], which means finding the secret key based solely on the knowledge of certain plain texts, cipher texts or details of the encryption mechanism.

Since then the continued improvement of computers has had the consequence that cryptosystems had to become more complex to prevent an easy algorithmical break of the system by an automated statistical analysis of plain and ciphertext pairs or by a blunt brute-force attack that tries out all possible keys until successful. This has lead to a stronger influence of modern mathematics in cryptography.

An important milestone in the history of cryptography is the inception of asymmetric cryptosystems in the 1970s. All the previous cryptosystems had the common property

[1]A notable example are the ultimately successful efforts of British cryptologists during the Second World War to break the German Enigma code with the help of a machine that has become known as the *bombe*.

that if the parties A and B want to exchange any encrypted information, they both had to be in possession of a shared secret key, which is used for both en- and decryption of the information. Hence these systems are called symmetric cryptosystems. The weakness of this system lies in the fact that in order to agree on such a key, the two parties have to first communicate it via a secure channel.

Asymmetric cryptosystems work with different keys for encryption and decryption. If A participates in an asymmetric cryptosystem, it has two keys k_{pub} and k_{priv}, where k_{priv} is the *private key*, which only A possesses and k_{pub} is the so-called *public key* that A publishes for everyone to see. If anybody wants to send an message p to A, this is done by encryption with k_{pub}, i.e. the ciphertext c is computed via

$$c = E_{k_{\mathrm{pub}}}(p).$$

Upon receiving c, A can decrypt the message with its private key k_{priv}, ie. computing

$$p = D_{k_{\mathrm{priv}}}(c) = D_{k_{\mathrm{priv}}}(E_{k_{\mathrm{pub}}}(p)).$$

In order for this idea to work, we need an encryption function E_k that can not easily be inverted, even if k is known. These functions are sometimes called "trap-door"-functions and an important example is the computation of powers in a finite field. If $\mathbb{F}_q$ is a finite field with $q = p^n$ for some $n \in \mathbb{N}$ and a prime p and g is a prime element in $\mathbb{F}_q$, then it is hard to compute k given only g and g^k. This is known as the discrete logarithm problem (DLP).

A cryptographic protocol employing the discrete logarithm problem is the one proposed by Whitfield Diffie and Martin Hellman in 1976. It enables two parties A and B to agree on a shared key without any previously shared keys using only asymmetric methods. The first asymmetric cryptosystem used for the exchange of information is the RSA cryptosystem (named after Ronald Rivest, Adi Shamir and Leonard Adleman, who presented it in 1978). The security of the RSA system depends on the difficulty of computing the prime factorization for large numbers $n = pq$, where p and q are large primes and only the product is known.

From a mathematical standpoint, in comparison to symmetric cryptosystems, the theory behind asymmetric cryptography has a much stronger connection to many fascinating mathematical fields such as abstract algebra and number theory. Conversely, the need for computerized analysis of possible weaknesses and other aspects of these systems has inspired advances in several fields such as computational number theory. For an overview of mathematical cryptography as well as an historic account see e.g. [15].

For the rest of this chapter, we will discuss some of the basics of elliptic curve cryptosystems and why their performance has a strong connection to different representation of integers, before presenting some results about Markov chains, which we will need later.

Afterwards, we will focus our attention entirely on these representations of integers as digit expansions with various sets of digits and two different bases. In Chapter 2 we will study different well known expansions. We will present their definitions and various algorithms used to compute them. Of special interest for the performance of elliptic curve operations will be the number of nonzero digits in the digit expansions of any given integer. In Chapter 3 we will focus on one basic type of expansion and study different representatives of this class. Additionally we will present real-world data concerning the effects on performance of an elliptic curve cryptosystem. For this we implemented all the conversions for the different representations and used a cryptographic computer library to test the speed of the cryptographic operations for each of them. The empiric timing results solidify the theoretical knowledge gathered in Chapter 2.

In the final chapter of this document we want to find bounds for what the best digit expansions can achieve on average in terms of a low number of non-zero digits. For some cases, where the standard expansion already achieve this optimal value, this will already have been answered in 2. For other cases, however it has been shown that it is impossible for any of the well-known expansions to achieve optimality. We quantify what the performance of an optimal expansion can be. We do so by computing these optimal expansions in a manner that is not efficient enough for practical use, but provides us with an algorithm that we can then model and analyze in order to compute the average weight of these optimal expansions. For this we use a Markov chain model of our algorithm. Our main result is Theorem 4, where we state the explicit and exact value of the optimal average number of non-zero digits for three scenarios, where it has not been previously known.

1.2 Elliptic curve scalar multiplication and the role of digit expansions

Elliptic curves have been a topic of a great amount of research even before their use for cryptographic purposes was presented by Neil Koblitz in [17]. Their cryptographic relevance stems from the fact that they allow for a formulation of an analog to the discrete logarithm problem, called the *elliptic curve discrete logarithm problem* (ECDLP).

An elliptic curve $E(K)$ over a field K is defined as the set of solutions $(x, y) \in K$ to Equation (1) if $\mathrm{char}(K) \notin \{2, 3\}$ or Equation (2) if $\mathrm{char}(K) = 2$.

$$y^2 = x^3 + ax + b \qquad (a, b \in K) \tag{1}$$

$$y^2 + cxy + dy = x^3 + ax + b \qquad (a, b, c, d \in K) \tag{2}$$

It is well known ([18], [15]) that the points on these curves, together with the so-called *point at infinity* $\mathcal{O}$ form an Abelian group, where the group operation is written additively. For cryptographic applications we are specifically interested in elliptic curves over finite fields $\mathbb{F}_q$, where $q = p^n$ for some prime number p. In this case $E(\mathbb{F}_q)$ is a finite abelian group and it can be shown that it is either cyclic, or the direct product of two cyclic groups. As Koblitz states, one of the two groups is much smaller than the other for most randomly generated curves. For cryptographic applications one typically works in a large cyclic subgroup $(G, +)$ of $(E(\mathbb{F}_q), +)$.

When given a curve point P in $(G, +)$, we can compute multiples of P by defining $nP := \sum_{k=1}^{n} P$. This operation is called elliptic curve scalar multiplication. If r is the order of G and P is a primitive element in G, then the elliptic curve discrete logarithm problem is finding $k \in \{1, 2, \ldots, r-1\}$ such that $kP = Q$, when only knowing P and Q beforehand.

Since EC cryptosystems consequently all involve such scalar multiplications as the basic operation, finding good strategies for computing them is a crucial step in making EC cryptosystems fast and efficient. In this section, we will see why this problem is very closely connected to investigating digit expansions of the given scalar n.

As a starting point, we consider two basic arithmetic operations that we can perform on a curve point P. They are addition and doubling. This approach can immediately be improved by using the doubling operation in combination with additions of P. In fact, elliptic curve scalar multiplication is analogous to modular exponentiation, where the operations squaring and multiplication correspond to doubling and addition of the exponents. Therefore many well-known techniques carry over to the problem at hand. The standard algorithm is known as the *binary method* (see [16]) which describes a combination of additions and doublings according to the binary expansion of k. So, as an example, for $n = 7 = (111)_2$, one would compute $7P$ as $2(2P + P) + P$ and only need two additions and two doublings instead of six addition.

A property specific to elliptic curves is the fact that subtraction of a point P is computationally just as cheap as addition. This can be used to modify the binary method to allow subtractions of P whenever this increases the efficiency of the operation. In our example above, instead of $2(2P + P) + P$ (two additions), we could simply write $7P$ as $2(2(2(P))) - P$ and compute the result with just one subtraction and three doublings. This corresponds to the idea of allowing -1 as a digit in the expansion of n, which has first been presented by Morain and Olivos in [19]. Doing so introduces ambiguity in the digit representation and such *signed binary expansions* with digits $\{0, \pm 1\}$ have been widely studied with the goal to find representations that lead to fast scalar multiplication. Another extension to the concept of digit expansions comes from the idea of precomputing $\eta_i \cdot P$ for some small values $\eta_i \in \mathbb{Z}$ and also including these in the set of possible digits for the expansion. The idea behind this is to reduce the overall number of non-zero digits (and therefore additions in the scalar multiplication), at the memory and time cost associated with the precomputation of the values $\eta_i \cdot P$.

All the concepts above use different ways of adding multiples of P in the computation combined with doubling. Hence all these expansions use 2 as the base of the expansion. We will study these *base* 2 *digit expansions* in Section 2.1. A completely different approach, suggested by Koblitz (in [17]), works with a certain type of elliptic curves now referred to as Koblitz curves. They come equipped with a endomorphism on the curve points that acts like multiplication by a complex number τ. This can be used to represent scalars with digit expansions to the base τ and replace the *double*-and-add paradigm of the binary method with a τ-and-add strategy. Just as in the case of base 2, we can again make use of subtraction of points and a larger digit set combined with precomputations. We will study these *base* τ *digit expansions* in Section 2.2.

A modification of the original problem of scalar multiplication is the computation of linear combinations $m \cdot P + n \cdot Q$ for integers m and n and curve points P and Q. These are needed in the verification algorithm for elliptic curve digital signatures. Instead of computing mP and nQ individually and adding the result, one can use a method proposed by Strauss [25] (alternatively referred to as Shamir's trick) in which one goes through the binary expansions of m and n simultaneously and adds P, Q or $P + Q$ to the result in between doubling operations. The expansions of m and n, put together, can be seen as a *joint* expansion of the vector $\binom{m}{n}$. Just as in the one-dimensional case, generalizations have been made to allow digit sets other than $\{0, 1\}$ and τ as a base instead of 2. We will look at some properties and techniques for joint expansions in Section 2.3. To formalize the ideas above we define the general form of an expansion:

Definition 1.1. A base b *(digit-)expansion* of an integer n is a sequence $\varepsilon = (\varepsilon_j)_{j\in\mathbb{N}_0} = (\ldots, \varepsilon_2, \varepsilon_1, \varepsilon_0)$ of digits $\varepsilon_j \in \mathcal{D} \subseteq \mathbb{Z}$ where only finitely many digits are nonzero and

$$n = \text{value}(\varepsilon) := \sum_{j\geq 0} \varepsilon_j b^j.$$

$\mathcal{D}$ is called the digit set of the expansion, $\ell = \max\{j \in \mathbb{N}_0 \mid \varepsilon_j \neq 0\} + 1$ the *length* of the expansion and we write $\ell = \text{length}(\varepsilon)$. The number of nonzero ε_j is called the *weight* of the expansion and written as $\text{weight}(\varepsilon)$.

1.3 Markov chains

Markov chains (see e.g. [21]) are a very common way to model a certain kind of random process, which for a given time n is in a certain state $s \in S$, where S is a countable set. The special property of Markov processes is their lack of memory, i.e. any step in the process depends only on the current state and not on any previous history. We will only be considering the case of discrete time Markov processes, which means that n will be in $\mathbb{N}_0$.
Markov chains will be used in Chapter 4, to compute minimal Hamming weights of integers encoded with special digit sets.

Definition 1.2. Let S be a countable set. A *discrete time Markov chain* is a sequence $(X_n)_{n\geq 0}$ of random variables with values in S, such that for $n \geq 0$ and $s_0, \ldots, s_{n+1} \in S$

$$\mathbb{P}(X_{n+1} = s_{n+1} \mid X_n = s_n, \ldots, X_0 = s_0) = \mathbb{P}(X_{n+1} = s_{n+1} \mid X_n = s_n).$$

We call S the *state space* and the distribution of X_0 the *initial distribution* of the Markov chain. Additionally, if $\mathbb{P}(X_{n+1} = s_j \mid X_n = s_i) = p_{ij}$ is independent of n, we say that the Markov chain is *time homogeneous* and define

$$P = (p_{ij})_{i,j\geq 0},$$

the *transition matrix* for the Markov chain $(X_n)_{n\geq 0}$.

We will only be discussing time homogeneous discrete time Markov chains with a finite state space $S = \{s_0, s_1, \ldots, s_N\}$. They are completely characterisized by giving an initial

distribution λ, written as a row vector $(\lambda_i)_{0\leq i\leq N} = (\mathbb{P}(X_0 = s_i))_{0\leq i\leq N}$ and the transition matrix P. The matrix P corresponds to a directed graph where the vertices are the elements of the state space and for every $i, j \in \mathbb{N}_0$ with $p_{ij} > 0$, there is an edge from s_i to s_j, labeled with its *transition probability* p_{ij}.
In a general graph theoretical context, it is a well known property of transition matrices, that their n-th powers provide information about paths of length n. For Markov chains the n-th power of P gives the probability of getting from one state to another in n steps. In other words, setting $P^n = (p_{ij}^{(n)})_{ij}$ for $n \geq 0$, we get

$$\mathbb{P}(X_n = s_j \mid X_0 = s_i) = p_{ij}^{(n)}$$

and combining this with the initial distribution λ, we can compute the distribution of X_n as

$$\mathbb{P}(X_n = s_j) = (\lambda P^n)_j.$$

Definition 1.3. A distribution $\pi = (\pi_1, \ldots, \pi_N)$ for the Markov chain with transition matrix P is called a *stationary distribution* if it satisfies

$$\pi \cdot P = P.$$

Next we will sum up some definition and results that provide information on when a Markov chain has a unique stationary distribution.

Definition 1.4. For two states $s_i, s_j \in S$, we say

- j is *reachable* from i, written $i \to j$, if

$$\mathbb{P}(X_n = s_j \mid X_0 = s_i) > 0 \text{ for some } n \geq 0$$

- i *communicates* with j, written $i \leftrightarrow j$, if $i \to j$ and $j \to i$.

It is easy to see that $\leftrightarrow$ is an equivalence relation on S and therefore partitions it into classes. We call them the *communicating classes* of the Markov chain. A communicating class $C \subseteq S$ is called *closed*, if

$$\forall i \in C : i \to j \Rightarrow j \in C,$$

which means that the the process will never leave this class again. A chain consisting of one single communicating class is called *irreducible.*

Definition 1.5. The *period* of a state s_i is defined as

$$p(s_i) = \gcd\{n \in \mathbb{N} \mid \mathbb{P}(X_n = s_i \mid X_0 = s_i) > 0\}$$

and we call s_i *aperiodic* if $p(s_i) = 1$ and *periodic* otherwise.

Lemma. *All states in one communicating class have the same period.*

Definition 1.6. An irreducible Markov Chain is aperiodic if and only if one of its states is aperiodic.

Theorem. *Let $(X_n)_{n\geq 0}$ be a finite state space irreducible Markov chain with transition matrix P. Then there exists a unique stationary distribution π and if $(X_n)_{n\geq 0}$ is aperiodic, we get that for all initial distributions λ,*

$$\lim_{n\to\infty} \lambda \cdot P^n = \pi$$

2 Digit expansions — An overview

2.1 Base 2 digit expansions

Definition 2.1. A *signed binary expansion* of an integer n is a base 2 digit expansion with digit set $\mathcal{D} = \{0, \pm 1\}$.

As discussed in Section 1.2, efficient scalar multiplication of an elliptic curve point P with an integer n using only the two operations addition and doubling, can by done with the binary method using the binary expansion of n. Modifying it to allow subtractions of P yields Algorithm 1. It takes signed binary expansions as input.

Algorithm 1 (Signed) Binary Method

```
Input:  Signed binary expansion $\varepsilon = (\varepsilon_{j-1}, \dots, \varepsilon_1, \varepsilon_0)$ of an integer $n$, curve point $P$
Output:  Point $Q$ such that $Q = nP$
  $Q := 0$
  $k := j - 1$
  while $k \geq 0$ do
    $Q := 2Q$
    if $\varepsilon_k = 1$ then
      $Q := Q + P$
    else if $\varepsilon_k = -1$ then
      $Q := Q - P$
    end if
    $k := k - 1$
  end while
```

The doubling step is performed in every iteration, so the total number of doublings coincides with length(ε). The additions or subtractions are performed for every nonzero digit in the input expansion. The number of these is given by the Hamming weight of the expansion and we will write it as weight(ε). Since the representation of an integer by a signed binary representation is by no means unique, it is the next logical step to find such representations with minimal weight, to optimize the performance of scalar multiplication.

2.1.1 The non-adjacent form

Definition 2.2. Then *non-adjacent form* (NAF, sometimes also "balanced binary representation") is a signed binary expansion with the additional property that there are no adjacent (hence the name) nonzero digits:

$$\forall i \in \mathbb{N}\colon \varepsilon_{i+1}\varepsilon_i = 0.$$

It was discovered independently by several authors, we refer to Reitwiesner [22], who also proved that the NAF minimizes weight(ε) under all signed binary expansions ε of any given integer.

The computation of the NAF is quite straightforward. When starting with a standard binary expansion of n, one simply scans it from right to left (least significant bit to most significant bit) and corrects any violations of the NAF-condition by replacing digit-sequences of the form $\underbrace{(1,\dots,1)}_{k\text{ bits}}$ by $(\underbrace{1}_{\text{carry}},\underbrace{0,\dots,0,-1}_{k\text{bits}})$.
Algorithm 2 expresses this in terms of modulo arithmetic.

Algorithm 2 Computing the NAF of n from right to left.

Input: Integer n.
Output: NAF $\varepsilon = (\varepsilon_{j-1}, \varepsilon_{j-2}, \dots, \varepsilon_1, \varepsilon_0)$ of n.

```
$c := n$
$i := 0$
while $c \neq 0$ do
  $\{n = c2^i + \sum_{j=0}^{i-1} \varepsilon_j 2^j\}$
  if $c$ is odd then
    $\varepsilon_i := 2 - c \bmod 4$
    $c := c - \varepsilon_i$
  else
    $\varepsilon_i := 0$
  end if
  $c := c/2$
  $i := i + 1$
end while
```

Algorithm 2 terminates because $|c|$ is decreased in every loop iteration. The loop-invariant $n = c2^i + \sum_{j=0}^{i-1} \varepsilon_j 2^j$ captures the fact that c is processed right-to-left, where $(\varepsilon_{i-1}, \varepsilon_{i-2}, \dots, \varepsilon_1, \varepsilon_0)$ holds the already computet rightmost i digits of the result. As soon as c is 0, we get value$(\varepsilon) = n$. The NAF condition is satisfied, because nonzero digits are only ever prepended to ε in the case where c is odd. Directly afterwards c is modified to satisfy $c = 0 \pmod 4$. Hence c will be even in the next loop iteration and the next NAF-digit will be a zero.

In terms of efficiency impact on scalar multiplication, we are—as stated earlier—mainly interested in the length and the Hamming weight of the expansion. With regards to the length, it is a known result ([23]) that the NAF of an integer can at most be one bit longer than its standard binary expansion. The weight of the NAF for a given integer $n < 2^\ell$ is minimal among all signed binary expansions of the same integer and on average it is $\frac{1}{3}\ell + \mathcal{O}(1)$ as demonstrated in [19], [13] or [14] for an analysis under various input statistics.

2.1.2 Window methods (w-NAF)

It is well known that if additional memory for precomputations is available, the number of additions in the scalar multiplication algorithm can be further reduced by precomputing $\eta \cdot P$ for some small $\eta \in \mathbb{Z}$ and allowing these η as digits in the expansion of n.

Definition 2.3. Let $w \geq 2$, then the *width-w-non-adjacent form* or w-NAF of an integer n is a base 2 integer expansion with the digit set $\mathcal{D}_w = \{0, \pm 1, \ldots, \pm(2^{w-1} - 1)\}$ and the additional property that among any w consecutive digits, there is at most one non-zero digit.

This expansion is a direct consequence of applying the concepts of the *sliding window* method to binary expansion of n. The name comes from the visual image of sliding a window of width w along the expansion from right to left. Arithmetically this is done by choosing digits from $\mathcal{D}$ according to the remainder of n modulo 2^w.

On the other hand, looking at Algorithm 3 (cf. [23]), it becomes obvious that it is also just the logical generalization of the NAF and in fact the ordinary NAF is identical to the w-NAF for $w = 2$.

Algorithm 3 Computing the w-NAF of n from right to left.

```
Input:  Integer $n$, $w \geq 0$
Output:  $w$-NAF $\varepsilon = (\varepsilon_{j-1}, \varepsilon_{j-2}, \ldots, \varepsilon_1, \varepsilon_0)$ of $n$.
  $c := n$
  $i := 0$
  while $c \neq 0$ do
    $\{n = c2^i + \sum_{j=0}^{i-1} \varepsilon_j 2^j\}$
    if $c$ is odd then
      Let $\varepsilon_i \in \mathcal{D}_w$ s.t.  $\varepsilon_i \equiv c \mod 2^w$
      $c := c - \varepsilon_i$
    else
      $\varepsilon_i := 0$
    end if
    $c := c/2$
    $i := i + 1$
  end while
```

As in the case for $w = 2$, the general w-NAF of an integer n is unique, it is at most one bit longer than the binary expansion and it has minimal weight among all integer expansions of n with digits of absolute value smaller then 2^w. ([1], [20]).

2.2 Base τ digit expansions

In [18], Koblitz discussed a class of elliptic curves that permit very efficient scalar multiplication. They are defined over $\mathbb{F}_2$ by the equation

$$E_a\colon Y^2 + XY = X^3 + aX^2 + 1,$$

where $a \in \{0, 1\}$ and we are interested in the group $E_a(\mathbb{F}_{2^n})$ of $\mathbb{F}_{2^n}$-rational points on the curve. The Frobenius map $\varphi\colon \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ that sends x to x^2 can be extended to an automorphism on $E_a(\mathbb{F}_{2^n})$, where $\varphi((x, y)) = (x^2, y^2)$. When using a normal basis representation for $\mathbb{F}_{2^n}$, the computation of this mapping is done by a simple bit-shift and hence virtually cost-free. Koblitz points out that (setting $\mu = (-1)^{1-a}$) φ satisfies the identity

$$\varphi(\varphi(P)) - \mu\varphi(P) + 2 = 0 \tag{3}$$

and therefore can be used to compute $2^\ell P$ efficiently for $\ell \leq 4$ (cf. [18, Section 2]). This would lead to a fast way of dealing with blocks of up to 4 zeros in a binary double-and-add scheme. More importantly however, in Section 6 of the same paper, Koblitz mentions the concept of a *base-φ expansion*, which leads away from doubling-based scalar multiplication altogether.

The idea is further developed by Solinas in [23], where the author concludes that since φ permutes the curve points and satisfies Equation (3), it can be seen as multiplication of P by the complex constant τ, satisfying

$$\tau^2 - \mu\tau + 2 = 0. \tag{4}$$

Explicitly, we get $\tau = \frac{\mu+\sqrt{-7}}{2}$. For the Frobenius map to take the place of the doubling operation in scalar multiplication, it is of interest, to represent any given integer n as

$$n = \sum_{i=0}^{\ell} \varepsilon_i \tau^i, \tag{5}$$

where $\ell \in \mathbb{N}$ and the ε_i are elements of some digit set $\mathcal{D}$ that remains to be found. In fact, as Solinas states, it is worthwhile to regard n as an element in $\mathbb{Z}[\tau]$.

The first algorithm to generate an encoding to the base τ is based on the observation that τ divides an element $c_0 + c_1\tau \in \mathbb{Z}[\tau]$ if and only if c_0 is even. In this case it is easy to see that

$$\frac{c_0 + c_1\tau}{\tau} = \frac{\mu c_0 + 2c_1}{2} - \frac{c_0}{2}\tau.$$

Hence any $n \in \mathbb{Z}[\tau]$ can be uniquely written in the form of (5), with the digits in $\mathcal{D} = \{0, 1\}$ by looking at the remainder modulo 2 and division by τ in between. According to an observation made in [3, Example 2.7], this is just another way of saying that τ is the base of a canonical number system in $\mathbb{Z}[\tau]$. For the scalar multiplication this means that instead of a double-and-add method, we now employ a "τ-and-add"-paradigm.

As in the binary case, it is now possible to allow digits other than $\{0, 1\}$ in a base-τ expansion and resolve the resulting ambiguity in favor of optimal expansions in terms of minimal Hamming weight leading to fast scalar multiplication algorithms.

2.2.1 The τ-NAF

The idea of using point subtractions works just as well for base τ-expansions as in the case for base 2. The corresponding digit set is $\mathcal{D} = \{0, \pm 1\}$ and Solinas uses it to construct the so-called τ *non-adjacent form* or τ-NAF, which he shows to be unique. The natural transcription of the ideas employed in the binary NAF suggest to pick the least significant digit according to the remainder upon division by τ^2. The remainder modulo τ is either 1 or -1 and selecting it in order to guarantee that after subtracting it and dividing by τ, the result is again divisible by τ gives exactly the desired non-adjacency property $\varepsilon_j \varepsilon_{j+1} = 0$.

Algorithm 4 Computing the τ-NAF of n from right to left.

```
Input: $z = z_0 + z_1\tau \in \mathbb{Z}[\tau]$.
Output: $\tau$-NAF $\varepsilon = (\varepsilon_{j-1}, \varepsilon_{j-2}, \ldots, \varepsilon_1, \varepsilon_0)$ of $z$.
  $(c_0, c_1) := (z_0, z_1)$
  $i := 0$
  while $c_0 + c_1\tau \neq 0$ do
    $\{z = (c_0 + c_1\tau)\tau^i + \sum_{j=0}^{i-1} \varepsilon_j \tau^j\}$
    if $c_0$ is odd then
      $\varepsilon_i := 2 - (c_0 - 2c_1 \mod 4)$
      $c_0 := c_0 - \varepsilon_i$
    else
      $\varepsilon_i := 0$
    end if
    $(c_0, c_1) := (c_1 + \mu c_0/2, -c_0/2)$
    $i := i + 1$
  end while
```

Solinas notes that τ-NAFs of a given length have the same average density as the ordinary NAF, because the same bit-strings occur. A formal proof of the minimality of the τ-NAF among all base τ expansions of a given $n \in \mathbb{Z}[\tau]$ with the digit set $\{0, \pm 1\}$ is given in [2].

In contrast to the binary case, the τ-NAF of any given integer n can be roughly twice as long as the binary expansion or the ordinary NAF of n. This problem is already mentioned in [18], but Solinas points out that since in $\mathbb{F}_{2^m}$, an m-fold concatenation of the Frobenius map is the identity map (i.e. $x^{2^m} = x$), we get

$$\tau^m P = P$$

for the curve points $P \in E_a(\mathbb{F}_{2^m})$. It follows that $(\tau^m - 1)P = \mathcal{O}$ and hence, for z_1, z_2 in $\mathbb{Z}[\tau]$ we have $z_1 P = z_2 P$, whenever $z_1 \equiv z_2 \mod (\tau^m - 1)$. This is even sharpened for the subgroup G of $E_a(\mathbb{F}_{2^m})$ of large prime order. With $\delta = \frac{\tau^m - 1}{\tau - 1}$, it follows that $z_1 P = z_2 P$, whenever $z_1 \equiv z_2 \mod \delta$. This is used to define a reduced version of the τ-NAF where for a given $n \in \mathbb{Z}[\tau]$, the reduced τ-NAF is taken to be the τ-NAF of its remainder upon division by δ.

2.2.2 The τ-w-NAF

When extending the ideas of the window method to base τ expansions, our width-w window is represented by looking at the input modulo τ^w rather than 2^w. This motivates looking at the congruence classes of τ^w in $\mathbb{Z}[\tau]$ for possible digit sets. In analogy to only considering odd digits for base 2 extensions, here, only residue classes coprime to τ are of interest. We call a set containing exactly one representative for every prime congruence class modulo τ^w a *reduced residue system* modulo τ^w and a logical choice for a digit set for the τ-w-NAF would be 0 together with such a reduced residue system.

Such a digit set can be used to compute digits according to Algorithm 5. Not surprisingly, it looks very similar to the encoding scheme for the base 2 w-NAF.

Algorithm 5 Computing the τ-w-NAF of $z \in \mathbb{Z}[\tau]$ from right to left.

```
Input: $z \in \mathbb{Z}[\tau]$, $w \geq 0$, a digit set $\mathcal{D} = \{0\} \cup \tilde{\mathcal{D}}$, where $\tilde{\mathcal{D}}$ is a reduced residue system modulo $\tau^w$
Output: $\tau$-$w$-NAF $\varepsilon = (\varepsilon_{j-1}, \varepsilon_{j-2}, \ldots, \varepsilon_1, \varepsilon_0)$ of $z$.
  $c := z$
  $i := 0$
  while $c \neq 0$ do
    $\{z = c\tau^i + \sum_{j=0}^{i-1} \varepsilon_j \tau^j\}$
    if $\tau \mid c$ then
      $\varepsilon_i := 0$
    else
      Let $\varepsilon_i \in \mathcal{D}$ s.t. $\varepsilon_i \equiv c \mod \tau^w$
      $c := c - \varepsilon_i$
    end if
    $c := c/\tau$
    $i := i + 1$
  end while
```

It is clear that the output satisfies the familiar width-w non-adjacency property and we get the same low expected Hamming weight as in the case for base 2. What is unclear however, is whether the algorithm terminates for any given reduced residue system $\tilde{\mathcal{D}}$. This is answered in [3], where the authors describe an algorithm recognizing exactly those digit sets that guarantee the finiteness of Algorithm 5.

Using the fact that every element of $\mathbb{Z}[\tau]$ has a unique base-τ expansion with the digit set $\{0, 1\}$, we see that $\{\sum_{j=0}^{w-1} \varepsilon_j \tau^j \mid \varepsilon_j \in \{0, 1\}\}$ is a complete residue system and there are exactly 2^w congruence classes modulo τ^w. In this system the elements with $\varepsilon_0 = 0$ are the ones divisible by τ, so eliminating them leaves 2^{w-1} prime congruence classes. The set $\mathcal{D}_w = \{0, \pm 1, \ldots, \pm(2^{w-1} - 1)\}$ which was used for the base 2 w-NAF also contains exactly one representative for every prime congruence class module τ^w (and the digit 0). However, as noted in [3], it does not admit a τ-w-NAF (i.e. Algorithm 5 does not terminate) for some inputs whenever $w \notin \{2, 3, 4, 5, 7, 8, 9, 10\}$.

Solinas forgoes the discussion about different possible choices for digit sets and simply suggests a digit set containing 0 and a representative of *minimal norm* from every prime residue class modulo τ^w. We refer to this set as MNR(w). In his construction (cf. [23, Section 5.1]) the author describes a division procedure in $\mathbb{Z}[\tau]$ and defines his digit set as the remainder upon division of the elements of $\mathcal{D}_w$ by τ^w. He introduces a rounding operation in $\mathbb{C}$, where complex numbers are rounded to the nearest element of $\mathbb{Z}[\tau]$. This is achieved by partitioning the complex plane in Voronoi Cells V_u, where

$$\begin{aligned} V := V_0 &= \{z \in \mathbb{C} \mid \forall y \in \mathbb{Z}[\tau]\colon \mathrm{N}(z) \leq \mathrm{N}(z+y)\}, \\ V_u &= V + u \text{ for } u \in \mathbb{Z}[\tau] \end{aligned}$$

and every $z \in \mathbb{C}$ is rounded to the center of its Voronoi cell[2]. Here $\mathrm{N}(z)$ refers to the norm, i.e. $\mathrm{N}(z) = z\bar{z}$. Note that $\mathrm{N}(a + b\tau) = a^2 + \mu ab + 2b^2$ and $\mathrm{N}(\tau) = 2$. The cell V is a hexagon, formed by 6 inequalities and all its corners lie on the circle formed by the set of all points of norm $\frac{4}{7}$.

This region V is of particular interest, because a scaled version $\tilde{V} := \tau^w V$ can be used to geometrically characterize MNR(w). Since $\tilde{V} = \{z \in \mathbb{C} \mid \forall y \in \mathbb{Z}[\tau]\colon \mathrm{N}(z) < \mathrm{N}(z + \tau^w y)\}$, it is evident that the elements in $\tilde{V} \cap \mathbb{Z}[\tau]$ are exactly those who have a smaller norm than any of the elements in the same residue class modulo τ^w. So the minimal norm representatives are made up of 0 and the elements $a + b\tau$ inside of $\tilde{V}$ with odd a.

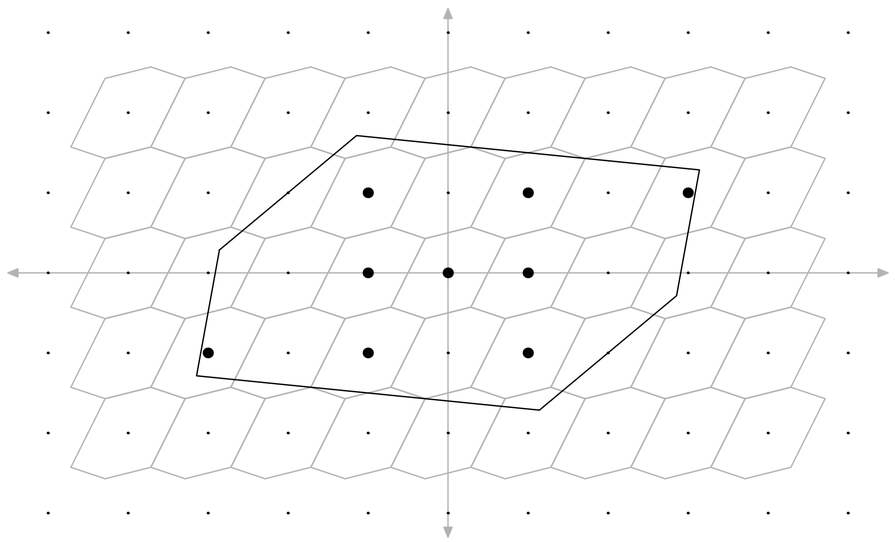

Figure 1: The set MNR(4) including the V_u-grid and the single cell $\tilde{V}$ ($\mu = -1$)

Figure 1 shows the set MNR(4) for $\mu = -1$ and the tiling of the complex plane into Voronoi regions V_u, where coordinate points (x, y) correspond to $x + y\tau \in \mathbb{C}$. The larger hexagon is the cell $\tilde{V}$ and it can be seen that the multiplication of V by τ^w causes a scaling and a rotation.

[2]To make this description completely unambiguous we would have to discuss what happens for points on the border between two Voronoi cells. For such a discussion we refer to [3]

2.2.3 Non-optimality of the τ-w-NAF

The question of optimality in case of the τ-w-NAF has already been answered positively for $w = 2$ (where the τ-w-NAF is just the ordinary τ-NAF and $\mathrm{MNR}(2) = \{0, \pm 1\}$) and $w = 3$ in [2]. As seen earlier, the base 2 w-NAF has lead to optimal expansions for all values of w. It would therefore be quite plausible that a similar result can be obtained for the τ-w-NAF. This is not the case as shown by Heuberger in [10]. For the case $w = 4$, $\mu = -1$ and $\mathcal{D} = \mathrm{MNR}(4)$, the author gives a simple example illustrating the failure of the w-NAF to produce an optimal expansion. Consider

$$\varepsilon = (1, 0, 0, 0, -1 - \tau, 0, 0, 0, 1 - \tau) \quad \text{and} \quad \tilde{\varepsilon} = (-3 - \tau, 0, 0, -1).$$

Here, ε is the unique τ-4-NAF of -9 and $\tilde{\varepsilon}$ is a different base τ-expansion of -9 with the same digit set, not satisfying the NAF-condition. However, ε has weight 3, while the weight of $\tilde{\varepsilon}$ is only 2.

This example alone voids all speculation for general optimality of the base τ-w-NAF, but in the main theorem of [10], the situation is further clarified. The problem is not that the τ-w-NAF is just not good enough, but rather that for $w \in \{4, 5, 6\}$, it is impossible to construct an online algorithm to give an optimal expansion. What is meant by that is that in order to even determine the least significant bit of an optimal expansion, one may have to read in the complete input, or more formally (cf. [10, Theorem 1]):

Theorem. *Let $w \in \{4, 5, 6\}$ and $\mathcal{D} = \mathrm{MNR}(w)$[3]. For every positive integer ℓ, there exist z_ℓ and $z'_\ell \in \mathbb{Z}[\tau]$, such that*

- $z_\ell \equiv z'_\ell \mod \tau^\ell$ *and*
- *minimal weight $\mathcal{D}$-expansions of z_ℓ and z'_ℓ differ in their least significant digit.*

2.3 Joint expansions

Joint expansions have been proposed by Solinas [24] in an effort to find a way to efficiently compute expressions of the form

$$m \cdot P + n \cdot Q, \tag{6}$$

where P and Q are points on an elliptic curve and m and n are integers as before. These computations represent a performance critical step in digital signature algorithms.

It had been known[4] earlier that, in order to compute (6), it is not necessary to compute mP and nQ individually (and subsequently add it). Instead, if one can precompute the point $R = P + Q$, it is possible to go through the (binary) expansions of m and n simultaneously in a double-and-add scheme. The doubling occurs in every step, just as in the 1-dimensional case and if m_j and n_j are the current digits in the binary expansions of

[3]Heuberger additionally allows two other types of digit sets

[4] as "Shamir's Trick", first introduced by Straus [25], see [12, Appendix A] for an algorithmic formulation

m and n respectively one then adds P if $(m_j, n_j) = (1, 0)$, Q if $(m_j, n_j) = (0, 1)$ and R if $(m_j, n_j) = (1, 1)$ in between. In analogy to the quest for low weights in our one-dimensional expansion, we would now like to see the case $(m_j, n_j) = (0, 0)$ as often as possible, since it corresponds to no addition at all. In the case of binary expansions of course, we can not influence the rate at which this happens, but as Solinas proceeds, one can allow signed expansions for m and n and since those are not unique, try to align them in a special way to get a low number of required additions when computing (6).

This motivates us to study *joint expansions* and we will define them in general terms before formally writing down the algorithm sketched above. In the following, we will denote the base of the expansion as $b \in \{2, \tau\}$. The inputs and digits will come from the set $\mathcal{R} := \mathbb{Z}[b]$ (which of course is just $\mathbb{Z}$ in case $b = 2$).

Definition 2.4. Let $b \in \{2, \tau\}$, $d \geq 1$ and $\mathcal{D} \subseteq \mathcal{R}$. A base b *joint $\mathcal{D}$-expansion* of a vector of integers $n \in \mathcal{R}^d$ is a sequence $\varepsilon = (\varepsilon_j)_{j \in \mathbb{N}_0} = (\ldots, \varepsilon_2, \varepsilon_1, \varepsilon_0)$ of digit vectors $\varepsilon_j \in \mathcal{D}^d$, where only finitely many digit vectors are nonzero[5] and

$$n = \operatorname{value}(\varepsilon) := \sum_{j \geq 0} \varepsilon_j b^j.$$

The number of nonzero[5] ε_j is called the *(joint) weight* of the expansion and written as $\operatorname{weight}(\varepsilon)$.

One way to look at joint expansions is seeing them as a matrix

$$\varepsilon = \left(\varepsilon_j^{(k)}\right)_{\substack{1 \leq k \leq d \\ 0 \leq j \leq \ell}} = \begin{pmatrix} \varepsilon_\ell^{(1)} & \cdots & \varepsilon_1^{(1)} & \varepsilon_0^{(1)} \\ \vdots & & \vdots & \vdots \\ \varepsilon_\ell^{(d)} & \cdots & \varepsilon_1^{(d)} & \varepsilon_0^{(d)} \end{pmatrix}$$

with sufficiently large ℓ and entries $\varepsilon_j^{(k)} \in \mathcal{D}$. This way, we can write

$$n = \begin{pmatrix} n^{(1)} \\ \vdots \\ n^{(d)} \end{pmatrix} = \begin{pmatrix} \varepsilon_\ell^{(1)} & \cdots & \varepsilon_1^{(1)} & \varepsilon_0^{(1)} \\ \vdots & & \vdots & \vdots \\ \varepsilon_\ell^{(d)} & \cdots & \varepsilon_1^{(d)} & \varepsilon_0^{(d)} \end{pmatrix} \cdot \begin{pmatrix} b^\ell \\ \vdots \\ b^1 \\ b^0 \end{pmatrix},$$

the row $\varepsilon^{(k)}$ of the matrix is an integer expansion of $n^{(k)}$ and the joint weight of ε is the number of nonzero columns.

Algorithm 6 is a generalization of the description above and works for general joint $\mathcal{D}$-expansions ε of $n = (n^{(1)}, \cdots, n^{(d)})^{\mathrm{T}}$ and curve points $P_1, \ldots, P_d$, to compute

$$\sum_{k=1}^{d} n^{(k)} \cdot P_k.$$

[5] In this context, "nonzero" means "not equal to the zero-vector"

Algorithm 6 Straus' algorithm

```
Input:  $\varepsilon = \left(\varepsilon_j^{(k)}\right)_{\substack{1\le k\le d\\ 0\le j\le \ell}}$ base $b$ joint $\mathcal{D}$-expansion of $n = (n^{(1)}, \cdots, n^{(d)})^{\mathrm{T}}$ and
  $P_1, \ldots, P_d$ elliptic curve points
Output:  $Q = \sum_{k=1}^{d} n^{(k)} \cdot P_k$
  for all $\eta \in \mathcal{D}^d$ do
    $R_\eta := \sum_{k=1}^{d} \eta^{(k)} \cdot P_k$
  end for
  $Q := R_{\varepsilon_\ell}$
  for $j = \ell - 1 \to 0$ do
    $Q := b \cdot Q$
    if $\varepsilon_j \neq 0$ then
      $Q := Q + R_{\varepsilon_j}$
    end if
  end for
  return  $Q$
```

The number of elliptic curve additions and hence the performance of this algorithm depends on the joint weight of the expansion ε. As in the one-dimensional case it has been the focus of a whole line of research to investigate expansions that minimize the weight for given parameters d, b and $\mathcal{D}$.

Obviously just generating minimal expansions of every $n^{(k)}$ individually is not a good solution. Consider the example $n = (45, 38)^{\mathrm{T}}$ and the two joint expansions ε and $\tilde{\varepsilon}$ of n given by

$$\varepsilon = \begin{pmatrix} 1 & 0 & -1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & -1 & 0 \end{pmatrix} \qquad \text{and} \qquad \tilde{\varepsilon} = \begin{pmatrix} 1 & 0 & -1 & 0 & 0 & -1 & -1 \\ 1 & 0 & -1 & -1 & 0 & -1 & 0 \end{pmatrix}.$$

The rows of ε are the unique NAF-representations of 45 and 38 respectively with minimal weights 4 and 3. The rows of $\tilde{\varepsilon}$ do neither satisfy the NAF-property, nor are they both minimal expansions ($\tilde{\varepsilon}_1$ is minimal, but $\tilde{\varepsilon}_2$ is not). However, when we turn our attention to the joint weights of the expansions, we have

$$5 = \text{weight}(\tilde{\varepsilon}) < \text{weight}(\varepsilon) = 7.$$

2.3.1 The joint sparse form and the simple joint sparse form

Solinas first investigated the most basic case, where $b = 2$, $d = 2$ and $\mathcal{D} = \{0, \pm 1\}$. His approach (cf. [24, Chapter 4]) is based on a generalization of the non-adjacent form and called the *joint sparse form* or JSF.

An base 2 joint expansion $\varepsilon = \left(\varepsilon_j^{(k)}\right)_{\substack{1\le k\le 2\\ 0\le j\le \ell}}$ with digits in $\mathcal{D} = \{0, \pm 1\}$ is in joint sparse form, if

(JSF-1) Of any three consecutive columns, one is the zero-column,

(JSF-2) Adjacent terms do not have opposing signs, i.e. $\varepsilon_{j+1}^{(k)}\varepsilon_j^{(k)} \neq -1$ $(k \in \{1,2\})$,

(JSF-3) If $\varepsilon_{j+1}^{(k)}\varepsilon_j^{(k)} \neq 0$, then $\varepsilon_{j+1}^{(3-k)} = \pm 1$ and $\varepsilon_j^{(3-k)} = 0$.

Based on these three syntactical conditions, he proves that every pair of integers has a unique JSF and that it minimizes the weight among all joint expansions of that pair with the same digit set. The average density of a JSF is given as $\frac{1}{2}$ and since it is at most one bit longer than the binary expansion of the larger of the two integers, one can say that using the JSF, (6) can be computed at the same cost as one simple scalar multiplication using the binary method (on an ordinary binary expansion of the same length).

Of course Solinas also includes an algorithm for the computation of the JSF, but we leave it out and instead present a slightly modified version of the JSF, which was described by Grabner et al. in [8], has a simpler definition and yields a less elaborate algorithm. The syntactical requirements for this *simple joint sparse form* (or SJSF) are

(SJSF-1) If $\left|\varepsilon_j^{(k)}\right| \neq \left|\varepsilon_j^{(3-k)}\right|$, then $\left|\varepsilon_{j+1}^{(k)}\right| = \left|\varepsilon_{j+1}^{(3-k+1)}\right|$ and

(SJSF-2) If $\left|\varepsilon_j^{(k)}\right| = \left|\varepsilon_j^{(3-k)}\right| = 1$, then $\varepsilon_{j+1}^{(k)} = \varepsilon_{j+1}^{(3-k+1)} = 0$.

As remarked by its inventors, the SJSF is also unique and very closely related to the JSF. It has the same (minimal) weight and in fact it even has the zero-columns in the same positions. It does however not generally satisfy (JSF-2). Algorithm 7 (cf. [8, Algorithm 1]) shows how to generate the simple joint sparse form of a pair $n = (n^{(1)}, n^{(1)})^{\mathrm{T}}$.

The basic idea behind the simple joint sparse form is that for odd integers x, we can freely choose the least significant bit x_0 of a signed binary encoding to be 1 or -1. When making the next step in the right-to-left encoding scheme, we are then looking at $(x-x_0)/2$. Through our choice of x_0, we can influence whether this value is even or odd. When looking at a pair $\binom{x}{y}$, we basically have 3 cases. If x and y are even, we immediately get a zero column in the encoding. If x and y are odd, we can pick x_0 and y_0 from $\{\pm 1\}$ in such a manner, that $(x - x_0)/2$ and $(y - y_0)/2$ are both even and we get a zero column in the next step. If wlog. x is odd and y is even, we can chose $x_0 \in \{\pm 1\}$ in order to guarantee $(x - x_0)/2 \equiv (y - y_0)/2 \mod 2$, bringing us to one of the first two cases in the next step and therefore getting a zero column within one or two steps.

2.3.2 Higher dimensions

One advantage of the SJSF over the JSF is that the authors of [8] were able to seamlessly generalize it to higher dimensions $d > 2$. Solinas already classified this as problem of interest, but admits that his JSF does not have an obvious generalization. The ideas of the SJSF however, can be applied to the case were $d > 2$ as follows:

Algorithm 7 Simple Joint Sparse Form

```
Input:   $n = \binom{n^{(1)}}{n^{(2)}}$
Output:   The simple joint sparse form $\varepsilon = \begin{pmatrix} \varepsilon_\ell^{(1)} \cdots \varepsilon_0^{(1)} \\ \varepsilon_\ell^{(2)} \cdots \varepsilon_0^{(2)} \end{pmatrix}$ of $n$
  $j := 0$
  while $n^{(1)} \neq 0$ or $n^{(2)} \neq 0$ do
    $\varepsilon_j := n \mod 2$
    if $\begin{pmatrix} \varepsilon_j^{(1)} \\ \varepsilon_j^{(2)} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ then
      $\varepsilon_j := \varepsilon_j - ((n - \varepsilon_j) \mod 4)$
    else if $\varepsilon_j^{(1)} \neq \varepsilon_j^{(2)}$ then
      if $(n^{(1)} - \varepsilon_j^{(1)})/2 \not\equiv (n^{(2)} - \varepsilon_j^{(2)})/2 \mod 2$ then
        $\varepsilon_j := -\varepsilon_j$
      end if
    end if
    $n := (n - \varepsilon_j)/2$
    $j := j + 1$
  end while
```

One tries to get to the case, where after j steps, the inputs $(n^{(1)}, \dots, n^{(d)})^{\mathrm{T}}$ are all either $\equiv 0 \mod 4$, or odd. By then again picking next digits $\varepsilon_j^{(k)} \in \{\pm 1\}$ for the columns k with odd $n^{(k)}$, such that $(n^{(k)} - \varepsilon_j^{(k)})/2$ is even, we get a zero column in the net step.

The algorithm for this *d-dimensional joint sparse form* can be found in [8, Algorithm 2]. While computing the expansion ε from right to left, at every step j it keeps track of the set of line indices of column j for which $\varepsilon_j^{(k)} \neq 0$.

$$A_j(\varepsilon) := \{1 \leq k \leq d \mid \varepsilon_j^{(k)} \neq 0\}$$

In step j, the set $A_{j+1}(\varepsilon)$ is initialized with the indices of the lines where the remaining input is odd. If $A_{j+1}(\varepsilon) \subseteq A_j(\varepsilon)$, we simply set $\varepsilon_j^{(k)} := -\varepsilon_j^{(k)}$ for all $k \in A_{j+1}(\varepsilon)$ and after the corresponding carries are considered, $A_{j+1}(\varepsilon)$ becomes the empty set. Conversely, if $A_{j+1}(\varepsilon) \setminus A_j(\varepsilon) \neq \emptyset$, set $\varepsilon_j^{(k)} := -\varepsilon_j^{(k)}$ for all $k \in A_j(\varepsilon) \setminus A_{j+1}(\varepsilon)$. Then we set $A_{j+1}(\varepsilon) := A_j(\varepsilon) \cup A_{j+1}(\varepsilon)$. Summing up, the algorithm works by generating the expansion ε, such that it satisfies

$$A_{j+1}(\varepsilon) \supsetneq A_j(\varepsilon) \text{ or } A_{j+1}(\varepsilon) = \emptyset \tag{7}$$

for all $0 \leq j \leq \ell$. Since any chain of the form $A_{j+r}(\varepsilon) \supsetneq \cdots \supsetneq A_j(\varepsilon) \neq \emptyset$ can at most have d elements (i.e. $r \leq d$), it follows that among $d+1$ consecutive digit columns, at least one is a zero-column. This is the logical generalization of the condition (JSF-1) and as shown in [8], for any $n = (n^{(1)}, \cdots, n^{(d)})^{\mathrm{T}} \in \mathbb{Z}^d$, a unique expansion ε of n satisfying (7) always exists and is minimal among all joint expansions of n.

2.3.3 Other generalizations

The d-dimensional SJSF addresses the limitations of the (S)JSF in terms of the value of d. The two other restrictions that remains are the limitation to the digit set $\mathcal{D} = \{0, \pm 1\}$ and the sole use of base 2.

In [12], Heuberger and Muir consider base 2 joint expansions of arbitrary dimension $d \geq 1$ using the digit set $\mathcal{D}_{\ell,u} = \{a \in \mathbb{Z} \mid \ell \leq a \leq u\}$ for integers $\ell \leq 0$ and $u \geq 1$. They describe an algorithm that computes minimal expansions and give the expectation of this minimal joint weight.

To also introduce joint expansions when working in base τ, Ciet et al. generalized Solinas' joint sparse form to a *τ-JSF* (cf. [6]).

A base τ joint integer expansion

$$\varepsilon = \begin{pmatrix} \varepsilon_j^{(1)} \\ \varepsilon_j^{(2)} \end{pmatrix}_{0 \leq j \leq \ell}$$

of $n \in \mathcal{R}^2$ is called a τ-JSF, if

(τ-JSF-1) Among three consecutive columns, at least one is a zero column.

(τ-JSF-2) For all $j \geq 0$ and $k \in \{1, 2\}$, we have $\varepsilon_j^{(k)} \varepsilon_{j+1}^{(k)} \neq \mu$.

(τ-JSF-3) If $\varepsilon_{j+1}^{(k)} \varepsilon_j^{(k)} \neq 0$ then $\left|\varepsilon_{j+1}^{(3-k)}\right| = 1$ and $\varepsilon_j^{(3-k)} = 0$.

Existence and uniqueness of the τ-JSF is established in [6, Theorem 4]. The authors point out[6] that minimality does not carry over from the JSF, but claim that for large inputs, the weight of the τ-JSF only deviates from the optimal weight by a small constant. As in the one-dimensional case, Heuberger shows in [10, Section 7] that an optimal expansion can not generally be computed by a right-to-left online algorithm.

3 Signed binary digit expansions

In this chapter we want to turn our attention to base 2 signed binary expansions, which according to Definition 2.1 are base 2 digit expansion with digit set $\mathcal{D} = \{0, \pm 1\}$. With the non-adjacent form, we already studied an expansion that has many desirable properties for our applications. These are: (cf. [24])

- The NAF exists for every integer.
- The NAF is unique.
- The algorithm to compute the NAF of an integer is simple and efficient.
- The NAF of a positive integer is at most one bit longer than its binary expansion.

[6]via a simple example

- The NAF of an integer has minimal Hamming weight among all its other signed binary expansions.
- The expected density of a NAF is $\frac{1}{3}$.

This warrants the question if there is even a need to consider any other way of representing integers as signed binary expansions than the NAF. To answer this, one can consider any possible weaknesses and find expansions that address these while not compromising on the properties above.

One such slight disadvantage of the NAF (and all its generalizations) is the way it is computed from right to left. This is problematic, because in the double-and-add algorithm used for scalar multiplication, we need the digits from left-to-right, so when using an expansion that can only be computed right-to-left, we first have to compute and store the whole expansion and then perform the multiplication. In the case of the basic NAF, the remedy for this is a modification of the double-and-add method that, instead of the intermediate result, doubles the point P in every step and works from the right to the left.

Algorithm 8 Right-to-left Binary Method using the NAF

```
Input: Integer $n$, curve point $P$
Output: Point $Q$ such that $Q = nP$
  $Q := 0$
  $c := n$
  while $c \neq 0$ do
    if $c \equiv 1 \mod 2$ then
      $\varepsilon := 2 - c \mod 4$
      $c := c - \varepsilon$
      if $\varepsilon = 1$ then
        $Q := Q + P$
      else
        $Q := Q - P$
      end if
    end if
    $c := \frac{c}{2}$
    $P := 2P$
  end while
```

Algorithm 8 illustrates this. In order to fully benefit from this situation, where the expansion can be computed in the same direction as the scalar multiplication, one has to combine the two algorithms to one *online*-algorithm and never even store the digit expansion.

However, this approach is not applicable for any of the methods beyond the simple NAF with $\mathcal{D} = \{0, \pm 1\}$. As soon as we use larger digit sets, or want to consider joint expansions, the multiplication algorithm involves the usage of precomputations. So instead of a possible addition or subtraction of one point P, we are facing addition or subtraction of a set of points (see for example Algorithm 6) and to get a right-to-left algorithm we would have to double all these points in every step, which is not efficient.

A second albeit closely related drawback of the NAF is that since it is generally computed from right to left, one can not determine an arbitrary digit in the NAF without first computing the complete expansion up to this digit. As we will see, this is a limitation that other expansions do not share.

In the following we will look at two signed binary expansions that where designed to address some of these issues. One is the *alternating greedy* expansion presented in [9] and the other one is based on building digit-wise complements in the binary expansion. As we will see, it is a misconception to think that any of the two are better suited for use in scalar multiplication than the NAF. At least the former of the two does have some very interesting applications as a “helper expansion”.

3.1 Alternating greedy expansion

A signed binary expansion $(\varepsilon_j)_{j \in \mathbb{N}_0}$ is called an *alternating greedy expansion*, if it fulfills the following conditions:

(AG-1) If $\varepsilon_j = \varepsilon_r \neq 0$ for some $j < r$, then there is a k with $j < k < r$, such that $\varepsilon_j = -\varepsilon_k = \varepsilon_r$.

(AG-2) The first and the last nonzero digit have opposing sign, ie. for $a := \max\{j \mid \varepsilon_j \neq 0\}$ and $b := \min\{j \mid \varepsilon_j \neq 0\}$, we have $\varepsilon_a = -\varepsilon_b$.

Existence of the alternating greedy expansion has been proved in [9] and uniqueness is shown in [11]. We can hence speak of *the* alternating greedy expansion of an integer. In fact it is easy to see that the alternating greedy expansion can be computed from the binary expansion of an integer. Since we must also consider negative integers, we have to note that the binary expansion of a negative integer n is the same as the binary expansion of $-n$, except that all the nonzero digits are negative. This expansion consisting either entirely of digits 0 and 1 (positive n) or 0 and -1 is sometimes referred to as *unisigned expansion* and is of course unique.

So if n is given in its unisigned expansion, the alternating greedy expansion of n can be computed by bitwise subtracting n from $2n$. More formally, if $\eta = (\eta_{\ell-1}, \dots, \eta_0)$ is the unisigned expansion of n, then the alternating greedy expansion is given by $\varepsilon = (\varepsilon_\ell, \dots, \varepsilon_0)$, where

$$\begin{aligned} &\varepsilon_\ell = \eta_{\ell-1}, \\ &\varepsilon_0 = -\eta_0 \text{ and} \\ &\varepsilon_j = \eta_{j-1} - \eta_j \text{ for } 0 < j < \ell. \end{aligned}$$

This expansion ε satisfies (AG-1), because if $\varepsilon_j = \varepsilon_r \neq 0$ for some $j < r$, we get that $\eta_j = \eta_r$, $\eta_{j-1} = \eta_{r-1}$ and $\eta_{r-1} \neq \eta_j$. So there is a k with $j < k \leq r-1$ such that $\eta_k = \eta_{r-1}$ and $\eta_{k-1} = \eta_j$. It follows that $\varepsilon_k = -\varepsilon_j = -\varepsilon_r$.

It satisfies (AG-2), because setting $a = \max\{j \mid \varepsilon_j \neq 0\}$ and $b = \min\{j \mid \varepsilon_j \neq 0\}$, we get $a = 1 + \max\{j \mid \eta_j \neq 0\}$ and $b = \max\{j \mid \eta_j \neq 0\}$. Therefore $\varepsilon_a = \eta_{a-1}$ and $\varepsilon_b = -\eta_b$ and because η is a unisigned expansion it follows that $\varepsilon_b = -\varepsilon_a$.

Algorithm 9 shows how to compute the alternating greedy expansion from left to right.

Algorithm 9 Computing the alternating greedy expansion from left to right

```
Input:   $\eta = (\eta_{\ell-1}, \dots, \eta_0)$ unisigned expansion of $n \in \mathbb{Z}$
Output:  the alternating greedy expansion $\varepsilon = (\varepsilon_\ell, \dots, \varepsilon_0)$ of $n$
  $\eta_\ell := 0$
  $\eta_{-1} := 0$
  for $j = \ell \to 0$ do
    $\varepsilon_j = \eta_{j-1} - \eta_j$
  end for
```

Note that if the input is already given in its unisigned binary expansion, the computation of the alternating greedy expansion can be done in parallel and very efficiently implemented in hardware. This is in strong contrast to the previously studied expansions, where we have to compute the individual digits separately from right to left. The key property of the alternating greedy expansion, however is that it can be used as a helper expansion to facilitate the computation of minimal expansions from left to right. This was actually the motivation to define this expansion as explained in the last chapter of [9]. There, the authors describe an algorithm that computes minimal base 2 joint expansion of pairs of integers in the digit set $\mathcal{D} = \{0, \pm 1\}$. The main observation is that if the input $\begin{pmatrix} n^{(1)} \\ n^{(2)} \end{pmatrix} \in \mathbb{Z}^2$ is given as a joint expansion $\begin{pmatrix} \varepsilon^{(1)}_{\ell-1} & \cdots & \varepsilon^{(1)}_0 \\ \varepsilon^{(2)}_{\ell-1} & \cdots & \varepsilon^{(2)}_0 \end{pmatrix} \in \mathbb{Z}^2$, where the rows $(\varepsilon^{(1)}_j)_{0 \leq j < \ell}$ and $(\varepsilon^{(2)}_j)_{0 \leq j < \ell}$ are both alternating greedy expansions of $n^{(1)}$ and $n^{(2)}$ respectively [7], one can compute a minimal joint expansion from left to right by looking at blocks of at most three consecutive columns at a time. These blocks can always be manipulated in order to produce one zero-column, without creating any carries that leave the block. A complete list of possible blocks and the necessary manipulations is given in [9, Table 6].

This was generalized for the d-dimensional joint expansions of minimal weight in [11]. There the input is also converted to a joint alternating greedy expansion and then the simple joint sparse form of blocks of columns is computed from left to right. The concatenation of these blocks in SJSF is then (though not a SJSF) an expansion of minimal joint weight.

3.2 Complementary recoding

In [4], the authors introduce a method for elliptic curve scalar multiplication that works based on building complements.

[7] We call a joint expansion consisting of rows of alternating greedy expansions a *joint alternating greedy expansion.*

Given a positive integer n in its binary representation $(\eta_{\ell-1}, \ldots, \eta_0)$, performing a complementary recoding of n refers to the representation

$$n = 2^\ell - \sum_{j=0}^{\ell-1} \bar{\eta}_j 2^j - 1, \tag{8}$$

where $\bar{\eta}_j := 1 - \eta_j$ is the binary complement of η_j (hence the name of this approach).

The correctness of Equation (8) follows from the fact that

$$\sum_{j=0}^{\ell-1} \bar{\eta}_j 2^j = \sum_{j=0}^{\ell-1} (1 - \eta_j) 2^j = \sum_{j=0}^{\ell-1} 2^j - \sum_{j=0}^{\ell-1} \eta_j 2^j = 2^\ell - 1 - n.$$

The corresponding integer expansion of n is somewhat loosely described in [4]. It is clear that the term $2^\ell - \sum_{j=0}^{\ell-1} \bar{\eta}_j 2^j$ can be represented as $\tilde{\varepsilon} = (1, -\bar{\eta}_{\ell-1}, \ldots, -\bar{\eta}_0)$ and that this can be done very efficiently. In the examples, the authors suggest that their integer expansion ε of n also includes the subtraction of 1 at the end of (8). This is straightforward if $\bar{\eta}_0 = 0$, i.e. if n is odd, since then $\varepsilon = (1, \bar{\eta}_{\ell-1}, \ldots, \bar{\eta}_1, -1)$ is an signed binary expansion of n. However, if n is even and $\bar{\eta}_0 = 0$ subsequently odd, a carry occurs when subtracting 1 from the representation $\tilde{\varepsilon}$. This is undesirable since it negates the strength of this method which is that it can otherwise be computed by bit-shifts and complement-building only.

A possible remedy for this is to represent n as $\tilde{\varepsilon} - 1$ without changing $\tilde{\varepsilon}$, and applying this to scalar multiplication, by computing nP as $(n+1)P - P$, whenever n is even. We suspect that this is what the authors actually meant and a look at the origin of their method (cf. [5]) also suggests this to be the case. From our point of view this workaround is equivalent to allowing -2 as a digit for the least significant bit of the resulting expansion ε and augmenting the signed binary method to accommodate for that by subtracting P twice.

Along these lines, we will from now on refer to the output of Algorithm 10 given the input n as the *complementary recoded form* of n.

Algorithm 10 Computing the complementary recoded form

```
Input:  Positive integer $n$ in its binary representation $\eta = (\eta_{\ell-1}, \ldots, \eta_0)$
Output:  $\varepsilon = (\varepsilon_\ell, \ldots, \varepsilon_0)$, the CRF of $n$
  $\eta_\ell := 1$
  for $j = \ell - 1 \to 0$ do
    $\varepsilon_j = -\bar{\eta}_j$
  end for
  $\eta_0 := \eta_0 - 1$
```

Note that the loop in this algorithm would practically be implemented by fast low-level machine operations. With xor being the familiar bit-wise exclusive or and $\ominus$ representing digit-wise subtraction, we could write it as $2^l \ominus ((2^\ell - 1) \operatorname{xor} n)$.

Since we are always interested in the contribution our digit expansions can make to fast elliptic scalar multiplication, the efficiency with which the expansions can be computed is only one part of our considerations. The far more important part is the weight of such expansions, as explained in Chapter 2. In this respect, the CRF does not compare favorably with optimal expansions such as the NAF. It is obvious that the expected hamming weight of the binary expansion of any given integer n is exactly the same as the expected hamming weight of the complement of said expansion.

What the authors of [4] neglect to mention, is that for their approach to be an improvement to even the ordinary binary expansion, the complementary recoding is only performed for ℓ-bit integers whose binary expansion has Hamming weight larger than $\frac{\ell}{2}$. Even if we consider this modification, the expected Hamming weight does not decrease dramatically[8].

As a final remark, we give an example comparing different signed binary expansion of $n = 271$. We get

$$\begin{aligned} n &= (0,1,0,0,0,0,1,1,1,1) && \text{(binary expansion, weight 5)} \\ &= (1,0,-1,-1,-1,-1,0,0,0,-1) && \text{(CRF, weight 6)} \\ &= (0,1,0,0,0,1,0,0,0,-1) && \text{(NAF, weight 3)} \end{aligned}$$

3.3 Empirical performance comparison

In an effort to try and reproduce the results presented in [4], we implemented a scalar multiplication test-suite in C++, using the same cryptographic library that was used there[9].

Table 1 shows the average timings for a series of elliptic curve scalar multiplications on five different curves. These are some of the curves standardized by NIST[10] in [7] and widely used in cryptographic protocols today. As inputs we use positive integers n with $n < 2^p$, where n is the size of the underlying field.

We compare the performances of 6 different implementations of integer expansions. They are:

BIN: The standard binary expansion; here no recoding whatsoever is performed, but only the standard binary method is used.

AG: The alternating greedy expansion,

NAF: The non-adjacent form,

CRF: The complementary recoded form, computed digit-wise from left to right,

CRF1: Here we only apply complementary recoding if the binary expansion of length ℓ has weight larger than $\ell/2$.

[8]and definitely not to $\frac{l}{4}$ as claimed in [5]

[9]A C/C++ library for number theory and cryptography called MIRACL, see http://www.shamus.ie

[10]The National Institute of Standards and Technology, a regulatory agency of the United States' government

CRF2: In this implementation of the CRF, we tried to capitalize on its suitability for parallel computation as much as possible.

The data produced by our tests nicely exemplifies several theoretical results. If we take a look at the average weights of our expansions, we see that the non-adjacent form always prevails. From our theoretical results we know of course that it is a minimal expansion for any given integer n, so this also shows when averaging over many inputs. All the other expansions fail to show any significant weight reduction over even just the ordinary binary expansion. For the alternating greedy this is not surprising since its purpose is not that of being a low weight expansion itself, but to assist in the left-to-right computation of minimal expansions. For the complementary recoded form, we see that an unconditional recoding for all inputs (corresponding to the label CRF in Table 1) even increases the average weight when compared to that of the binary expansion. The improved implementation where recoding is only done if more than half of the digits of the binary expansion are non-zero (label CRF1) has a slightly improved average weight, but is still far off from that of the NAF.

In an effort to bring out the strengths of the complementary recoding approach, we implemented a version (CR2), where the recoding operations where largely done in parallel. We show the overall computation time (t_{total}) and its composition as the sum of recoding time (t_{recode}) and time used for the actual scalar multiplication (t_{mult}). While the implementation CR2 does achieve a considerable speedup by a factor between 10 and 30 when compared to the NAF, it becomes apparent that the time needed for the multiplication procedure far outweighs any gains made in the recoding step.

4 Automated minimal weight analysis

In this chapter, our goal is to compute values for the minimal average Hamming weight of integers achievable by any expansion, given a fixed digit set $\mathcal{D}$. So far we have always seen a more constructive approach in which one typically chooses a digit set, gives a method specifying how to encode the input and then analyzes how these choices affect average weight (and possibly length) of the resulting expansions. The encoding methods are typically designed to be efficiently carried out in an on-line-algorithm.

Now we focus on the minimal average weight that can be achieved, given a certain digit set, without imposing any syntactic restrictions or encoding rules on the result. Our goal is not to devise an algorithm that actually delivers these expansions in a computationally competitive fashion, but to determine what the optimal bound for such an algorithms is.

This has been done in [26], although restricted to a limited type of digit set and only for expansions with base 2. The basic steps are as follows:

1. We describe an algorithm that computes minimal weight $\mathcal{D}$-expansions.
2. We observe that even for arbitrary length inputs, our algorithm only goes through a finite number of states. Based on this observation we model it as a finite state

NIST-$\mathbb{F}(2^m)$-163				
Representation	Average weight	t_{recode}	t_{mult}	t_{total}
BIN	81.18	0	1.395	1.395
AG	81.36	0.0046	1.415	1.421
NAF	54.20	0.0038	1.180	1.184
CRF	87.86	0.0032	1.495	1.497
CRF1	78.74	0.0010	1.380	1.382
CRF2	87.86	0.0004	1.495	1.495
NIST-$\mathbb{F}(2^m)$-283				
Representation	Average weight	t_{recode}	t_{mult}	t_{total}
BIN	140.85	0	4.26	4.260
AG	141.79	0.0080	4.27	4.278
NAF	94.51	0.0064	3.58	3.586
CRF	148.13	0.0056	4.39	4.395
CRF1	137.40	0.0022	4.22	4.222
CRF2	148.13	0.0002	4.39	4.390
NIST-Kob-163				
Representation	Average weight	t_{recode}	t_{mult}	t_{total}
BIN	81.01	0	1.43	1.431
AG	81.34	0.0046	1.44	1.447
NAF	54.39	0.0038	1.21	1.214
CRF	88.01	0.0032	1.53	1.535
CRF1	78.95	0.0010	1.42	1.422
CRF2	88.01	0.0004	1.52	1.520
NIST-Kob-283				
Representation	Average weight	t_{recode}	t_{mult}	t_{total}
BIN	140.81	0	4.12	4.122
AG	140.76	0.0080	4.12	4.131
NAF	93.84	0.0064	3.44	3.447
CRF	148.16	0.0054	4.26	4.271
CRF1	137.03	0.0018	4.05	4.053
CRF2	148.16	0.0004	4.26	4.260
NIST-$\mathbb{F}(p)$-256				
Representation	Average weight	t_{recode}	t_{mult}	t_{total}
BIN	127.91	0	3.83	3.837
AG	128.27	0.0072	3.87	3.882
NAF	85.95	0.0060	3.37	3.378
CRF	129.06	0.0048	3.95	3.967
CRF1	122.10	0.0024	3.83	3.835
CRF2	129.06	0.0002	3.94	3.941

Table 1: Elliptic curve multiplication timing results for different integer expansions

machine, where each state transition corresponds to reading a new input digit. With each such transition we also associate a change of the overall Hamming weight of the result.

3. Using Markov chain terminology, we can then determine the asymptotic average weight change when reading a randomly distributed new digit. This is the expected density of the expansion, which directly gives us the minimum average joint Hamming weight.

We will use this idea, extend it to integer expansions with base τ and a more general class of digit sets. For base 2, the digit set under examination will be of the form $\mathcal{D} = \{0, \pm 1, \ldots, \pm(2h+1)\}$ (for some $h \geq 0$) whereas for base τ, we will use the minimal norm representatives introduced Chapter 2 (cf. [23]) and can then compare our results to the performance of expansions studied there. Depending on the base chosen, the domain for the input will either be $\mathbb{Z}$ or $\mathbb{Z}[\tau]$. We will always denote it by $\mathcal{R}$.

Note that we will in fact be studying *joint expansions* (c.f. Section 2.3), where the input is actually d-dimensional and what we will call *digit*, will in fact be a d-dimensional vector for some $d > 0$. However, due to the way our algorithm works, the explicit value of d is not a big consideration for the construction and in order to understand it, one only needs to understand the case $d = 1$. Our notation will largely avoid stressing the dimension of the problem. In fact we will be calling our digit set $\mathcal{D} := \mathcal{D}_0^d$, where $\mathcal{D}_0$ in an ordinary one-dimensional digit set as discussed before. Properties like *being divisible by* τ, or being zero will be intended component-wise.

4.1 Minimal weight expansion: A dynamic programming approach

Definition 4.1. Let $b \in \{2, \tau\}$ be the base, $d \in \mathbb{N} \setminus \{0\}$ and $\mathcal{D}$ a suitable digit set. A *joint $\mathcal{D}$-expansion* of $z \in \mathcal{R}^d$ is a sequence $\eta = (\ldots, \eta_2, \eta_1, \eta_0)$ of elements in $\mathcal{D}$ such that

1. only finitely many η_j are nonzero and
2. $\operatorname{value}(\eta) := \sum_{j \geq 0} \eta_j b^j = z$, so η is an expansion of z.

We call $n = \max\{j \in \mathbb{N}_0 \mid \eta_j \neq 0\}$ the *length* of the expansion and write $n = \operatorname{length}(\eta)$. The number of nonzero η_j is called the *(joint) weight* of the expansion and we write $\operatorname{weight}(\eta)$. A joint $\mathcal{D}$-expansion of z with minimal weight among all joint $\mathcal{D}$-expansions of z is called a *minimal weight joint $\mathcal{D}$-expansion* of z and we refer to its weight as $\operatorname{mw}_{\mathcal{D}}(z)$.

Definition 4.2. For $\mathcal{D}$, d as above and $n \in \mathbb{N}$, let

$$\mathcal{R}_n = \{z \in \mathcal{R}^d \mid \text{the unique } \{0,1\}\text{-expansion of } z \text{ has length } \leq n\}.$$

Then we define the *minimal average joint weight* of all $z \in \mathcal{R}_n$ as

$$\operatorname{majw}_{\mathcal{D}}(n) = \frac{1}{2^n} \sum_{z \in \mathcal{R}_n} \operatorname{mw}_{\mathcal{D}}(z)$$

and the *asymptotic minimal joint density* as

$$\mathrm{amjd}_{\mathcal{D}} = \lim_{n\to\infty} \frac{1}{n} \mathrm{majw}_{\mathcal{D}}(n).$$

Now we will discuss the algorithm that gives a minimal weight joint $\mathcal{D}$-expansion. We assume that the input has already been encoded to base b using digits from what we will call the *input digit set* $\mathcal{D}_{\text{in}} = \{0,1\}^d$. In the binary case this is the standard binary encoding of integers. With base τ, we can also easily encode every $z \in \mathbb{Z}$ this way by just looking at the last digit $\bmod \tau$. Another way to look at it, is that our input always comes from $\mathbb{Z}[\tau]$ to begin with.

As noted in [10], it is in principle easy to compute optimal expansions of $z \in \mathcal{R}$ recursively. If b divides the input, the least significant digit is always zero. If that is not the case, one simply picks $d \in \mathcal{D} \setminus \{0\}$, such that $\frac{z-d}{b}$ has minimal joint weight (which is to be computed recursively) among all possible choices for d. This is basically how our algorithm works, except that we avoid the recursion and introduce a dynamic programming scheme processing the input from left to right.

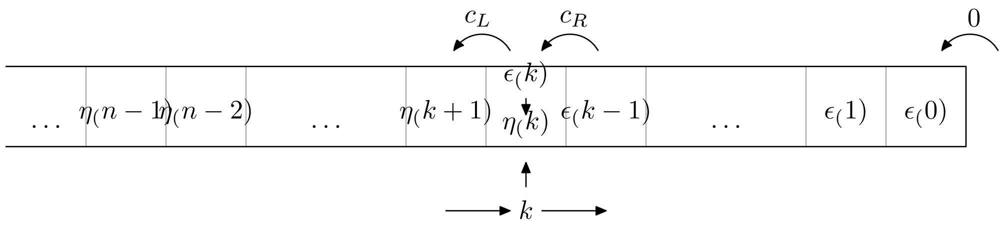

Figure 2: Scheme for minimal weight conversion

When at step k, we read the input digit $\varepsilon_k \in \mathcal{D}_{\text{in}}$ and recode it as some $\eta_k \in \mathcal{D}$. This recoding triggers a carry c_L, which we pass on to the left, just as we receive a carry of c_R from the right (see Figure 2). In order not to change the value of the expansion, c_L and η_k must fulfill

$$b \cdot c_L + \eta_k = \varepsilon_k + c_R. \tag{9}$$

4.1.1 The carry set

The first question that arises is, what values we can expect for the carries c_L and c_R. We say that a carry c *occurs* while recoding a $\mathcal{D}_{\text{in}}$-expansion $\varepsilon = (\dots, \varepsilon_1, \varepsilon_0)$ to a $\mathcal{D}$-expansion $\eta = (\dots, \eta_1, \eta_0)$ if for some $\ell \geq 0$,

$$c = \frac{1}{b^\ell} \sum_{j=0}^{\ell-1} (\varepsilon_j - \eta_j) b^j$$

and define the set of all possible carries below.

Definition 4.3. Let $\mathcal{D}$ be a digit set as before, then we define the *associated carry set* $\mathcal{C}$ as

$$\mathcal{C} = \left\{ \frac{1}{b^\ell} \sum_{j=0}^{\ell-1} (\varepsilon_j - \eta_j) b^j \;\middle|\; \varepsilon_j \in \mathcal{D}_{\text{in}},\, \eta_j \in \mathcal{D},\ \text{value}(\dots, \varepsilon_1, \varepsilon_0) = \text{value}(\dots, \eta_1, \eta_0),\, \ell \geq 0 \right\}.$$

Proposition 4.4. *Setting $M = max\{||\varepsilon|| + ||\eta|| \mid \varepsilon \in \mathcal{D}_{in},\, \eta \in \mathcal{D}\}$, we get that*

$$||c|| < \begin{cases} M & (\textit{if } b = 2) \\ M(1+\sqrt{2}) & (\textit{if } b = \tau) \end{cases}$$

holds for all $c \in \mathcal{C}$.

Proof. Let $z = \sum_{j=0}^{l-1} \varepsilon_j \cdot b^j \in \mathcal{R}$ with $\varepsilon_j \in \mathcal{D}_{\text{in}}$ and $z = c \cdot b^\ell + \sum_{j=0}^{\ell-1} \eta_j \cdot b^j$ an encoding with $\eta_j \in \mathcal{D}$ and $c \in \mathcal{C}$. We have

$$\begin{aligned} \sum_{j=0}^{\ell-1} \varepsilon_j \cdot b^j &= c \cdot b^\ell + \sum_{j=0}^{\ell-1} \eta_j \cdot b^j, \\ c \cdot b^\ell &= \sum_{j=0}^{\ell-1} (\varepsilon_j - \eta_j) \cdot b^j, \\ c &= \sum_{j=0}^{\ell-1} (\varepsilon_j - \eta_j) \cdot b^{j-\ell}. \end{aligned}$$

Now, using $||\varepsilon_j - \eta_j|| \leq M$, we get

$$\begin{aligned} ||c|| \leq M \sum_{j=0}^{\ell-1} |b|^{j-\ell} &= M \sum_{j=1}^{\ell} \left(\frac{1}{|b|} \right)^j \\ &< M \frac{1}{|b|} \frac{1}{1 - \frac{1}{|b|}} = \begin{cases} M & (\text{if } b = 2), \\ M(1+\sqrt{2}) & (\text{if } b = \tau). \end{cases} \end{aligned}$$

□

Proposition 4.5. *Let $c \in \mathcal{R}$. Then we have $c \in \mathcal{C}$ if and only if there are $\ell \in \mathbb{N}_0$, $\varepsilon_{\ell-1}, \dots, \varepsilon_0 \in \mathcal{D}_{in}$, $\eta_{\ell-1}, \dots, \eta_0 \in \mathcal{D}$ and $c_0, c_1, \dots, c_\ell \in \mathcal{R}$ with $c_0 = 0$, $c_\ell = c$ and*

$$c_{i+1} = \frac{c_i + \varepsilon_i - \eta_i}{b} \tag{10}$$

for all $0 \leq i \leq \ell$. In this case, all c_i are elements of $\mathcal{C}$.

Proof. If $c \in \mathcal{C}$, the definition of $\mathcal{C}$ gives us $\ell \in \mathbb{N}_0$, $(\varepsilon_j)_{0\le j\le \ell-1}$ and $(\eta_j)_{0\le j\le \ell-1}$ with $c = \frac{1}{b^\ell}\sum_{j=0}^{\ell-1}(\varepsilon_j - \eta_j)b^j$. We show that the claim holds using $c_i := \frac{1}{b^i}\sum_{j=0}^{i-1}(\varepsilon_j - \eta_j)b^j$. Obviously $c_0 = 0$, $c_\ell = c$ and all $c_i \in \mathcal{C}$, so we only need to show (10):

$$\begin{aligned} c_{i+1} = \frac{1}{b^{i+1}}\sum_{j=0}^{i}(\varepsilon_j - \eta_j)b^j &= \frac{1}{b^{i+1}}\left(\sum_{j=0}^{i-1}(\varepsilon_j - \eta_j)b^j + (\varepsilon_i - \eta_i)b^i\right) \\ &= \frac{\frac{1}{b^i}\sum_{j=0}^{i-1}(\varepsilon_j - \eta_j)b^j + (\varepsilon_i - \eta_i)}{b} \\ &= \frac{c_i + \varepsilon_i - \eta_i}{b} \end{aligned}$$

Conversely, again using $c_i = \frac{1}{b^i}\sum_{j=0}^{i-1}(\varepsilon_j - \eta_j)b^j$, it follows immediately that $c = c_\ell \in \mathcal{C}$. □

Proposition 4.5 implies $\mathcal{C}$ can be computed by starting with $\mathcal{C} = \{0\}$ and repeatedly applying Equation (10) to the carries that have already been found. The only restriction is that carries must be in $\mathcal{R}$, i.e. b must divide $c_i + \varepsilon_i - \eta_i$, which is equivalent to saying that $\ldots\eta_1, \eta_0$ is a valid recoding of $(\ldots\varepsilon_1, \varepsilon_0)$. The complete process is executed by Algorithm 11. The Algorithm is correct because of the observation above and it terminates because of Proposition 4.4.

Algorithm 11 algCarryset: Generating the carry set $\mathcal{C}$ for a given $\mathcal{D}$

Input: The digit set $\mathcal{D}$ and the base b
Output: The associated carry set $\mathcal{C}$

```
$\mathcal{C} := \{0\}$
$N := \mathcal{C}$ {New carries yet to be explored}
while $N \neq \{\}$ do
  $c_R$ :=POP($N$)
  for all $\varepsilon_k \in \mathcal{D}_{\text{in}}$, $\eta_k \in \mathcal{D}$ do
    $d := \varepsilon_k - \eta_k + c_R$
    if ($b$ divides $d$) and ($\frac{d}{b} \notin \mathcal{C}$) then
      $c_L$ :=$d/b$
      PUSH($\mathcal{C}$,$c_L$)
      PUSH($N$,$c_L$)
    end if
  end for
end while
```

Remark 4.6. While our definition and the algorithm for the carry set are valid for a d-dimensional digit set $\mathcal{D} = \bar{\mathcal{D}}^d$, it is obviously more straightforward to compute the carry set $\mathcal{C}_0$ as the carry set associated to $\mathcal{D}_0$ and then set $\mathcal{C} = \mathcal{C}_0^d$.

Example 4.7. We want to compute the carry set $\mathcal{C}$ for the case $b = \tau$, $\mu = 1$ and $\mathcal{D} = \{0, \pm 1\}$. Table 2 shows the set N from Algorithm 11 with the element under examination in bold in the first line and the discovered carries below.

$\boldsymbol{N}$	$\{\boldsymbol{0}\}$	$\{\boldsymbol{1-\tau}\}$	$\{\boldsymbol{-1}, -\tau\}$	$\{\boldsymbol{-\tau}, -1+\tau\}$
c_L	$1-\tau = \frac{1+1+0}{\tau}$	$-1 = \frac{0-1+(1-\tau)}{\tau}$ $-\tau = \frac{0+1+(1-\tau)}{\tau}$	$-1+\tau = \frac{0-1+(-1)}{\tau}$	

$\boldsymbol{N}$	$\{\boldsymbol{-1+\tau}\}$	$\{\boldsymbol{\tau}, 1\}$	$\{\boldsymbol{1}, 2-\tau\}$	$\{\boldsymbol{2-\tau}\}$
c_L	$\tau = \frac{0-1+(-1+\tau)}{\tau}$ $1 = \frac{0+1+(-1+\tau)}{\tau}$	$2-\tau = \frac{1+1+\tau}{\tau}$		$1-2\tau = \frac{1+1+(2-\tau)}{\tau}$

$\boldsymbol{N}$	$\{\boldsymbol{1-2\tau}\}$	$\{\boldsymbol{-2}, -1-\tau\}$	$\{\boldsymbol{-1-\tau}\}$	$\{\boldsymbol{-2+\tau}\}$
c_L	$-2 = \frac{0-1+(1-2\tau)}{\tau}$ $-1-\tau = \frac{0+1+(1-2\tau)}{\tau}$		$-2+\tau = \frac{0-1+(-1-\tau)}{\tau}$	

Table 2: Computing the carry set for base τ, $\mathcal{D} = \{0, \pm 1\}$ and $\mu = 1$

So in this case the complete carry set is given by

$$\mathcal{C} = \{0, \pm 1, \pm\tau, \pm(\tau-1), \pm(\tau-2), 1-2\tau, -2, -\tau-1\},$$

has 12 elements and we observe that it is not symmetric.

Example 4.8. Somewhat surprisingly, the carry set $\mathcal{C}$ for $b = \tau$, $\mathcal{D} = \{0, \pm 1\}$ as above, but $\mu = -1$ is quite a bit smaller than the one for $\mu = 1$. It is given by

$$\mathcal{C} = \{0, \pm 1, \pm\tau, \pm(1+\tau), -2-\tau\}$$

and has only 8 elements.

So apparently the size of the carry set depends on the size of the digit set as well as the base used and for $b = \tau$, even the choice of the curve parameter $\mu \in \{\pm 1\}$ has a non-negligable impact. Next we see that the digit set is not necessarily a subset of the carry set.

Example 4.9. Let $b = 2$ and $\mathcal{D}$ a finite digit set with $\delta := \max\{|d| \mid d \in \mathcal{D}\}$ and $\{\pm\delta\} \subseteq \mathcal{D}$. We want to check if $-\delta$ is in the carry set. By Proposition 4.5, we get that if this were the case, we could find $\eta \in \mathcal{D}, \varepsilon \in \mathcal{D}_{\text{in}}$ and c_R already in the carry set, such that

$$\begin{aligned} -\delta &= \frac{c_R + \varepsilon - \eta}{2} \quad \text{and hence} \\ c_R &= -2\delta + \eta - \varepsilon \\ &\leq -2\delta + \delta - 0 = -\delta, \end{aligned}$$

which means that $c_R = -\delta$. So in Algorithm 11, $-\delta$ would already have to be in the carry set, to be discovered as a new carry. Since the carry set is initialized with $\{0\}$, this can not happen.

w	$\lvert\mathcal{D}\rvert$	$\lvert\mathcal{C}\rvert$		
		$b=2$	$b=\tau,\ \mu=1$	$b=\tau,\ \mu=-1$
2	3	2	12	8
3	5	6	27	28
4	9	14	85	75
5	17	30	159	178

Table 3: Dimensions of digit and carry sets

Remark 4.10. In the examples presented in [26], the carry set does include the element $-\delta$ as described above. It is unclear why that is the case considering that the algorithm to compute the carry set is equivalent to ours. However as a consequence, for $b = 2$, the carry sets we use are smaller, which reduces the number of states that our algorithm goes through.

Table 3 gives an overview for the behavior of this size depending on the input parameters. As we will see, the complexity of the minimal weight algorithm and the size of the model we will use to compute the minimal average weight also depend strongly on the size of the carry set. For joint expansions, the numbers in Table 3 have to be taken to the power of d and ultimately this restricts the feasibility of our approach to relatively small values of d and w.

4.1.2 Minimal weight strategy

Now that we have identified the carry set, we know the domains for all the terms in Equation (9). Our goal is to give a recoding, i.e. specify the left hand side of the equation. The only variable that is predetermined (by the input) is $\varepsilon_k \in \mathcal{D}_{\text{in}}$, the input digit at step k. The carry c_R, from the right is not known at this point, because it depends on the recoding of $(\varepsilon_{k-1}\varepsilon_{k-2}\dots\varepsilon_0)$, which is yet to be computed. So we have to consider all possible $c_R \in \mathcal{C}$. For any given c_R, we can compute all possible values for c_L and η_k:

Definition 4.11. For $v \in \mathcal{R}$, we define

$$\operatorname{Cand}(v) = \{(c, \eta) \in \mathcal{C} \times \mathcal{D} \mid b \cdot c + \eta = v\},$$

the set of candidates for encoding the value v.

Cand(v) can be computed by running through all $\eta \in \mathcal{D}$ and checking for divisibility of $v - \eta$ by b (See Algorithm 12).

All pairs $(c_L, \eta_k) \in \operatorname{Cand}(\varepsilon_k + c_R)$ would yield a valid encoding, but we are looking for an encoding of *minimal weight*.

Definition 4.12. Let $(\dots 0\,\varepsilon_{l-1}\dots\varepsilon_1\varepsilon_0)$ be the input. To keep track of weight-information we define $w_k \colon \mathcal{C} \to \mathbb{N}_0$ $(0 \le k \le l)$ as

$$\begin{aligned} w_l(c_R) &:= \operatorname{mw}_{\mathcal{D}}(c_R), \\ w_k(c_R) &:= \min\{w_{k+1}(c) + \operatorname{weight}(\eta) \mid (c, \eta) \in \operatorname{Cand}(\varepsilon_k + c_R)\} \quad (0 \le k < l). \end{aligned}$$

Algorithm 12 Cand(v): Computing all possible carry-representations of a value $v \in \mathcal{R}$

```
Input: The input value $v \in \mathcal{R}$
Output: A list $C$ of all tuples $(c, \eta) \in \mathcal{C} \times \mathcal{D}$ with $bc + \eta = v$
  $C = \{\}$
  for all $\eta \in \mathcal{D}$ do
    if $b$ divides $v - \eta$ then
      $C := C \cup \{(\frac{v-\eta}{b}, \eta)\}$
    end if
  end for
```

Proposition 4.13. *The function $w_k(c_R)$ gives the minimal weight of a $\mathcal{D}$-encoding of $\sum_{j=k}^{l-1} \varepsilon_j b^{j-k} + c_R$. The minimal weight among all $\mathcal{D}$-encodings of $\sum_{j=0}^{l-1} \varepsilon_j b^j$ is given by $w_0(0)$.*

Our definition of w doesn't tell us how to compute $w_l(c_R) = \mathrm{mw}_{\mathcal{D}}(c_R)$. The next result will give us more information about the initial state.

Definition 4.14. Define a sequence of functions $v_i \colon \mathcal{C} \to \mathbb{N}_0 \cup \{\infty\}$, $i \geq 0$ as

$$v_0(c_R) := \begin{cases} 0 & \text{if } c_R = 0, \\ \infty & \text{else,} \end{cases}$$
$$v_i(c_R) := \min\{v_{i-1}(c) + \mathrm{weight}(\eta) \mid (c, \eta) \in \mathrm{Cand}(c_R)\} \quad (k > 0).$$

Lemma 4.15. *Let $\mathcal{D}$ be a suitable digit set and $\mathcal{C}$ the corresponding carry set as before. There exists a $k \in \mathbb{N}$ such that $\forall c \in \mathcal{C} \colon v_k(c) = \mathrm{mw}_{\mathcal{D}}(c)$. Furthermore, we have $v_{k+j} = v_k$ for all $j \in \mathbb{N}_0$.*

Remark 4.16. This means that we can compute the elements of the sequence $(v_i)_{i \in \mathbb{N}_0}$ and as soon as we see the sequence stabilizing, i.e. find $k \in \mathbb{N}$ such that $v_k = v_{k+1}$, we set $w_l := v_k$.

Proof. We start with the second part, which is quite simple, because

$$\begin{aligned} v_{k+1}(c) &= \min\{v_k(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(c)\} \\ &= \min\{\mathrm{mw}_{\mathcal{D}}(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(c)\} \\ &= \mathrm{mw}_{\mathcal{D}}(c), \end{aligned}$$

therefore $v_k = v_{k+1}$ and inductively also $v_k = v_{k+j}$ for all $j \in \mathbb{N}$.
For the first part of the claim, we set

$$k = \max_{c \in \mathcal{C}}\{n \in \mathbb{N}_0 \mid \text{There exists a minimal weight } \mathcal{D}\text{-expansion } \eta \text{ of } c \text{ and } \mathrm{length}(\eta) = n\},$$

i.e. k is the greatest $\mathcal{D}$-*bitlength* of any minimal weight expansion of any $c \in \mathcal{C}$.

For $i \geq 0$, we set

$$\mathrm{mw}_{i,\mathcal{D}}(c) = \min\{\mathrm{weight}(\eta) \mid \eta = (\eta_{i-1}, \dots \eta_1, \eta_0) \text{ is a length } i \text{ } \mathcal{D}\text{-expansion of } c\}$$

where the convention $\min \emptyset = \infty$ is used. We show that

$$v_i(c) = \mathrm{mw}_{i,\mathcal{D}}(c) \tag{11}$$

holds for all $i \geq 0$ and $c \in \mathcal{C}$.

For $i = 0$, the right side only looks at expansions of length 0. Only $c = 0$ has such an expansion, which is the empty word and has weight 0 per convention. For any other c, the right side is $\min \emptyset$ which we consider to be ∞ and the case $i = 0$ is dealt with.
If the claim holds for i, we get

$$\begin{aligned} v_{i+1}(c) &= \min\{v_i(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(c)\} \\ &= \min\{\mathrm{mw}_{i,\mathcal{D}}(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(c)\} \\ &= \min\{\mathrm{mw}_{i+1,\mathcal{D}}(b \cdot c_L + \eta) \mid (c_L, \eta) \in \mathrm{Cand}(c)\} \\ &= \mathrm{mw}_{i+1,\mathcal{D}}(c). \end{aligned}$$

This proves the claim made in (11) and for $i = k$ we get $v_k(c) = \mathrm{mw}_{k,\mathcal{D}}(c)$. Since no minimal weight expansion of any $c \in \mathcal{C}$ is longer than k, we get $\mathrm{mw}_{k,\mathcal{D}}(c) = \mathrm{mw}_{\mathcal{D}}(c)$ and the proof is complete. □

4.1.3 Complete algorithm

For our algorithm to give us an actual minimal weight $\mathcal{D}$-expansion as an output, we have to keep track of the digits used to achieve minimality. This will not be used for the analysis of the average weight, but aides in understanding the algorithm and writing an example.

Definition 4.17. Let $c_R \in \mathcal{C}$ and

$$(c, \eta) \in \{(c, \eta) \in \mathrm{Cand}(\varepsilon_k + c_R) \mid w_{k+1}(c) + \mathrm{weight}(\eta) = w_k(c_R)\} \neq \{0\}$$

arbitrarily chosen. For $0 \leq k < l$ set

$$\begin{aligned} \eta_k(c_R) &:= \eta \\ c_{L,k}(c_R) &:= c \end{aligned}$$

to be the information necessary to generate a minimal weight expansion after the minimal weight has been computed.

With this terminology we can compute the expansion by setting:

$$\begin{aligned} \eta_0 &= \eta_0(0) \\ \eta_k &= \eta_k(c_{L,k-1}) \quad 0 < k < l \end{aligned}$$

Algorithm 13 puts together all the pieces and in Example 4.18, we see it at work.

Algorithm 13 algMinWeightExp: Computing a minimal joint weight expansion

```
**Input:** The input expansion $\varepsilon_{l-1} \dots \varepsilon_1 \varepsilon_0$ and the target digit set $\mathcal{D}$
**Output:** The minimal weight expansion $\eta_{l-1} \dots \eta_1 \eta_0$ with digit set $\mathcal{D}$
  $\mathcal{C}$ :=algCarryset($\mathcal{D}$)
  for all $c_R \in \mathcal{C}$ do
    $w_l(c_R)$ :=mw$_\mathcal{D}(c_R)$
  end for
  $k$ :=$l - 1$
  while $k > 0$ do
    for all $c_R \in \mathcal{C}$ do
      $w_k(c_R)$ :=min$\{w_{k+1}(c) + \text{weight}(\eta) \mid (c, \eta) \in \text{Cand}(\varepsilon_k + c_R)\}$
      $(c_{L,k}(c_R), \eta_k(c_R))$ :=First$(\{(c, \eta) \in \text{Cand}(\varepsilon_k + c_R) \mid w_{k+1}(c) + \text{weight}(\eta) = w_k(c_R)\})$
      $k$ :=$k - 1$
    end for
  end while
  $w_0(0)$ :=min$\{w_1(c) + \text{weight}(\eta) \mid (c, \eta) \in \text{Cand}(\varepsilon_0)\}$
  $(c_{L,0}(0), \eta_0(0))$ :=First$(\{(c, \eta) \in \text{Cand}(\varepsilon_0) \mid w_{0+1}(c) + \text{weight}(\eta) = w_0(0)\})$
  $\eta_0$ :=$\eta_0(0)$
  $k$ :=$k + 1$
  while $k < l$ do
    $\eta_k$ :=$\eta_k(c_{L,k-1})$
    $k$ :=$k + 1$
  end while
```

Example 4.18. In the following example, we shall compute a minimal weight expansion for $z = 374$, given in its binary representation $(101110110)_2$, i.e. $b = 2$. We will use the digit set $\mathcal{D} = \{0, \pm 1\}$ for which the carry set evaluates to $\mathcal{C} = \{0, 1\}$.

Our first operation is setting $w_9(0) = 0$ and $w_9(1) = 1$. We then start our iteration for $k = 8$, where we see an input bit of $\varepsilon_8 = 1$. Looking at all carries c_R, we compute:

- $\boldsymbol{c_R = 0}$: This yields $\varepsilon_k + c_R = 1$, and $\text{Cand}(1) = \{(0, 1), (1, -1)\}$.
  Now $w_9(0) + \text{weight}(1) = 0 + 1 = 1$ and $w_9(1) + \text{weight}(-1) = 2$. So we pick $(c_{L,8}(0), \eta_8(0)) = (0, 1) \in \text{Cand}(1)$ to get $w_8(0) = 1$.

- $\boldsymbol{c_R = 1}$: This yields $\varepsilon_k + c_R = 2$, and $\text{Cand}(2) = \{(1, 0)\}$.
  Now $w_9(1) + \text{weight}(0) = 1$ and we set $(c_{L,8}(1), \eta_8(1)) = (1, 0)$ to get $w_8(1) = 1$.

This process is now repeated for decreasing k and Table 4 shows all the steps. Note that all elements from $\text{Cand}(\varepsilon_k + c_R)$ are listed, the first one with minimal weight is chosen for $(c_{L,k}(c_R), \eta_k(c_R))$. In the last row we find our result for the minimum weight, which is $w_0(0) = 4$. Now we backtrack up through the table to get the output expansion with that weight, which is $11000(-1)0(-1)0$. The corresponding $\eta_k(c_R)$ are marked bold in the table.

k	ε_k	c_R	$\varepsilon_k + c_R$	Cand	$w_k(c_R)$	$c_{L,k}(c_R)$	$\eta_k(c_R)$
9		0			0		
		1			1		
8	1	0	1	$\{(0,1),(1,-1)\}$	1	0	**1**
		1	2	$\{(1,0)\}$	1	1	0
7	0	0	0	$(0,0)$	1	0	0
		1	1	$\{(0,1),(1,-1)\}$	2	0	**1**
6	1	0	1	$\{(0,1),(1,-1)\}$	2	0	1
		1	2	$(1,0)$	2	1	**0**
5	1	0	1	$\{(0,1),(1,-1)\}$	3	0	1
		1	2	$(1,0)$	2	1	**0**
4	1	0	1	$\{(0,1),(1,-1)\}$	3	1	-1
		1	2	$(1,0)$	2	1	**0**
3	0	0	0	$(0,0)$	3	0	0
		1	1	$\{(0,1),(1,-1)\}$	3	1	**−1**
2	1	0	1	$\{(0,1),(1,-1)\}$	4	0	1
		1	2	$(1,0)$	3	1	**0**
1	1	0	1	$\{(0,1),(1,-1)\}$	4	1	**−1**
		1	2	$(1,0)$	3	1	0
0	0	0	0	$(0,0)$	4	0	**0**

Table 4: MinWeight computation for input 101110110

4.2 Asymptotic analysis

As stated at the beginning of this chapter, the minimal weight algorithm is just a stepping stone on our way to computing the minimal average joint weight of digit expansions in a given digit set $\mathcal{D}$. To do this we no longer look at individual input bit-strings, but rather consider all possible inputs of arbitrary length and see how the minimal weight behaves asymptotically. We do so by building an automaton that models the algorithm's behavior. We have to define a set of states that our algorithm traverses through on an arbitrary length input string and hope that we can find a way to keep this set finite. Fortunately the authors of [26] have suggested just that and their idea can be applied to base-τ-expansions and their respective digit sets as well.

Our computation of the minimal asymptotic density is in effect an automated proof for the minimal weight of all digit expansions using the same digit sets. Section 4.2.2 deals mostly in proving that our algorithm terminates, by showing that the number of states it has to explore is finite (Theorem 1). Our methods are different from what was done by the authors of [26] and our proof covers expansions of base τ as well as base 2. We have to note however that the results leading up to Theorem 1 only cover the case $d = 1$ and so the guarantee for the state space exploration to terminate can not be given for $d \geq 2$. As can be seen in the presentation of our results, our analysis method also succeeds in quite a few cases with $d \geq 2$ and while the number of states grows exponentially with d, we strongly believe that even for the cases where memory constraints of our computational environment prevent the computation of all the states, the number of states remains finite (albeit very large).

4.2.1 States and transitions

When looking at the way the minimum weight algorithm works, it becomes apparent that all the information concerning minimum weight is concentrated in the functions

$$w_k \colon \mathcal{C} \to \mathbb{N}_0 \qquad (k \geq 0)$$

and the values for these functions depend only on the input at step k and on the previous weights, w_{k+1}. So it would seem like a good idea to use the w_k as states and have a transition $w_{k+1} \xrightarrow{\varepsilon} w_k$, whenever we arrive at state w_k, after reading the input ε in state w_{k+1}. Unfortunately the values $w_k(c)$ tend to grow with the length of the input (unless it is a null-string, which doesn't concern us as we are looking at randomized input) and so there is definitely an infinite number of such states.
Even though for any state w_k, the values $w_k(c)$ are not individually bounded, one could conjecture that they are relatively close together. Hence, the idea will be to just look at the variability of $w_k(c)$ for different $c \in \mathcal{C}$.

Definition 4.19. Let $W = \{w \colon \mathcal{C} \to \mathbb{Z} \cup \infty\}$, then we define the equivalence relation $\equiv$ for any $w_\alpha, w_\beta \in W$ as

$$w_\alpha \equiv w_\beta \Leftrightarrow \exists z \in \mathbb{Z}\ \forall c \in \mathcal{C} \colon w_\alpha(c) = w_\beta(c) + z.$$

Example 4.20. In Example 4.18, we see w_6 and $w_8 \in W$ satisfying $w_6 \equiv w_8$, because $\forall c \in \mathcal{C} \colon w_6(c) = w_8(c) + 1$.

In the example above we see a second equivalence, between w_5 and w_7, which are direct successors (on same input $\varepsilon_6 = \varepsilon_8 = 1$) of w_6 and w_8 respectively.

Definition 4.21. We define the *transition function* by

$$\begin{aligned} &\mathrm{t} \colon W \times \mathcal{D}_{\text{in}} \to W \\ &\mathrm{t}(w_\alpha, \varepsilon) = w_\beta, \end{aligned}$$

with w_β defined by $w_\beta(c) = \min\{w_\alpha(c_L) + \text{weight}(\eta) \mid (c_L, \eta) \in \text{Cand}(\varepsilon + c)\}$.

Lemma 4.22. *The transition function is compatible with our definition of $\equiv$, i.e.*

$$\forall w, v \in W \colon w \equiv v \Rightarrow \mathrm{t}(w, \varepsilon) \equiv \mathrm{t}(v, \varepsilon).$$

Proof. By definition, there is a $z \in \mathbb{Z}$ such that $w(c) = v(c) + z$ holds for all $c \in \mathcal{C}$. Thus

$$\begin{aligned}
\mathrm{t}(w, \varepsilon)(c) &= \min\{w(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(\varepsilon + c)\} \\
&= \min\{(v(c_L) + z) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(\varepsilon + c)\} \\
&= z + \min\{v(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(\varepsilon + c)\} \\
&= z + \mathrm{t}(v, \varepsilon)(c) \\
&\equiv \mathrm{t}(v, \varepsilon)(c).
\end{aligned}$$

□

Thanks to Lemma 4.22, we can now consider our transition function as a function

$$\mathrm{t} \colon \bar{V} \times \mathcal{D}_{\mathrm{in}} \to \bar{V},$$

where $\bar{V} := W/_{\equiv}$ will be called the *state space* and we will represent its classes by the unique member $w \in W$ with $w(0) = 0$. Sometimes we will write a succession of transitions from v_0 to v_n with $v_{k+1} = \mathrm{t}(v_k, \varepsilon_k)$ $(0 \le k \le n-1)$ as $\mathrm{t}(v_0, \varepsilon_0\varepsilon_1 \dots \varepsilon_{n-1}) = v_n$.
Next we introduce a notation to preserve the information about the change of weight that occurs with such a transition. We introduce the function $\mathrm{wc} \colon \bar{V} \times \mathcal{D}_{\mathrm{in}} \to \mathbb{Z}$, with

$$\mathrm{wc}(w, \varepsilon) = \min\{w(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(\varepsilon)\}.$$

Now we can finally give the description of our model of the algorithm:

Definition 4.23. We define the *initial state* $w_0 \in \bar{V}$ as

$$w_0(c) = \mathrm{mw}_{\mathcal{D}}(c)$$

and the set $V \subset \bar{V}$ of states that Algorithm 13 can actually reach, by

- $w_0 \in V$
- $\forall w \in V,\, \varepsilon \in \mathcal{D}_{\mathrm{in}} \colon\ \mathrm{t}(w, \varepsilon) \in V$.

The set $E \subseteq V \times V \times \mathbb{Z}$ of all possible transitions is given by

$$E = \{(v, w, \rho) \in V \times V \times \mathbb{Z} \mid \exists\, \varepsilon \in \mathcal{D}_{\mathrm{in}} \colon\ \mathrm{t}(v, \varepsilon) = w,\ \mathrm{wc}(v, \varepsilon) = \rho\}.$$

Now the states and transitions form a directed graph $G = (V, E)$. We call it the *state graph* for Algorithm 13. It is of different structure for every choice of b, d and $\mathcal{D}$ and the number of states also depends on the size of the carry set.

The initial state of the state graph is w_0. On every read input bit ε_i, the transition $w \to \mathrm{t}(w, \varepsilon_i)$ is made. After reading the input $\varepsilon_{l-1}\varepsilon_{l-2} \dots \varepsilon_k$ the current state is $v \in V$, which by our choice of the representative we can write as

$$v(c) = \mathrm{mw}_{\mathcal{D}}\left(c + \sum_{j=k}^{l-1} \varepsilon_j b^{j-k}\right) - \mathrm{mw}_{\mathcal{D}}\left(\sum_{j=k}^{l-1} \varepsilon_j b^{j-k}\right).$$

The value of $\mathrm{wc}(v, \varepsilon_k)$ gives the (signed) change of minimal weight after reading ε_k, i.e.

$$\mathrm{wc}(v, \varepsilon_k) = \mathrm{mw}_{\mathcal{D}}\left(\sum_{j=k}^{l-1} \varepsilon_j b^{j-k}\right) - \mathrm{mw}_{\mathcal{D}}\left(\sum_{j=k+1}^{l-1} \varepsilon_j b^{j-k-1}\right).$$

4.2.2 The Markov chain model and its stationary distribution

After the careful design of the graph (V, E) for the minimum weight algorithm, we will now prove a few properties that enable us to apply the knowledge about Markov chains, we introduced in Chapter 1.3 to get stationary distribution of (V, E), respectively the Markov chain that we will associate it to. This distribution holds the information about the probability that the algorithm is in a certain state after reading long inputs, which in turn will give us the data needed to compute the minimal average joint weight.
First, we show that V is in fact finite.

Note that as stated in the beginning of this chapter, the results leading up to and including 1 are only valid for $d = 1$.

Lemma 4.24. *For any two digits $\varepsilon, \delta \in \mathcal{D}$, we have* $\operatorname{mw}_{\mathcal{D}}(\varepsilon + \delta) \leq 2$

Proof. Consider the case $b = 2$: If $\varepsilon = 0$ or $\delta = 0$, the claim is trivially true, hence let $\varepsilon \neq 0$ and $\delta \neq 0$. Our digit sets for base 2 always have the form $\{0, \pm 1, \ldots, \pm(2h+1)\}$ for some $h \in \mathbb{N}_0$. Let $k = \max\{n \in \mathbb{N} \mid \frac{\varepsilon+\delta}{2^n} \in \mathbb{Z}\}$. Since we know that $\varepsilon + \delta$ is even, we know that $k \geq 1$ and therefore $\gamma = \frac{\varepsilon+\delta}{2^k}$ is odd and $|\gamma| \leq \frac{|\varepsilon|+|\delta|}{2} \leq \max\{|d| \mid d \in \mathcal{D}\}$. This means that $\gamma \in \mathcal{D}$ and $(\ldots 0\gamma\underbrace{0\ldots0}_{k})$ is an expansion of $\varepsilon + \delta$ of weight 1.

For base τ, we recall that since we are using the minimal norm representatives as our digit set, we get that $\varepsilon \in \mathbb{Z}[\tau] \setminus \{0\}$, coprime to τ is in $\mathcal{D}$ if and only if $\varepsilon \in \tau^w V$, where V is the Voronoi region defined in [23, Section 5.1]. In the following we will use subset-relations between different sums of complex multiples of V. Since these expressions do not maintain the original shape of V, we will use U, the circumdisc of V and I, the inndisc of V. Since V is convex and symmetric with respect to inversion through 0, I has its center in 0, and its radius r_I is given by the closest distance that any of the edges of V pass by 0. The edges of V are the borders between the Voronoi region of 0 and the Voronoi regions of $\pm 1, \pm\tau, \pm 1 \pm \tau$, thus we get

$$r_I = \frac{1}{2}\min\{|1|, |\tau|, |1+\tau|, |1-\tau|\} = \frac{1}{2}.$$

All the vertices of V have absolute value $\frac{2}{\sqrt{7}}$ and together with its convexity this means that U has a radius of $\frac{2}{\sqrt{7}}$. Summing up, we have

$$U = \left\{z \in \mathbb{C} \mid |z| \leq \frac{2}{\sqrt{7}}\right\},$$
$$I = \left\{z \in \mathbb{C} \mid |z| \leq \frac{1}{2}\right\} \quad \text{and}$$
$$I \subseteq V \subseteq U.$$

Additionally, let $\tilde{V} := \tau^w V$, $\tilde{U} := \tau^w U = 2^{w/2} U$ and $\tilde{I} := \tau^w I = 2^{w/2} I$ and in analogy to the previous case, let $k = \max\{n \in \mathbb{N} \mid \frac{\varepsilon+\delta}{\tau^n} \in \mathbb{Z}[\tau]\}$.

$\boldsymbol{\varepsilon}$	$\boldsymbol{\delta}$	$\boldsymbol{\varepsilon+\delta}$	optimal expansion
1	1	2	$(-1-\tau, 0)$
1	$1+\tau$	$2+\tau$	$(-1, 0, 0)$
1	$-1-\tau$	$-\tau$	$(-1, 0)$
$1+\tau$	$1+\tau$	$2+2\tau$	$(-1, 1, 0)$

Table 5: Sums of two digits $\varepsilon+\delta$ for $w=4$ and $\mu=-1$

If $k \geq 4$, let $\gamma := \frac{\varepsilon+\delta}{\tau^k}$. We get

$$\gamma \in \frac{2\tilde{V}}{\tau^k} \subseteq \frac{2}{2^{k/2}} \cdot \tilde{U} = \frac{2}{2^{k/2}} \cdot \frac{4}{\sqrt{7}}\tilde{I} \subseteq \frac{2}{2^{4/2}} \cdot \frac{4}{\sqrt{7}}\tilde{I} \subseteq \tilde{I} \subseteq \tilde{V}$$

and since γ is not divisible by τ it follows that $\gamma \in \mathcal{D}$ and $(\ldots 0\gamma\overbrace{0\ldots0}^{k})$ is an expansion of $\varepsilon+\delta$ of weight 1.

For $k \in \{1,2,3\}$, we pick $\eta \in \mathcal{D}$ such that $\eta \equiv \frac{\varepsilon+\delta}{\tau^k} \pmod{\tau^w}$ which exists and is unique since $\mathcal{D}$ contains exactly one representative for every residue class modulo τ^w. Now let $r = \max\{n \in \mathbb{N} \mid \frac{\varepsilon+\delta-\tau^k\eta}{\tau^n} \in \mathbb{Z}[\tau]\}$ and $\gamma = \frac{\varepsilon+\delta-\tau^k\eta}{\tau^r}$. Then

$$\gamma \in \frac{2\tilde{V}+\tau^k\tilde{V}}{\tau^r} \subseteq \frac{2+2^{k/2}}{2^{r/2}}\tilde{U} = \underbrace{\frac{2+2^{k/2}}{2^{r/2}} \cdot \frac{4}{\sqrt{7}}}_{=:f(k,r)}\tilde{I}.$$

We get

$$f(k,r) \leq 1 \Leftarrow \begin{cases} k=1 \text{ and } r \geq 5, \text{ or} \\ k=2 \text{ and } r \geq 6, \text{ or} \\ k=3 \text{ and } r \geq 6 \end{cases}$$

and since we know that $r \geq k+w$, the proof is complete for all $w \geq 4$ and an expansion of $\varepsilon+\delta$ of weight 2 is given by $(\ldots 0\gamma\underbrace{0\ldots0\eta0}_{r})$.

For $w=2$ it is easily verified, that since $\mathcal{D}=\{0,\pm1\}$, the only interesting case is $\varepsilon+\delta=\pm2$ and we get $\frac{\varepsilon+\delta}{\tau} = \pm\mu \mp \tau$, which yields a trivial $\mathcal{D}$-expansion of weight 2.

For $w=3$ the digit set has 4 non-zero elements. There are 12 nonzero sums of the form $\varepsilon+\delta$ ($\varepsilon,\delta \in \mathcal{D}$) and only 4 of them need to be checked (the rest follows from symmetry). Table 5 and Table 6 show those four elements for $\mu=-1$ and $\mu=1$ respectively and we see that all their minimal expansions have weight less than or equal to 2.

□

Proposition 4.25.

$$\forall x,y \in \mathcal{R}\colon\ \mathrm{mw}_{\mathcal{D}}(x+y) \leq \mathrm{mw}_{\mathcal{D}}(x) + \mathrm{mw}_{\mathcal{D}}(y).$$

$\boldsymbol{\varepsilon}$	$\boldsymbol{\delta}$	$\boldsymbol{\varepsilon+\delta}$	optimal expansion
1	1	2	$(1-\tau,0)$
1	$1-\tau$	$2-\tau$	$(-1,0,0)$
1	$-1+\tau$	τ	$(1,0)$
$1-\tau$	$1-\tau$	$2-2\tau$	$(1,0,0,1,0)$

Table 6: Sums of two digits $\varepsilon+\delta$ for $w=4$ and $\mu=1$

Proof. Given $x,y\in\mathcal{R}$, we will prove the proposition by constructing a $\mathcal{D}$-expansion of $x+y$ with weight $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$. Any minimal weight expansion of $x+y$ then has weight less or equal to $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$. In the following let $(\alpha_j)_{j\geq 0}=(\ldots,\alpha_1,\alpha_0)$ be a minimal weight expansion of x and $(\beta_j)_{j\geq 0}=(\ldots,\beta_1,\beta_0)$ be a minimal weight expansion of y. Furthermore let

$$n=\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)\quad\text{and}\quad v=v(x,y)=\max\left\{k\in\mathbb{N}\mid\frac{x+y}{b^k}\in\mathcal{R}\right\}.$$

Using induction on (n,v) we are going to prove that $x+y$ has a $\mathcal{D}$-expansion of weight $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$:

$\underline{n=0}$: The claim is trivially true since $x=y=x+y=0$.

$\underline{v=0,n>0}$: Without loss of generality, let $\alpha_0=0$ and $\beta_0\neq 0$. Define $\tilde{x}=\frac{x}{b}$ and $\tilde{y}=\frac{y-\beta_0}{b}$. Applying the induction hypothesis for $(n-1,*)$ we get an expansion $(\ldots,\gamma_1,\gamma_0)$ of $\tilde{x}+\tilde{y}$ with weight $\mathrm{mw}_{\mathcal{D}}(\tilde{x})+\mathrm{mw}_{\mathcal{D}}(\tilde{y})$. Now $(\ldots,\gamma_1,\gamma_0,\beta_0)$ is an expansion of $x+y$ with weight $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$.

$\underline{n>0,\ v>0}$: There are two cases:

- $\alpha_0=\beta_0=0$:
  Let $\tilde{x}=\frac{x}{b}$ and $\tilde{y}=\frac{y}{b}$. Now $v(\tilde{x},\tilde{y})=v(x,y)-1$ and $\mathrm{mw}_{\mathcal{D}}(\tilde{x})+\mathrm{mw}_{\mathcal{D}}(\tilde{y})=\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$. So via the induction hypothesis for $(n,v-1)$, we get an expansion $(\ldots,\gamma_1,\gamma_0)$ of $\tilde{x}+\tilde{y}$ with weight $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$. Now $(\ldots,\gamma_1,\gamma_0,0)$ is an expansion of $x+y$ with weight $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$.

- $\alpha_0\neq 0,\ \beta_0\neq 0$:
  Let $\tilde{x}_1=\frac{x+\beta_0}{b}$, $\tilde{y}_1=\frac{y-\beta_0}{b}$. Now we use the induction hypothesis twice: Let $\tilde{x}_2=x$ and $\tilde{y}_2=\beta_0$. We want to use the hypothesis for $(n-1,*)$, but $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(\beta_0)=\mathrm{mw}_{\mathcal{D}}(x)+1<n$ is only true if $\mathrm{mw}_{\mathcal{D}}(y)>1$. We will check the case $\mathrm{mw}_{\mathcal{D}}(y)=1$ separately below. For $\mathrm{mw}_{\mathcal{D}}(y)>1$, the hypothesis gives us $\mathrm{mw}_{\mathcal{D}}(\tilde{x}_1)\leq\mathrm{mw}_{\mathcal{D}}(x)+1$. So we can use it a second time on $\tilde{x}_1$ and $\tilde{y}_1$ because $v(\tilde{x}_1,\tilde{y}_1)=v-1$ and $\mathrm{mw}_{\mathcal{D}}(\tilde{x}_1)+\mathrm{mw}_{\mathcal{D}}(\tilde{y}_1)\leq n$. Once more we get an expansion $(\ldots,\gamma_1,\gamma_0)$ of $\tilde{x}_1+\tilde{y}_1$ with weight $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$. Now $(\ldots,\gamma_1,\gamma_0,0)$ is an expansion of $x+y$ with weight $\mathrm{mw}_{\mathcal{D}}(x)+\mathrm{mw}_{\mathcal{D}}(y)$.
  $\underline{\mathrm{mw}_{\mathcal{D}}(y)=1}$:
  If $\mathrm{mw}_{\mathcal{D}}(x)>1$, we can set $\tilde{x}=y$, $\tilde{y}=x$ and continue above. So we consider the case $\mathrm{mw}_{\mathcal{D}}(x)=\mathrm{mw}_{\mathcal{D}}(y)=1$. Since $\alpha_0\neq 0$ and $\beta_0\neq 0$, it follows that $x,y\in\mathcal{D}$ and Lemma 4.24 completes the proof.

□

Proposition 4.26. *For any choice of b and $\mathcal{D}$, we can find an $M \in \mathbb{N}$ such that for all $w \in V$ in our state graph,*

$$\forall c \in \mathcal{C} \colon |w(c)| \leq M$$

Proof. First off, we recall that $|w(c)| = |\operatorname{mw}_{\mathcal{D}}(z+c) - \operatorname{mw}_{\mathcal{D}}(z)|$ for some $z \in \mathcal{R}$. By Proposition 4.25, $\operatorname{mw}_{\mathcal{D}}(z+c) \leq \operatorname{mw}_{\mathcal{D}}(z) + \operatorname{mw}_{\mathcal{D}}(c)$ and if we set $\bar{z} := z + c$ and $\bar{c} = -c$, we get

$$\operatorname{mw}_{\mathcal{D}}(z) = \operatorname{mw}_{\mathcal{D}}(\bar{z} + \bar{c}) \leq \operatorname{mw}_{\mathcal{D}}(\bar{z}) + \operatorname{mw}_{\mathcal{D}}(\bar{c}) = \operatorname{mw}_{\mathcal{D}}(z+c) + \operatorname{mw}_{\mathcal{D}}(c)$$

So summing up, we get

$$|\operatorname{mw}_{\mathcal{D}}(z) - \operatorname{mw}_{\mathcal{D}}(z+c)| \leq \operatorname{mw}_{\mathcal{D}}(c)$$

and to complete the proof we set $M = \max_{c \in \mathcal{C}}\{\operatorname{mw}_{\mathcal{D}}(c)\}$. □

Theorem 1. *The set V is finite.*

Proof. Let $C = \{c_0, c_1, \ldots, c_{|\mathcal{C}|})$ and every $w \in V$ be represented by the vector $u_w = (w(c_0), w(c_1), \ldots, w(c_{|\mathcal{C}|}))$. Then, by Proposition 4.26, we have $|w(c)| \leq M$, so u_w is an element of $\{-M, -(M-1), \ldots, M\}^{|\mathcal{C}|}$, which is a finite set. □

Remark 4.27. From now on we will write $V = \{w_0, w_1, \ldots, w_m\}$, where w_0 is the initial state as in the definition of V and the remaining states are numbered in an arbitrarily but fixed order. For $d = 1$ we can do this because of Theorem 1. For $d > 1$ we can do it if the state generation algorithm terminates.

Lemma 4.28. *The matrix $P = (p_{ij})_{0 \leq i,j \leq m} \in [0,1]^{m \times m}$, with*

$$p_{ij} = \frac{1}{|\mathcal{D}_{in}|} |\{(v, w, \rho) \in E \mid v = v_i,\ w = v_j\}|$$

satisfies $\forall i \in \{0, 1, \ldots, m\}\colon$

$$\begin{aligned}\sum_{j=0}^{n} p_{ij} &= \frac{1}{|\mathcal{D}_{in}|} \sum_{j=0}^{n} |\{(v, w, \rho) \in E \mid v = v_i,\ w = v_j\}| \\ &= \frac{1}{|\mathcal{D}_{in}|} |\{(v, w, \rho) \in E \mid v = v_i\}| = \frac{1}{|\mathcal{D}_{in}|} |\mathcal{D}_{in}| = 1\end{aligned}$$

Definition 4.29. Let (V, E) be the state graph for Algorithm 13. We define the corresponding discrete time, time homogeneous Markov chain $(X_n)_{n \geq 0}$ with the finite set of states $V = \{w_0, w_1, \ldots w_{m-1}\}$, by giving its initial distribution $(\lambda_i)_{0 \leq i < m}$, with $\lambda_0 = 1$ and $\lambda_j = 0$ for $0 < j < m$ and transition matrix P as above.

Lemma 4.30. *Let $x \in \mathcal{R}$, then*

$$\forall y \in \mathcal{R}\ \exists k_y \in \mathbb{N} \colon \forall k > k_y \colon \ \mathrm{mw}_{\mathcal{D}}(xb^k + y) = \mathrm{mw}_{\mathcal{D}}(x) + \mathrm{mw}_{\mathcal{D}}(y)$$

.

Proof. By Proposition 4.25, we know

$$\mathrm{mw}_{\mathcal{D}}(xb^k + y) \le \mathrm{mw}_{\mathcal{D}}(x) + \mathrm{mw}_{\mathcal{D}}(y).$$

Let $x \in \mathcal{R}$. With $\delta := \max\{\mathrm{mw}_{\mathcal{D}}(c) \mid c \in \mathcal{C}\}$, define:

$$\mathrm{Exp}(\mathcal{D}, n, x) := \{(\varepsilon_i)_{i \in \mathbb{N}_0} \mid (\varepsilon_i)_{i \in \mathbb{N}_0} \text{ is a } \mathcal{D}\text{-expansion of } x \text{ and } \exists m \ge n \colon \varepsilon_m \ne 0\} \text{ and}$$
$$k_y := \min\{n \in \mathbb{N} \mid \forall (\varepsilon_i)_{i \in \mathbb{N}_0} \in \mathrm{Exp}(\mathcal{D}, n, y) \colon \ \mathrm{weight}(\varepsilon_{n-1}, \dots, \varepsilon_0) > \mathrm{mw}_{\mathcal{D}}(y) + \delta\}$$

Let $k > k_y$ and $(\dots, \gamma_1, \gamma_0)$ be a minimal weight expansion of $xb^k + y$. If $(\gamma_{k-1}, \dots, \gamma_1, \gamma_0)$ is an expansion of y, then $(\dots, \gamma_{k+1}, \gamma_k)$ is an expansion of x and both must be of minimal weight. Then we have: $\mathrm{mw}_{\mathcal{D}}(xb^k + y) = \mathrm{mw}_{\mathcal{D}}(x) + \mathrm{mw}_{\mathcal{D}}(y)$.
On the other hand, if $(\gamma_{k-1}, \dots, \gamma_1, \gamma_0)$ is not an expansion of y, but of $y - cb^k$, for some $c \in \mathcal{C}$, then $\mathrm{weight}(\gamma_{k-1}, \dots, \gamma_1, \gamma_0) > \mathrm{mw}_{\mathcal{D}}(y) + \delta$ by definition of k, $\mathrm{weight}(\dots, \gamma_{k+1}, \gamma_k) = \mathrm{mw}_{\mathcal{D}}(x + c)$ and $\mathrm{mw}_{\mathcal{D}}(x + c) \ge \mathrm{mw}_{\mathcal{D}}(x) - \delta$ by Proposition 4.26. Altogether we get

$$\begin{aligned}
\mathrm{mw}_{\mathcal{D}}(xb^k + y) &= \mathrm{weight}(\dots, \gamma_1, \gamma_0) \\
&= \mathrm{weight}(\dots, \gamma_k) + \mathrm{weight}(\gamma_{k-1}, \dots, \gamma_0) \\
&> (\mathrm{mw}_{\mathcal{D}}(x) - \delta) + (\mathrm{mw}_{\mathcal{D}}(y) + \delta),
\end{aligned}$$

which is a contradiction. □

Proposition 4.31. *The state w_0 is reachable from any other state in V and $\mathrm{t}(w_0, 0) = w_0$.*

Proof. Let $(\alpha_r, \dots, \alpha_0)$ be the input that brought the automaton to v, i.e. $\mathrm{t}(w_0, \alpha_r \dots \alpha_0) = v$ and $x = \mathrm{value}(\alpha_r, \dots, \alpha_0)$. Now, because of the proposition above, we can find a k_c for every $c \in \mathcal{C}$, such that

$$\forall k > k_c \colon \ \mathrm{mw}_{\mathcal{D}}(xb^k + c) = \mathrm{mw}_{\mathcal{D}}(x) + \mathrm{mw}_{\mathcal{D}}(c).$$

Let $k := \max\{k_c \mid c \in \mathcal{C}\}$, and $w := \mathrm{t}(v, \overbrace{0 \dots 0}^{k}) = \mathrm{t}(w_0, \alpha_r \dots \alpha_0 \overbrace{0 \dots 0}^{k})$ the state after k times reading the digit 0 starting from state v. Then

$$\begin{aligned}
w(c) &= \mathrm{mw}_{\mathcal{D}}(xb^k + c) - \mathrm{mw}_{\mathcal{D}}(xb^k) \\
&= \mathrm{mw}_{\mathcal{D}}(xb^k) + \mathrm{mw}_{\mathcal{D}}(c) - \mathrm{mw}_{\mathcal{D}}(xb^k) \\
&= \mathrm{mw}_{\mathcal{D}}(x) + \mathrm{mw}_{\mathcal{D}}(c) - \mathrm{mw}_{\mathcal{D}}(x) \\
&= \mathrm{mw}_{\mathcal{D}}(c)
\end{aligned}$$

and therefore $w = w_0$ and we just described a way to reach w_0 from an arbitrary state v, which means that w_0 can be reached from any other state.
For the second claim, for $v = \mathrm{t}(w_0, 0)$ and $c \in \mathcal{C}$, consider:

$$\begin{aligned} v(c) &= \min\{w_0(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(c)\} \\ &= \min\{\mathrm{mw}_{\mathcal{C}}(c_L) + \mathrm{weight}(\eta) \mid (c_L, \eta) \in \mathrm{Cand}(c)\} \\ &= \mathrm{mw}_{\mathcal{C}}(c) \end{aligned}$$

Hence $v = \mathrm{t}(w_0, 0) = w_0$ and the proof is complete.

□

Theorem 2. *The Markov chain $(X_n)_{n\in\mathbb{N}_0}$ corresponding to the state graph (V, E) is irreducible and aperiodic. Therefore it has a unique stationary distribution $\pi \colon V \to [0, 1]$.*

Proof. Any state $v \in V$ is communicating with w_0, since V was defined as all the set of all reachable states starting from w_0 and conversely in Proposition 4.31, we showed that w_0 is reachable from any state in V. Therefore the whole graph just consists of the one communicating class containing w_0 which means it is irreducible.
Since the loop $(w_0, w_0, 0)$ is always in E, w_0 is aperiodic and therefore the whole Markov chain is aperiodic.
The rest follows by Theorem 1.3. □

4.2.3 Asymptotic minimal joint density

Now that we established the existence and uniqueness of the stationary distribution of the Markov chain representing our problem, we describe how to compute it and how to subsequently compute the asymptotic minimal joint density of $\mathcal{D}$-expansions of inputs $z \in \mathcal{R}^d$. In the following, let $\mathcal{D}$ be a suitable digit set for the given base b, (V, E) the state graph corresponding to all possible states $V = \{w_0, \ldots, w_{m-1}\}$ during a run of the minimal weight algorithm and $(X_n)_{n\in\mathbb{N}}$ the associated Markov chain with transition matrix $P \in [0, 1]^{m\times m}$ and initial distribution $\lambda = (1, 0, \ldots, 0)$.
We already established that the stationary distribution $(X_n)_{n\in\mathbb{N}}$ exists and is unique. We write it as a row vector $\pi = (\pi_0, \ldots, \pi_{m-1})$. It satisfies $\pi = \pi P$, $\pi_i \geq 0$ and

$$\sum_{i=0}^{m-1} \pi_i = ||\pi||_1 = 1 \tag{12}$$

in other words it is the left eigenvector to the eigenvalue 1 of P, or the solution to the system

$$x(P - I) = 0, \tag{13}$$

scaled to satisfy (12). Solving (13) presents the computational challenge that—unless done numerically—ultimately limits the applicability of our approach in cases with very large digit sets or high choices of d (see Section 4.4). Once the stationary distribution π is computed however, it leads us directly to the asymptotic minimal joint density as shown in the following theorem.

Theorem 3. *The asymptotic minimal joint density can be computed as*

$$\mathrm{amjd}_{\mathcal{D}} = \sum_{w_i \in V} \sum_{\varepsilon \in \mathcal{D}_{in}} \frac{1}{|\mathcal{D}_{in}|} \cdot \pi_i \cdot \mathrm{wc}(w_i, \varepsilon).$$

Proof. Let $\pi^{(n)}$ be the probability distribution of X_n, which is

$$\begin{aligned}\pi^{(0)} &= \lambda \\ \pi^{(n)} &= \pi^{(n-1)} \cdot P = \lambda \cdot P^n.\end{aligned}$$

Now we define the *average weight change* at step n:

$$\mathrm{awc}(n) = \sum_{w_i \in V} \sum_{\varepsilon \in \mathcal{D}_{\mathrm{in}}} \frac{1}{|\mathcal{D}_{\mathrm{in}}|} \cdot \pi_i^{(n-1)} \cdot \mathrm{wc}(w_i, \varepsilon)$$

Upon closer inspection we see that there is a connection between $\mathrm{awc}(n)$ and $\mathrm{majw}_{\mathcal{D}}(n)$, which is

$$\mathrm{majw}_{\mathcal{D}}(n) = \sum_{j=1}^{n} \mathrm{awc}(j).$$

Now if we divide by n and let $n \to \infty$, we get

$$\begin{aligned}\lim_{n\to\infty} \frac{1}{n} \mathrm{majw}_{\mathcal{D}}(n) &= \lim_{n\to\infty} \frac{1}{n} \sum_{j=1}^{n} \mathrm{awc}(j) \\ \mathrm{amjd}_{\mathcal{D}} &= \lim_{n\to\infty} \frac{1}{n} \sum_{j=1}^{n} \mathrm{awc}(j)\end{aligned}$$

and since

$$\lim_{n\to\infty} \mathrm{awc}(n) = \sum_{w_i \in V} \sum_{\varepsilon \in \mathcal{D}_{\mathrm{in}}} \frac{1}{|\mathcal{D}_{\mathrm{in}}|} \cdot \pi_i \cdot \mathrm{wc}(w_i, \varepsilon) \quad \text{and}$$

$$\lim_{n\to\infty} x_n = x \quad \Rightarrow \quad \lim_{n\to\infty} \frac{1}{n} \sum_{j=0}^{n} x_j = x,$$

we get

$$\mathrm{amjd}_{\mathcal{D}} = \sum_{w_i \in V} \sum_{\varepsilon \in \mathcal{D}_{\mathrm{in}}} \frac{1}{|\mathcal{D}_{\mathrm{in}}|} \cdot \pi_i \cdot \mathrm{wc}(w_i, \varepsilon).$$

□

4.3 Implementation notes

In this section we will take a brief look at the implementation of the concepts introduced in Section 4.2. We will give the complete algorithm for the generation of the finite automaton and discuss some precomputation strategies. We will still be using a somewhat simplified pseudo-code notation, but the relevant pieces of real-world Mathematica source code can be found in Appendix A.

4.3.1 State graph

The main framework for building the state graph is established in Algorithm 16. In it, we make use of the subroutines algCarrySet (covered earlier as Algorithm 11), getInitialState and getNextState.

The real work is done in Algorithm 14, where for a given state $v \in V$ and input $\varepsilon \in \mathcal{D}_{\text{in}}$, we compute the successor $w \in V$ with

$$w(c) = \min\{v(c_L) + \text{weight}(\eta) \mid (c_L, \eta) \in \text{Cand}(\varepsilon + c)\}.$$

At this point we note, that elements in $w \in V$ are implemented as arrays of the form $(w(0), w(c_1), \ldots, w(c_{|\mathcal{C}|}))$, where $\mathcal{C} = \{0, c_1, \ldots, c_{|\mathcal{C}|-1}\}$ and $c_1, \ldots, c_{|\mathcal{C}|-1}$ is an arbitrary but fixed enumeration of the elements of $\mathcal{C}$.
We recall the definition of $\text{Cand}(\varepsilon + c)$ as

$$\text{Cand}(v) = \{(c, \eta) \in \mathcal{C} \times \mathcal{D} \mid b \cdot c + \eta = v, \text{ for some } b \in \mathcal{R}\}$$

and realize that given that our algorithm has to compute this set for every combination of $c \in \mathcal{C}$ and $\varepsilon \in \mathcal{D}_{\text{in}}$ every time we compute a transition, we can boost performance by precomputing it once and then just use a lookup-table. In fact for $(c, \eta) \in \text{Cand}(\varepsilon + c)$ we don't even need the actual value of η, but only its weight. Also in our real world implementation we don't store values for c_L, but just indexes (as in positions in the enumeration $c_1, \ldots, c_{|\mathcal{C}|}$ mentioned above). In the code here however we leave this detail out and hide it behind an associative array notation.

From here on out, let $\text{CandList}(v)$ be a list of elements (c, ω), such that there exists an elements $(c, \eta) \in \text{Cand}(v)$ with $\text{weight}(\eta) = \omega$. We precompute it once for every possible value v of $c + \varepsilon$ for $c \in \mathcal{C}$, $\varepsilon \in \mathcal{D}_{\text{in}}$ via a simple adaptation of Algorithm 12.
When given the input v and ε, Algorithm 14 now loops through all carries $c \in \mathcal{C}$ and looks at $\text{CandList}(c + \varepsilon)$. Out of this list, we generate a list W of all the candidates for the minimal weight, by looking up $v(c_L)$ and computing $v(c_L) + \omega$, for the elements (c, ω) in $\text{CandList}(c + \varepsilon)$. The value for $w(c)$ is then simply found by taking the minimum of W.

One remark to be made for Algorithm 14 is that in the Mathematica implementation it runs for all $c \in \mathcal{C}$ independently by taking a lookup-table of the form

$$\{\text{CandList}(\varepsilon), \text{CandList}(\varepsilon + c_1), \ldots, \text{CandList}(\varepsilon + c_{|\mathcal{C}|})\}$$

and performing the operations within the outer for loop on all elements of the table in parallel. This directly transforms the table into the result w.
The routine to compute the initial state w_0 is given in Algorithm 15. It is a direct application of Definition 4.23 of the initial state and Algorithm 14. We have already shown that the algorithm terminates and we note that our way of dealing with the initial state also presents an improvement on the method suggested by the authors of [26], which leads to a slight reduction in states of the resulting automaton.

Finally the process of building the state graph (Algorithm 16) is a simple breadth first search by means of maintaining a list V_u of not yet explored states. We could just

Algorithm 14 algGetNextState: Computes $\mathrm{t}(v, \varepsilon)$ for $v \in V$ and $\varepsilon \in \mathcal{D}_{\text{in}}$

```
Input:   The current state $v$ and an input $\varepsilon \in \mathcal{D}_{\text{in}}$
Output:   The successor state $w$, such that $w = \mathrm{t}(v, \varepsilon)$
  $w(0)$:=0
  for all $c \in \mathcal{C}$ do
    $W := \{\}$
    $C$ :=CandList($\varepsilon + c$)
    for all $(c_L, \omega) \in C$ do
      $W$ :=$W \cup \{v(c_L) + \omega\}$
    end for
    $w(c)$ :=min $W - w(0)$
  end for
```

Algorithm 15 algGetInitialState: Finding the initial state w_0

```
Input:   The carry set $\mathcal{C}$
Output:   The initial state $w_0$ with $w_0(c) = \mathrm{mw}_{\mathcal{D}}(c)\ \forall c \in \mathcal{C}$
  $w := \{\}$
  $v(0) := 0$
  for all $c \in \mathcal{C}$ do
    $v(c) := \infty$
  end for
  while $w \neq v$ do
    $w := v$
    $v :=$ getNextState$(v, 0)$
  end while
  $w_0 := v$
```

as easily use a depth first search approach without any changes in the result except for the sequence in which states and edges are discovered (which does of course not matter for the resulting graph since its states and edges both form sets). A subtlety in terms of performance considerations hides behind the check $w \notin V$. This step is necessary, to differentiate between states that we come across for the first time and states that are just re-visited (in which case they don't have to be explored again, but only the new edge on which we found them needs to be recorded). For cases where the number of states in the automaton gets sufficiently large, this check which corresponds to a search for w in the ever growing set V, gets slower as V grows. So it makes sense to optimize the representation of states and the way they are stored with this problem in mind. We solved this problem by using functionality native to our programming environment.

Algorithm 16 algStateGraph: Generating the state graph (V, E)

```
Input:   The base $b$ and digit set $\mathcal{D}$
Output:   The state graph $(V, E)$
  $\mathcal{C} := \text{algCarrySet}(\mathcal{D}, b)$
  $w_0 := \text{getInitialState}()$
  $V := \{w_0\}$
  $V_u := \{w_0\}$
  $E := \{\}$
  while $V_u \neq \{\}$ do
    $v := \text{POP}(V_u)$
    for all $\varepsilon \in \mathcal{D}_{\text{in}}$ do
      $w := \text{getNextState}(v, \varepsilon)$
      if $w \notin V$ then
        PUSH($V$,$w$)
        PUSH($V_u$,$w$)
      end if
    end for
    PUSH($E$,$(v, w, \text{wc}(v, w))$)
  end while
```

Example 4.32. We will show how the algorithm works at the example of the state machine for the case $b = 2$, $\mathcal{D} = 0, \pm 1$ and $d = 1$. As shown earlier, the carry set is $\mathcal{C} = \{0, 1\}$ and therefore our states can be represented by two-dimensional vectors $w = (w(0), w(1))$ with $w(0) = 0$ at all times. In Table 7, we show our computational progress. To find the initial state we start with $w = (0, \infty)$ and repeatedly read inputs $\varepsilon = 0$ until we find a loop $w_0 = \text{getNextState}(w_0, 0)$. The two runs (for the two elements of $\mathcal{C}$) of the for-loop in getNextState are represented by two-fold subdivision of the lines for any given w. Note that for $\varepsilon + c = 1$, we have $\text{CandList}(1) = \{(1, 1), (0, 1)\}$, because $\text{Cand}(1) = \{(1, -1), (0, 1)\}$ but in contrast to before, we are now only interested in the weight of the digit written.

$\boldsymbol{w}$	$\boldsymbol{\varepsilon}$	$\boldsymbol{c}$	$\boldsymbol{\varepsilon+c}$	**CandList**	$\mathbf{t}(\boldsymbol{w},\boldsymbol{\varepsilon})$	$\mathbf{wc}(\boldsymbol{w},\boldsymbol{\varepsilon})$	Remark
$(0,\infty)$	0	0	0	$\{(0,0)\}$	$(0,1)$	0	
		1	1	$\{(1,1),(0,1)\}$			
$(0,1)$	0	0	0	$\{(0,0)\}$	$(0,1)$	0	w_0 found
		1	1	$\{(1,1),(0,1)\}$			
$(0,1)$	1	0	1	$\{(1,1),(0,1)\}$	$(0,0)$	1	normalized by -1
		1	2	$\{(1,0)\}$			
$(0,0)$	0	0	0	$\{(0,0)\}$	$(0,1)$	0	
		1	1	$\{(1,1),(0,1)\}$			
$(0,0)$	1	0	1	$\{(1,1),(0,1)\}$	$(0,-1)$	1	normalized by -1
		1	2	$\{(1,0)\}$			
$(0,-1)$	0	0	0	$\{(0,0)\}$	$(0,0)$	0	
		1	1	$\{(1,1),(0,1)\}$			
$(0,-1)$	1	0	1	$\{(1,1),(0,1)\}$	$(0,-1)$	0	
		1	2	$\{(1,0)\}$			

Table 7: Computing the state graph for $b=2$, $\mathcal{D}=0,\pm1$ and $d=1$

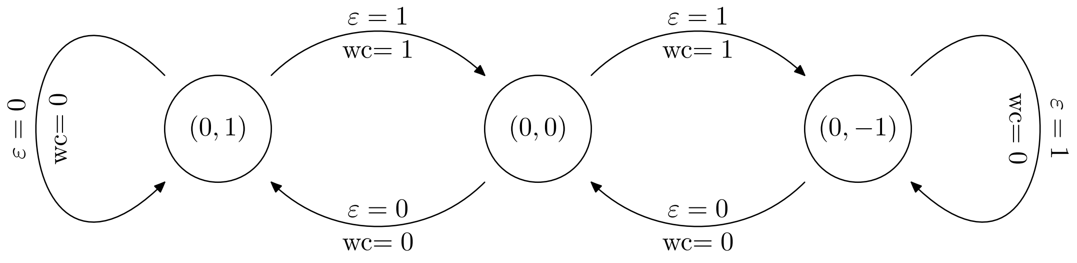

Figure 3: The graph $G=(V,E)$ for $b=2$, $\mathcal{D}=0,\pm1$ and $d=1$

4.3.2 Computation of the asymptotic minimal joint density

After the state graph has been generated, we use the information contained in its edges to compute the asymptotic minimal joint density. The theory is established in Theorem 3 and Algorithm 17 translates it into code. Note that the sum in the computation of $\mathrm{amjd}_{\mathcal{D}}$ goes over all the edges in E, which is equivalent to summing over all states $v\in V$ and input digits $\varepsilon\in\mathcal{D}_{\mathrm{in}}$. The transition matrix P for our Markov chain is generated by going through the list of edges and adding $\frac{1}{|\mathcal{D}|}$ at position i,j for an edge (v_i,v_j,ρ). The weight of the edge is not relevant for P, but double edges have to be accounted for. P is of dimension $m\times m$, where $m=|V|$ is the number of states in the graph.

The statement computeStationary(P) refers to finding a vector π satisfying $\pi P=P$ (i.e. the left eigenvector to the eigenvalue 1) which can be done by non-trivially solving the linear system

$$\pi(P-I)=0.$$

Our result about the uniqueness of the stationary distribution of the Markov chains we

Algorithm 17 algcomputeAMJD: Computing the asymptotic minimal joint density

```
Input:  The set of edges $E$ and the number of states $m$
Output:  The asymptotic minimal joint density $\mathrm{amjd}_{\mathcal{D}}$
  $p_{ij} := \frac{1}{|\mathcal{D}_{\mathrm{in}}|} \left|\{(v, w, \rho) \in E \mid v = v_i,\, w = v_j\}\right| \quad (0 \le i, j < m)$
  $P := (p_{ij})_{0 \le i,j < m}$
  $\pi :=$ computeStationary($P$)
  $\pi := \frac{\pi}{||\pi||_1}$
  $\mathrm{amjd}_{\mathcal{D}} := \sum_{(v_i, v_j, \rho) \in E} \frac{1}{|\mathcal{D}_{\mathrm{in}}|} \cdot \pi_i \cdot \rho$
```

are studying, tells us that the solution space is one-dimensional. In our computational experiments[11], we were able to find an exact rational solution with standard methods of linear algebra for values of m up to around 11000.

For greater m, we can only solve our problem numerically, using iterative methods to find a good approximation for the stationary distribution π. One such method is simply computing the probability distribution of the Markov chain after n steps for large values of n:

$$\begin{aligned} &x^{(0)} \in \mathbb{R}^m \\ &x^{(n+1)} = x^{(n)} P = x^{(0)} P^n \end{aligned}$$

This amounts to a simple power iteration to find the eigenvector π for the largest eigenvalue in magnitude. The properties of stochastic matrices like P dictate this to be $\lambda_1 = 1$ and the Perron-Frobenius theorem tells us that all other eigenvalues are smaller in magnitude. The convergence rate of the power iteration is $O(|\lambda_2|^n)$, where λ_2 is the second largest eigenvalue in magnitude. We can use the Rayleigh quotient

$$R_P(x) = \frac{x^T P x}{x^T \cdot x}$$

to get an idea about the error of our approximation.

The value of an approximated numerical solution to our problem however is to be questioned. Our computation of the asymptotic joint weight for different digit sets, can be seen as an automated proof and unless we can compute it as an exact rational expression, we are not proving anything, but rather just giving a ballpark number (to whatever precision it may be) for orientation so to speak.

In certain cases, we manage to use the iterative solution as a stepping stone for finding the exact solution in a second step. The idea is to first compute an approximation $\tilde{\pi}$ for our stationary distribution as described above and in the second step converting $\tilde{\pi}$ to a vector

[11]Computations carried out on a 32-bit Unix machine with a 2 GHz processor, 2 GB of main memory running Mathematica 6.0

of rationals, by replacing every entry $\tilde{\pi}_i$ by a rational $\bar{\pi}_i = \frac{a_i}{b_i}$, where $\frac{a_i}{b_i}$ is the fraction with the smallest denominator b_i in a certain neighborhood δ_r of $\tilde{\pi}_i$. Setting $\bar{\pi} = (\bar{\pi}_i)$, we can then check if we found the correct solution, by verifying $\bar{\pi} = \bar{\pi} \cdot P$.

As m gets larger for the more involved cases, the entries of π get smaller and their denominators get bigger. This has two consequences: Firstly, we need to decrease δ_r, in order to only have rationals with even bigger denominators within $(\tilde{\pi}_i - \delta_r, \tilde{\pi}_i + \delta_r,)$ and secondly it becomes more and more computationally challenging to compute $\bar{\pi} \cdot P$ when checking for correctness of the rationalization.

It takes some experimentation to find the appropriate precision and number of iterations for the first step and a good value for δ_r, but with this method we are definitely able to compute exact solutions for some cases where explicitly solving the linear system is not an option.

4.4 Results

The results of our computations shall be divided into two sections, for the two different bases 2 and τ. We will give exact values for the the asymptotic minimal joint density of every case where the computation of (V, E) and subsequently amjd were computationally feasible with the hardware at hand.

4.4.1 Base 2

The results for base $b = 2$ and digit sets $\mathcal{D} = \{0, \pm 1, \pm 3, \ldots, \pm(2h+1)\}$ are shown in Table 8. For $d = 1$ we are actually capable of computing results for higher values of h than shown in the table. For $d = 2$, we run into computational limitations at $h = 2$ and can not directly find an exact solution to the linear system at hand. This is one of the cases, where we succeed in finding an exact rational solution by first iteratively approximating the stationary distribution and then rationalizing the result.

For $d \in \{3, 4, 5\}$, the computation was completed only for $h = 0$ and exact results were achieved in all cases. For greater h, the graph becomes to large immediately because of the exponential growth with d.

When we compare our results for base $b = 2$ with [26], we see a slight difference in the number of states $|V|$ for most cases. This is most likely due to differences in the carry sets as noted earlier. The results for the densities are the same. In some of the results in their paper it is unclear whether they are exact or approximated. For $d = 2$ and $h \in \{3, 4, 5\}$, the results for the density are only shown in decimal with a precision of 10^{-4} and we don't know if this is just the product of an approximation, the actual exact value or just a rounded version of the exact value for better readability in the paper. Since we were unable to find exact values in precisely these cases, we can not comment. The result given for $d = 3$ and $h = 1$ seems to be exact, since in this case the actual rational expression is given as $\frac{20372513}{49809043}$. While our algorithm—given enough time and memory—is perfectly capable of building the state graph for that case, it is very questionable that we could devise an exact

d	h	$\|\mathcal{C}\|$	$\|V\|$	amjd
1	0	2	3	1/3
	1	6	23	1/4
	2	10	46	2/9
	3	14	83	1/5
	4	18	108	4/21
	5	22	144	2/11
2	0	4	11	1/2
	1	36	2074	281/786
	2	100	14920	1496396/4826995[a]
3	0	8	81	23/39
4	0	16	681	115/179
5	0	32	5921	4279/6327[a]

[a]Stationary distribution approximated - exact solution found by rationalization.

Table 8: Number of states and amjd for $b = 2$, $\mathcal{D} = \{0, \pm 1, \dots, \pm(2h+1)\}^d$

solution for a graph with more than 1 million nodes by means of the techniques presented in this document.

4.4.2 Base τ

With the results of our analysis for base $b = \tau$, we break new ground in terms of the application of this automated minimal weight computation approach. The values which we compare our results against come from the expansions studied in Chapter 2.

Table 9 shows the number of states $|V|$ and the asymptotic minimal joint density for several instances of our problem. Note that due to the different carry sets for different values of the curve parameter $\mu \in \{\pm 1\}$, the state graph also looks different for those two cases. Our table reflects that by showing the size of the carry set and the state graph for $\mu = 1$ and $\mu = -1$ separately. For the cases we computed, the value of amjd was invariant with respect to the choice of μ. This is by no means a proven statement for general $\mathcal{D}$ and d, but our results would certainly lead to a suspicion in that direction. A consequence would be that in our algorithm, for any given value of w, one would always chose the value of μ yielding the smaller carry set and subsequently the smaller state graph.

These choices definitely do matter in terms of computation time, as can be seen in the case $d = 2$, $w = 2$, where the number of states for $\mu = 1$ is almost five times higher then the value for $\mu = -1$.

Theorem 4. *The asymptotic minimal density for base τ integer expansions using the digit set $\mathcal{D} = \mathrm{MNR}(w)$ is given by $\frac{28}{141}$ for $w = 4$ and $\frac{30}{181}$ for $w = 5$. The asymptotic minimal joint density for base τ expansions of pairs of integers using the digit set $\mathcal{D} = \mathrm{MNR}(2)^2 =$*

d	w	$\mu = 1$		$\mu = -1$		**amjd**
		$\lvert\mathcal{C}\rvert$	$\lvert\boldsymbol{V}\rvert$	$\lvert\mathcal{C}\rvert$	$\lvert\boldsymbol{V}\rvert$	
1	2	12	54	8	27	1/3
	3	27	324	28	336	1/4
	4	85	3746	75	3202	28/141
	5	159	9065	178	10404	30/181
2	2	144	730121	64	151593	~ 0.4649[a]

[a]Exact value given in Theorem 4

Table 9: Number of states and amjd for $b = \tau$, $\mathcal{D} = \mathrm{MNR}(w)^d$

$\{0, \pm 1\}^2$ *is given by*

$$\frac{1448604769522580699609705328661062532744479345709762207494957917971}{3115686690556104018109087307773736176521524892246828413592245388950} \approx 0.4649.$$

Remark 4.33. In [10] it is shown that it is impossible to devise an online algorithm that generates optimal expansions achieving the asymptotic weights given in Theorem 4.

A Source code

The algorithms covered in Chapter 4 were all implemented in Mathematica[12]. In the program code, τ is written as `tau` and treated as a symbolic quantity by Mathematica. Its arithmetic interpretation is established through the implementation of division by τ in the function `tauDiv[]` and the Norm function in $\mathbb{Z}[\tau]$, written as `Ntau[]`. The curve parameter $\mu \in \{\pm 1\}$ is a global variable and written as `mu`.

```
mu = 1;
(* mu = -1; *)

tauDiv[{a1_Integer, a0_Integer}] := ({-a0/2, a1 + mu*a0/2});

tauDiv[z_] := (tauDiv[{Coefficient[z, tau, 1],Coefficient[z, tau,0]}]
              /. {a1_,a0_}-> a1*tau + a0);

Ntau[{b_, a_}] := a^2 + mu*a*b + 2*b^2;

Ntau[z_] :=  Ntau[{Coefficient[z, tau, 1], Coefficient[z, tau, 0]}]
```

[12]Version 6.0, see `www.wolfram.com` for details

A.1 Building the state machine

The following is the Mathematica-Module we implemented for computing the state machine for given dimension `d`, digitset `Ds` and base `base`.

```
genStateMachine[d_, Ds_, base_] := Module[
  {
   divVec, checkDivByBase,
   getNextState, normalize, getMStateID, getStateID, getState, explore,
   getInitialState, computeCarrylookup,

   carrylookup,DsIn = {0, 1}, Cs, csvecs,
   V = {},           (* vertices / states *)
   numstates,
   E = {},           (* edges / state-transitions *)
   Vu,               (* states yet to be explored *)
   newstate, curstate
   },

  (*******  Helper functions *******************************************)

  checkDivByBase[val_] :=
       If[And @@ EvenQ[Coefficient[val, tau, 0]], True,False];
  divVec[val_, 2] := val/2;
  divVec[val_, tau] := tauDiv /@ val;
  (* The norm function *)
  Ntau[{b_, a_}] := a^2 + mu*a*b + 2*b^2;
  Ntau[z_] :=
  Ntau[{Coefficient[z, tau, 1], Coefficient[z, tau, 0]}]

  (*******  SUB-MODULE: computeCarrylookup ****************************)
  (* This sets up a table that, given an input-vector in and a        *)
  (* carry-vector C_R holds a list of pairs {IndexOf(C_L),weight(out)} *)
  (* corresponding to all possible recodings:                         *)
  (*    base*C_L + out =  in + C_R                                    *)

  computeCarrylookup[] := Module[{},
     carrylookup = Map[
       Function[in,
        Map[
         Function[carry,

          Map[Function[out, {carry + in - out, colweight[out]}],
           Tuples[Ds, d]]
          ], csvecs]]
       , Tuples[DsIn, d]];
     carrylookup =
      Map[Function[list, Select[list, checkDivByBase[ #[[1]]] &]],
       carrylookup, {2}];
     carrylookup =
      Map[{divVec[#[[1]], base], #[[2]]} &, carrylookup, {3}];
     carrylookup =
      Map[{First[Position[csvecs, #[[1]]]], #[[2]]} &,
       carrylookup, {3}];
     ];

  (*******  SUB-MODULE: getNextState ***********************************)

  getNextState[curstate_, input_] := Module[
     {nextstate },
     nextstate =
      carrylookup[[ Position[Tuples[DsIn, d], input][[1, 1]] ]];
     nextstate =
      Map[curstate[[ #[[1]] + 2]] + #[[2]] &, nextstate, {2}];
     nextstate = Min /@ nextstate;
     nextstate = Join[{"ID", "hash"}, nextstate];
     Return[nextstate];
     ];

  (*******  SUB-MODULE: normalize  ************************************)
  (*   subtracts entry for carry 0 from all weights                   *)
  (*   this gives the representative for this state                   *)
  (*   additionally we also compute and store the hash                *)

  normalize[state_] := Module[{res, k},
     k = state[[3]];
     If[k == \[Infinity],
      Print["Error in normalize: \[Infinity] on 0-carry"];
      Abort[];
      ];
     res = Delete[state, {{1}, {2}}] - k; (* normalizing *)

     res = Join[{"ID", Plus @@ res}, res]; (* add hash info *)
     Return[res];
     ];

  (*******  SUB-MODULE: getMStateID ***********************************)
```

```
(*   assigns a sequential ID to new states,                          *)
(*   adds them to V and just returns the ID on                       *)
(*   subsequent calls with same argument                             *)

getMStateID[state_] := Module[{newID},
   newID = ++numstates;
   V = Append[V, Prepend[Rest[state], newID]];
   getMStateID[state] = newID;
   ];

(*******  SUB-Function: getState ************************************)

getState[stateID_] := Return[V[[stateID]]];

(*******  SUB-Function: getInitialState ********************)

getInitialState[] := Module[{ st, stold = {}},
   st =
    Join[{1, \[Infinity], 0},
     Table[\[Infinity], {(Length[Cs]^d) - 1}]];
   While[st != stold,
    stold = st;
    st = getNextState[st, Table[0, {d}]];
    ];
   st[[2]] = Plus @@ st[[3 ;; -1]]; (* hash *)
   Return[st];
   ];

(*******  SUB-MODULE: explore  **************************************)
(* This function returns all IDs of new states that are              *)
(* found as direct sucessors of startstate                           *)

explore[startStateID_] := Module[
   {newstateIDs = {}, startState,
    nextState, numstates0, weightchange, nextStateID, digit,
    allinputs},
   startState = getState[startStateID];
   allinputs = Tuples[DsIn, d];
   Do[ (* for digit in all possible digit vectors \[Element] Ds^d *)
    numstates0 = numstates;
    digit = allinputs[[digitindex]];
    nextState = getNextState[startState, digit];
    weightchange = nextState[[3]];
    nextState = normalize[nextState];
    nextStateID = getMStateID[nextState];
    If[nextStateID > numstates0,
     newstateIDs = Append[newstateIDs, nextStateID]];
    E = Append[E, {startStateID, nextStateID, weightchange}];
    , {digitindex, 1, Length[allinputs]}
    ];
   Return[newstateIDs];
   ];


(******* Main Code ***************************************************)

Cs = Sort[genCarrySet[Ds, base], Ntau[#1] < Ntau[#2] &];
csvecs = Tuples[Cs, d];
computeCarrylookup[];
numstates = 0;
newstate = getInitialState[];
getMStateID[newstate]; (* adds it to V and indexes it *)

Vu = {1};

While[Vu != {},
  Vu = Flatten[Map[explore, Vu]];
  ];
Return[{V, E}];
];
```

A.2 Computing weight

Once the state machine in form of the graph (V, E) has been computed, we can compute the asymptotic minimal joint density. As a first step the helper routine **getSysMat[]** generates the (sparse) transition Matrix. Note that the Tally[]-Function helps us deal with double edges. for (V, E).

```
getSysMat[E_, d_] := Module[
  {digitProb, EdgesCount},
  digitProb = 1/(2^d);
  EdgesCount = Tally[E[[All, {1, 2}]]];
  Return[
   SparseArray[{#[[1, 1]], #[[1, 2]]} -> #[[2]]*digitProb & /@
     EdgesCount]
   ];
  ];
```

Next we show the Function to directly compute an exact solution for the stationary distribution of the Markov chain by solving the linear system $x(P - I) = 0$. The `NullSpace[]`-Routine runs into trouble if the matrix is too big. At some point during the computations, the matrix is transformed from a sparse to a normal matrix which is a problem and we get an error:

```
parseArray::ntb: "Cannot convert the sparse array SparseArray[...] to an
ordinary array because the 22980437649 elements required exceeds the
current size limit."
```

For Graphs with less than 10000 vertices, the code worked fine in our system environment. We can blindly take the first part of the result from the call to `NullSpace[]`, because in the theoretic part we already established that the solution is one-dimensional. Another noteworthy comment is that since `NullSpace`$[A]$ gives a basis for the solution of $Ax = 0$ and since we need a solution for $xP = x$, we call `NullSpace[`$P^T - I$`]`.

```
 computeStationaryExact[P_] := Module[
   {Eye, timing, statDist},
   Eye = SparseArray[{{i_, i_} -> 1}, {Length[P], Length[P]}];
   deb[5, "Computing marginal distribution..."];
   timing = Timing[
      statDist = NullSpace[(P\[ConjugateTranspose] - Eye)][[1]];
      ][[1]];
   deb[5, "Finished computing marginal distribution (", timing, " s)"];
   statDist = statDist/(Plus @@ statDist);
   Return[statDist];
   ];
```

Computing the asymptotic minimal joint density is easily done by summing over all edges as described in the theoretical part. Note that the function below provides the possibility to supply the stationary distribution directly when calling it. We will need this later. For now, it suffices to know, that when the optional third argument `sD` is left out, the stationary distribution is computed via `computeStationaryExact[]`.

```
computeAMJD[E_, d_, sD_: Null] := Module[
   {P, statDist, amjd, DsIn = {0, 1}},
   If[sD === Null,
    P = getSysMat[E, d];
    statDist = computeStationaryExact[P],
    statDist = sD; (* else *)
    ];
   amjd = (Plus @@ (statDist[[ #[[1]] ]]*#[[3]] & /@ E))/(Length[DsIn]^d);
   Return[amjd];
   ];
```

Next we take a look at the source code for the numeric approximation of the stationary distribution for the transition matrix. There are two mechanisms to control the number of iterations. One is a hard limit for the number of iterations, the other is the specification of the floating point precision `precisionIter`. By specifying this as the optional second argument to `N[]` when generating the start vector `x1`, all computations are starting off with this precision. Anytime a numeric operation causes a loss of precision, Mathematica just drops the additional, numerically dirty digits. Such numerically problematic operations are subtractions and divisions. This leads to the (intended) behavior, that the variable `error` drops to zero at a certain point, meaning the quantity, by which our approximation fails to fulfill $xP = x$ is no longer measurable in the given precision `precisionIter` and the iteration stops. Note that the result suffers only a negligible loss of precision (one digit) in the last step, when it is normed to sum to one.

```
computeStationaryIter[P_, precisionIter_, maxIt_] := Module[
   {pnorm, numit = 0, x1, x2, error, xact, xactOK = True},
   pnorm[x_] := x/Plus @@ x;

   x1 = Table[N[RandomInteger[{0, 20}], precisionIter], {Length[P]}];

   While[True,
    x2 = x1.P;
    error = Norm[pnorm[x1] - pnorm[x2]];
    x1 = x2;

    If[error == 0,
     deb[5, "Error no longer numerically significant after ",
         numit,"steps"];
     Break[];
     ];

    If[numit == maxIt,
     Print["Iteration Limit ", maxIt, " reached"];
     Break[];
     ];
    numit++;
    ];
   Return[pnorm[x1]];
   ];
```

The iterative solution for the stationary distribution of P, can be used to compute an approximation for amjd:

```
  (* Edges either computed earlier or loaded from file *)
  P = getSysMat[Edges, d];
  SDiter = computeStationaryIter[P, 25, 200];
  amjdapprox = computeAMJD[Edges, d, SDiter];
```

The more important role of the iterative solution is that of a base point for an attempt to find an exact solution by rationalization. The function implementing this is `computeStationaryRational[]`. The actual rationalization is done by the Mathematica-native function `Rationalize[]` and we just follow that up with doing several validity checks on the result.

```
computeStationaryRational[P_, SDiter_, precisionRationalize_] :=
  Module[
   {SDratio, xactOK = True},

   SDratio = Rationalize[SDiter, 10^(-precisionRationalize)];

   If[Or @@ (MatchQ[#, _Real] & /@ SDratio),
    xactOK = False;
    deb[5, "No rationalisation in desired bounds found"];
    Return[False];
    ];
   If[Not[And @@ (# >= 0 & /@ SDratio)],
    xactOK = False;
    deb[4, "Result of rationalisation not positive"]];

   If[Plus @@ SDratio != 1,
    xactOK = False;
    deb[4, "Result of rationalisation is not a density (sum=",
     N[Plus @@ SDratio, precisionRationalize], ")"]];

   If[SDratio.P != SDratio,
    xactOK = False;
    deb[4, "Result of rationalisation is not stationary"];
    ];
   If[xactOK,
    deb[6, "Found exact solution by rationalisation!"],
    deb[6, "No valid rationalisation found."];
    ];
   Return[SDratio]
  ];
```

References

[1] R. Avanzi, *A Note on the Signed Sliding Window Integer Recoding and a Left-to-Right Analogue*, Selected Areas in Cryptography: 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004, Revised Selected Papers, Lecture Notes in Comput. Sci., vol. 3357, Springer-Verlag, Berlin, 2004, pp. 130–143.

[2] R. M. Avanzi, C. Heuberger, and H. Prodinger, *Scalar multiplication on koblitz curves using the frobenius endomorphism and its combination with point halving: Extensions and mathematical analysis*, Algorithmica **46** (2006), 249–270.

[3] ______, *Redundant τ-adic expansions I: non-adjacent digit sets and their applications to scalar multiplication*, Des. Codes Cryptography **58** (2011), 173–202.

[4] P. Balasubramaniam and E. Karthikeyan, *Elliptic curve scalar multiplication algorithm using complementary recoding*, Appl. Math. Comput. **190** (2007), no. 1, 51–56.

[5] C. Chang, Y. Kuo, and C. Lin, *Fast algorithms for common-multiplicand multiplication and exponentiation by performing complements*, Proceedings of the 17th International Conference on Advanced Information Networking and Applications (Washington, DC, USA), AINA '03, IEEE Computer Society, 2003, pp. 807–.

[6] M. Ciet, T. Lange, F. Sica, and J.-J. Quisquater, *Improved algorithms for efficient arithmetic on elliptic curves using fast endomorphisms*, Advances in cryptology — EUROCRYPT 2003. International conference on the theory and applications of cryptographic techniques, Warsaw, Poland, May 4–8, 2003. Proceedings (E. Biham, ed.), vol. 2656, Springer, Berlin, 2003, pp. 388–400.

[7] P. Gallagher and C. Furlani, *FIPS pub 186-3 federal information processing standards publication digital signature standard (DSS)*, 2009.

[8] P. J. Grabner, C. Heuberger, and H. Prodinger, *Distribution results for low-weight binary representations for pairs of integers*, Theoret. Comput. Sci. **319** (2004), 307–331.

[9] P. J. Grabner, C. Heuberger, H. Prodinger, and J. Thuswaldner, *Analysis of linear combination algorithms in cryptography*, ACM Trans. Algorithms **1** (2005), 123–142.

[10] C. Heuberger, *Redundant τ-adic expansions II: Non-optimality and chaotic behaviour*, Mathematics in Computer Science **3** (2010), no. 2, 141–157.

[11] C. Heuberger, R. Katti, H. Prodinger, and X. Ruan, *The alternating greedy expansion and applications to left-to-right algorithms in cryptography*, Theoret. Comput. Sci. **341** (2005), 55–72.

[12] C. Heuberger and J. A. Muir, *Minimal weight and colexicographically minimal integer representations*, J. Math. Cryptol. **1** (2007), 297–328.

[13] C. Heuberger and H. Prodinger, *On minimal expansions in redundant number systems: Algorithms and quantitative analysis*, Computing **66** (2001), 377–393.

[14] ———, *The Hamming weight of the Non-Adjacent-Form under various input statistics*, Period. Math. Hungar. **55** (2007), 81–96.

[15] J. Hoffstein, J.C. Pipher, and J.H. Silverman, *An introduction to mathematical cryptography*, Undergraduate texts in mathematics, Springer, 2008.

[16] D. E. Knuth, *Seminumerical algorithms*, third ed., The Art of Computer Programming, vol. 2, Addison-Wesley, 1998.

[17] N. Koblitz, *Elliptic curve cryptosystems*, Math. Comp. **48** (1987), no. 177, 203–209.

[18] ———, *CM-curves with good cryptographic properties*, Advances in cryptology—CRYPTO '91 (Santa Barbara, CA, 1991), Lecture Notes in Comput. Sci., vol. 576, Springer, Berlin, 1992, pp. 279–287. MR 94e:11134

[19] F. Morain and J. Olivos, *Speeding up the computations on an elliptic curve using addition-subtraction chains*, RAIRO Inform. Théor. Appl. **24** (1990), 531–543. MR 91i:11189

[20] J. A. Muir and D. R. Stinson, *Minimality and other properties of the width-w nonadjacent form*, Math. Comp. **75** (2006), 369–384. MR MR2176404

[21] J. R. Norris, *Markov chains*, Cambridge University Press, Cambridge, 1997.

[22] G. W. Reitwiesner, *Binary arithmetic*, Advances in computers, vol. 1, Academic Press, New York, 1960, pp. 231–308.

[23] J. A. Solinas, *Efficient arithmetic on Koblitz curves*, Des. Codes Cryptogr. **19** (2000), 195–249. MR 2002k:14039

[24] ———, *Low-weight binary representations for pairs of integers*, Tech. Report CORR 2001-41, Centre for Applied Cryptographic Research, University of Waterloo, 2001.

[25] E. Straus, *Addition chains of vectors (Problem 5125)*, Amer. Math. Monthly **71** (1964), 806–808.

[26] V. Suppakitpaisarn, M. Edahiro, and H. Imai, *Optimal average joint hamming weight and minimal weight conversion of d integers*, Cryptology ePrint Archive, Report 2010/300, 2010.