

Automata in SageMath—Combinatorics meets Theoretical Computer Science

Clemens Heuberger¹Daniel Krenn¹Sara Kropf¹¹ *Institut für Mathematik, Alpen-Adria-Universität Klagenfurt**received 13th Jan. 2016, revised 13th Apr. 2016, accepted 2nd May 2016.*

The new finite state machine package in the mathematics software system SageMath is presented and illustrated by many examples. Several combinatorial problems, in particular digit problems, are introduced, modeled by automata and transducers and solved using SageMath.

In particular, we compute the asymptotic Hamming weight of a non-adjacent-form-like digit expansion, which was not known before.

Keywords: Transducer, automaton, non-adjacent form, Hamming weight, tutorial

0 Prelude

0.1 First Act: Digits and Multiplication

Ladies and Gentlemen, let me introduce you to the first guest of this opening ceremony: the equation

$$9000 + 900 + 90 + 9 = 10000 - 1.$$

You might ask now: Why?—Why exactly this?—A small hint: It has to do with multiplication as the title of the first act says. Already guessed?—Yes?—No?—Then, here is the answer.

Suppose that we want to multiply a huge number n by 9999. Suddenly today's guest list becomes clearer. Such a multiplication should not be done according to the schoolbook, but just by subtracting n from $10000n$. Since nowadays such calculations are done by computers, we switch to binary digit expansions at this point. That is our cue to introduce the next guests: the digits 0 and 1. Usually they are used when writing a number in standard binary expansion, for example

$$14 = 1 \cdot 8 + 1 \cdot 4 + 1 \cdot 2 + 0 \cdot 1 = (1110)_2.$$

Tonight they will perform in a special way, but they need help from their digit sibling $\bar{1} = -1$, which brings us to the following riddle:

Write 14 in base 2 with the digits -1 , 0 and 1 such that there are fewer nonzero digits than in the standard binary expansion.

Our digit triplet needs only a moment to prepare a solution. The lights go on again and we immediately see

$$14 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 - 1 \cdot 2 + 0 \cdot 1 = (100\bar{1}0)_2. \quad (1)$$

After thinking about this for a moment, we deduce that it cannot work with only one nonzero, so we are satisfied with this perfect solution. This brings us directly to the special guest of this evening, the non-adjacent form, which will be present in the whole article. It guarantees that no two adjacent digits are nonzero at the same time; and the digit expansion (1) is indeed such a non-adjacent form. A bunch of questions appear:

The authors are supported by the Austrian Science Fund (FWF): P 24644-N26 and by the Karl Popper Kolleg "Modeling–Simulation–Optimization" funded by the Alpen-Adria-Universität Klagenfurt and by the Carinthian Economic Promotion Fund (KWF).

Email-addresses: clemens.heuberger@aau.at, math@danielkrenn.at or daniel.krenn@aau.at, sara.kropf@aau.at

So how do we determine such a digit expansion of a general number? Is there a way to simply get it from the binary expansion? Is this the best possible way to write an integer?

Questions over questions—all of which will be answered after a short break.

0.2 *Second Act: Automata and Other Boxes*

In this second act we will first meet the main characters of this article. The multiplication, expansions and digits from the previous act still play their role, which is to be nice examples. As a small side note, the authors Clemens, Daniel and Sara work with digit expansions a lot in their research, so these examples seem to be a natural choice.

Back to business; there are some presents for each of you—these little boxes over there called automata. You will find them everywhere. Some of these magic boxes can only be opened giving them a special digit expansion (see Section 2.1). And, it becomes better and better. Some of them even speak some words; they are known as transducers and will return, for example, a non-adjacent form for its binary input. The trick behind these magic boxes will be revealed in Section 1; even better you can take a look inside them, even see how they are constructed and create new such boxes in Sections 2, 3 and 4.

0.3 *Third Act: Strange New World*

Welcome back everybody to this third act. The next guest was already present, but hidden behind the scenes. It offers an immense amount of mathematical objects combined with algorithms for working with them. All the automata and transducers mentioned previously are integrated in it. Contributions to its code-base are subjected to a transparent peer-review process. It stands for reproducible results as all parts of its library are tested with each future release. For the last couple of years, more and more researchers are using it. Ladies and Gentlemen, we proudly present the free and open-source mathematics software system **SageMath** [31].

There are great news these days: **SageMath** has now a module for finite state machines, automata, and transducers [17], which was written and is still improved by the three authors of this article. One of the main motivations of using **SageMath** as the home for this finite state machine module was to allow the use of more or less everything in the zoo of mathematical objects available in **SageMath**, in particular during the construction and manipulation of automata and transducers. In general, the use of tools from different areas is one of the big strengths of **SageMath**.

The aim of this piece of work is to demonstrate the functionality of the new finite state machines package in the form of a tutorial. The interplay between automata and transducers on the one hand and, for example, graph theory, linear algebra, and asymptotic analysis on the other hand will be delineated and worked out. A more detailed overview can be found in Section $\frac{1}{2}$.3.

0.4 *Fourth Act: What Next?*

Now, in this final act of the prelude, we make things more concrete. Let us have another look at the binary expansion and the non-adjacent form of integers. Intuitively it is clear that on average one half of the digits in a binary expansion are 1 and, because of the restrictions in non-adjacent forms, these expansions should have less nonzeros. But what is the true value?—Can someone guess this number?—Maybe one third of the digits are nonzeros?—Yes, indeed, this is correct, however, it is not a consequence of having used exactly 3 digits, but a coincidence. So, guess what does it take to calculate these numbers and their corresponding asymptotic expansions?

Typically (weighted) adjacency matrices of the underlying graphs of a transducer and their eigenvalues are used, and indeed **SageMath** can do this out of the box (see the finale on the last couple of pages of this article). Another example is testing whether the non-adjacent form of any integer minimizes the number of nonzero digits. Huh!?—How to do this?—The answer is: Construct a suitable transducer, use the underlying digraph and apply the Bellman–Ford algorithm on it, ... Eureka!

We are already in the final phase of this introduction and there is something missing. So far all these quantities and results were known before. But what about finding new results to open

problems? This brings us to our surprise guest, which is related to the non-adjacent form, but uses digits -2 and 2 as well and is constructed in a special way. Is this a good expansion to use for our multiplication problem mentioned in the first act? How do we compute it, and is it the best possible? What about its average number of nonzeros? Many questions here, but all will be answered illustratively in Section 4.

There are only a few things left to say. Thank you for reading these first four acts and stay with us to get to know the really cool, interesting and practical stuff. Enjoy the rest of the evening!

$\frac{1}{2}$ Bonus: Questions to and Answers from the Authors

$\frac{1}{2}.1$ *How Do I Get This Awesome New Finite State Machines Package?*

To keep it short: It is already included in SageMath.⁽ⁱ⁾ If you are using the finite state machines package of SageMath in your own work, please let us know.

$\frac{1}{2}.2$ *Can I Easily Use SageMath?*

Definitely yes, it is very simple to use SageMath. All you need is a modern web-browser, going to <https://cloud.sagemath.com>, create a free account and start playing around in the SageMath-Cloud. Even better: All the code examples of this article are available there.⁽ⁱⁱ⁾

Of course and if you prefer, SageMath can be run locally on your computer as well.

$\frac{1}{2}.3$ *What is This Tutorial about?*

As mentioned in the prelude, the aim of this piece of work is to demonstrate some of the functionality of the new finite state machines package in the form of a tutorial. Detailed documentation of the available methods and their parameters can, as usual, be found in the SageMath documentation.⁽ⁱⁱⁱ⁾ There, further examples are presented as well.

Along the way, a new result on a digit system related to the non-adjacent form [30] is proved. We consider a new digit expansion with base 2 and digit set $\{-2, -1, 0, 1, 2\}$. We compute the expected value of the Hamming weight, i.e., the number of nonzero digits, of this digit expansion of integers less than 2^k . Although the digit set is larger than the one for the non-adjacent form, it turns out that the expected value of this new digit expansion is worse than that of the standard binary expansion and therefore worse than that of the non-adjacent form.

$\frac{1}{2}.4$ *Where Do I Find What?*

While answering this question, we give a brief overview on how to show the result mentioned above. We also demonstrate the enormous advantages of using SageMath for constructing and simulating automata and transducers. We start at the beginning and get to know automata in Section 1. We will see various ways to construct them and use them as building blocks to construct larger and better machines (the whole Section 2). Some first insights in the interplay between the concept of automata and combinatorics are found in Section 2.3.

We then go on to the generation of transducers, which can be done in several ways as well. First, we can simply list all transitions of a finite state machine (cf. Section 3.1). Second, we can use transition functions written in SageMath to construct a transducer (cf. Section 3.3, but we will use it in several other places as well). This is of course possible for any type of finite state machine. Another way is to construct machines by a suitable combination of smaller building blocks: We manipulate and combine several finite state machines properly (Sections 3.6 and 4.2). A couple of variants and more advanced constructions will also be shown (see, for example, Sections 3.5 and 4.4).

⁽ⁱ⁾ The basic version was included in SageMath 5.13 [32] (for details see the relevant ticket [17] on the SageMath trac server). To work with all features of this tutorial, use SageMath 6.10 or later.

⁽ⁱⁱ⁾ A worksheet with all code examples is available in the SageMathCloud at <https://cloud.sagemath.com/projects/9a0d876c-4515-4a30-968a-96bd139c3c19/files/fsm-in-sage/fsm-in-sage.sagews>

⁽ⁱⁱⁱ⁾ The SageMath documentation of the finite state machine module can be found at http://doc.sagemath.org/html/en/reference/combinat/sage/combinat/finite_state_machine.html.

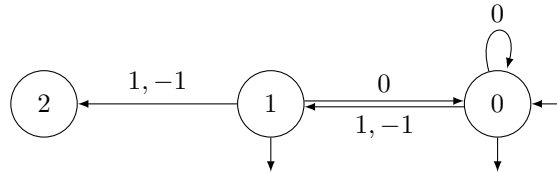


Figure 1: An example of an automaton.

Equipped with the wisdom gained up to this point, we go on to a new challenge. Section 4 is devoted to a new digit expansion related to the non-adjacent form. We want to analyze it and investigate its properties. The main result will be the asymptotic behavior of the average output (cf. Section 4.7). This can be obtained by just one function call. However, behind the scenes, the interplay between automata and transducers on the one side and other areas of mathematics within SageMath on the other side plays an essential role. Linear algebra with adjacency matrices (Section 4.6) and asymptotic analysis will appear.

One final remark before getting started: All of these steps can be computed within SageMath. Thus, we can use everything offered by the mathematical software system and construct powerful finite state machines for the analysis, without shuffling data from one system to another.

1 Using an Automaton Properly

You are already well acquainted with automata and transducers?—Great, skip to Section 2. Otherwise, stay with us, a gentle introduction to these useful machines is provided here.

Figure 1 shows an example of an *automaton*. Obviously, an automaton has little to do with a vending machine which returns a snack after inserting a coin. So what shall we stuff in instead and what bubbles out?

This is easy to explain: The input is a sequence of numbers, letters, etc. The automaton then returns an answer “Yes, I accept this input,” or “No, I do not accept this input.” So apparently there is something going on, and we want to understand: How does the automaton know which answer to give?

Let us say we insert the input sequence $0, -1, 0, 0, 1$ in the automaton of Figure 1. There is one arrow entering this picture; this is at the *initial state* 0 where we start. The automaton reads the first element 0 of the input sequence and then decides to which state it goes next. It chooses the *transition* (i.e. a labeled arc) starting in the *current state* 0 whose label equals the current input 0. This loop leads to state 0 again, which is now our new current state. We proceed as above and read the next element -1 of the sequence. This leads to state 1. The next elements are 0, which leads back to state 0, another 0, which keeps us in state 0, and finally 1, which takes us to state 1.

As there is no more input, the automaton will stop and return an answer: State 1 is a *final state* (in Figure 1 marked by a small outgoing arrow), thus the answer is “Yes, I accept this input.”

If the last current state is not final, then the answer is “No”. Moreover, if there is no valid transition to choose, then the answer is “No” as well. These cases are illustrated by the following examples. Suppose we use the input sequence $1, 1$ and the automaton in Figure 1. The negative answer is returned since we end up in state 2, which is nonfinal. Using the input sequence $1, 1, 0$, the answer is still “No”, but for a different reason. State 2 is reached by the first two 1, and then a 0 is read. There is no transition starting in state 2 with label 0, so the automaton cannot proceed further.

So much for an informal description.^(iv) You may have got the impression that automata are not very entertaining because they can only answer “Yes” or “No”. If so, we want to change this—there exist transducers as well, which provide a more sophisticated output. In principle, a *transducer* is

^(iv) Of course, it is possible to define an automaton formally as a quintuple consisting of a set of states, a set of initial states, a set of final states, an alphabet and a transition function (see Hopcroft, Motwani and Ullman [25] or Sakarovitch [33]). But for this tutorial, it is enough to think of an automaton as a graph with additional labels as in Figure 1.

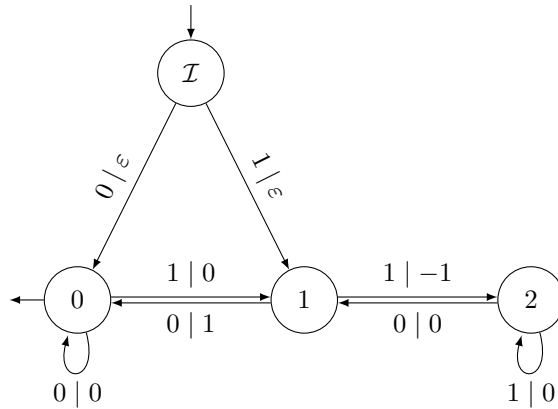


Figure 2: An example of a transducer.

the same as an automaton with an additional second label (the *output label*) for every transition. Figure 2 shows such an animal: For example, the label $1 \mid 0$ means that this transition has an input label 1 (as used by automata when reading the input sequence) and an output label 0. Every time this transition is used, the output 0 is written at the end of the previous outputs. There is one new symbol, namely an output label ε , which stands for the empty word (i.e., nothing is written). Let us illustrate this by an example, too. For the transducer in Figure 2, the input sequence 0, 1, 1, 1, 0, 0 produces “Yes” together with the output sequence 0, -1 , 0, 0, 1 (and note that our first output was an ε).

It is almost time to get our hands dirty, so let us pose a couple of questions: How do we get a transducer like the one in Figure 2? Do we have to construct it with pen and paper? Luckily, the answer is no; SageMath can do this construction for us automatically. We will see this in the next sections.

2 Making Us Comfortable with the Non-Adjacent Form as a Warm-Up

Before we start constructing automata—and later on transducers—let us have a look at the terms used in this section. We start by explaining the classical *non-adjacent form* of an integer, abbreviated as *NAF*. It is a representation with base 2 and digits -1 , 0 and $+1$, such that two adjacent digits are not both nonzero. By setting $\bar{1} = -1$, this means that we forbid all blocks 11 , $\bar{1}1$, $1\bar{1}$ and $\bar{1}\bar{1}$ in the digit expansion. For example, we have

$$14 = 1 \cdot 16 + 0 \cdot 8 + 0 \cdot 4 - 1 \cdot 2 + 0 \cdot 1 = (100\bar{1}0)_2. \quad (2)$$

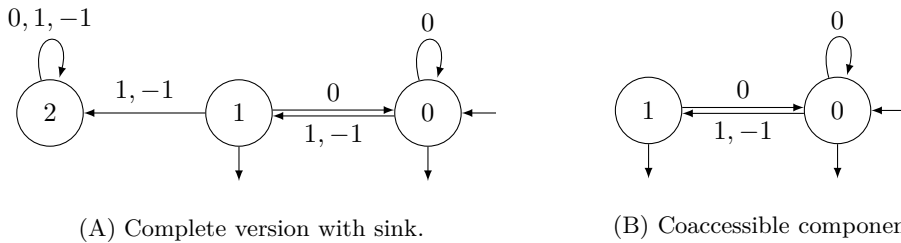
In contrast, $14 = (10\bar{1}10)_2$ is not a non-adjacent form. It can easily be shown that each integer has a unique representation by a NAF, cf. Reitwiesner [30].

Usually, digit expansions are written from the most significant digit (on the left-hand side) to the least significant digit (on the right-hand side). As we compute digit expansions exactly in the other direction in this article^(v), we also write them in that other direction when used as input or output of finite state machines. Therefore, we have two different notations. For digit expansions, like in $(100\bar{1}0)_2$, we write the least significant digit on the right. For inputs and outputs of finite state machines, like in $[0, -1, 0, 0, 1]$, we write the least significant digit on the left. Consequently, we also speak of trailing zeros if we append zeros after the most significant digit.

2.1 Accepting Everything

It is time to create our first automaton. We have already learned about the non-adjacent form, so let us construct an automaton recognizing such digit expansions. So a better title of this section

^(v) This approach uses information modulo the base. An alternative approach, not adopted here, uses greedy expansions, where digits are computed from the most significant to the least significant one.



(A) Complete version with sink.

(B) Coaccessible component.

Figure 3: The automaton `NAF_language` which accepts non-adjacent forms.

would have been “Accepting Something”.

The language of non-adjacent forms (with possible trailing zeros) can be written as the regular expression

$$(0 + 10 + \bar{1}0)^* (1 + \bar{1} + \varepsilon)$$

meaning: A non-adjacent form starts with 0, 10 or $\bar{1}0$, where “or” is usually denoted by $+$. This part can be repeated arbitrarily often (also zero times), which is denoted by the Kleene star $*$. Up to now, we get a valid non-adjacent form since each nonzero is followed by a zero, but there are some expansions missing. So at the end we might have to add a 1 or $\bar{1}$, or not (denoted by the empty word ε).

Such regular expressions are closely related to automata; even better, there is a direct way to translate them to obtain an automaton which accepts non-adjacent forms. We type^(vi)

```
one = automata.Word([1])
zero = automata.Word([0])
minus_one = automata.Word([-1])
epsilon = automata.EmptyWord(input_alphabet=[-1, 0, 1])
NAF_language = ((zero + one*zero + minus_one*zero).kleene_star() *
                (one + minus_one + epsilon)).minimization()
```

In fact, we have now created many (smaller) automata^(vii) and have joined them to get the automaton `NAF_language`.

For now the result is stored in the variable `NAF_language`. By typing `NAF_language` we get

```
Automaton with 3 states
```

Since a picture says a thousand words, let us draw this automaton^(viii) in Figure 3A. Hey!—State 2 of this automaton is a black hole, a so-called sink: Once an input sequence reaches this state, it is lost forever. Can’t we get rid of such states?—Of course; the command is

```
NAF_language_coaccessible = NAF_language.coaccessible_components()
```

and the result is shown in Figure 3B. More on drawing finite state machines comes later in Section 4.3.

Maybe this is a good point to mention that a couple of commonly used automata and transducers, like `automata.Word`, are already prebuilt in `SageMath` and can be used directly; take a look into the `SageMath` documentation.

So, we have our first own creation of an automaton.—Now we want to see it working. The command

```
NAF_language([0, -1, 0, 0, 1])
```

^(vi) This document was created using `SageTeX`, which comes along with `SageMath`. It allows the following: We type `SageMath` source code directly in the `TeX`-document, this code is then executed by `SageMath` and the corresponding outputs (results) are typeset here.

^(vii) For example, `one = automata.Word([1])` is an automaton which only accepts the input word of the single letter 1 and nothing else. A `*` concatenates two automata. The method `minimization()` returns a minimal version of the given automaton. If you want to know more about how it works, consult the documentation of `SageMath`, for example, by typing `NAF_language.minimization?`.

^(viii) More precisely, Figure 3A shows `NAF_language.relabeled()`; this provides nice labels.

returns `True`, i.e., it accepts the input as it should. In contrast,

```
NAF_language([0, 1, 1, 1])
```

returns `False`, so the input is rejected by the automaton. We lean back and are satisfied for now.

2.2 Accepting Everything a Second Time

Our successful first construction motivates us to proceed. Letting our minds flow, we remember that non-adjacent forms are also characterized by forbidding each of the subwords `11`, `1̄1`, `11̄` and `1̄1̄`. Shouldn't this fact be used somehow?

Looking up some prebuilt automata and ways to transform them, we think that

```
automata.ContainsWord([1, 1], input_alphabet=[-1, 0, 1]).complement()
```

will be helpful. This machine accepts words without the subword `1,1`. As a small explanation, `complement()` constructs an automaton which accepts exactly the input sequences which were rejected by the previous automaton. It seems to be logical to copy and paste this command to create automata dealing with the other forbidden subwords. But what now?—We need some intersection of them.—No problem at all, just use the operator `&` for intersection!

So all together we type

```
NAF_language2 = \
    (automata.ContainsWord([1, 1], input_alphabet=[-1, 0, 1]).complement() &
     automata.ContainsWord([1, -1], input_alphabet=[-1, 0, 1]).complement() &
     automata.ContainsWord([-1, 1], input_alphabet=[-1, 0, 1]).complement() &
     automata.ContainsWord([-1, -1], input_alphabet=[-1, 0, 1]).complement()
    ).minimization()
```

Is this the same as our first automaton `NAF_language`? The name and our ideas for the construction suggest that it is. So let us check this by

```
NAF_language2.is_equivalent(NAF_language)
```

This gives—suspense builds—the result `True`. Puh, what a relief!

Equivalently, by De Morgan's law, we “negate” this construction and forbid all words which contain any of the subwords `11`, `1̄1`, `11̄`, or `1̄1̄`. This is done via

```
NAF_language3 = \
    (automata.ContainsWord([1, 1], input_alphabet=[-1, 0, 1]) +
     automata.ContainsWord([1, -1], input_alphabet=[-1, 0, 1]) +
     automata.ContainsWord([-1, 1], input_alphabet=[-1, 0, 1]) +
     automata.ContainsWord([-1, -1], input_alphabet=[-1, 0, 1])
    ).determinisation().complement().minimization()
```

Again `NAF_language3.is_equivalent(NAF_language)` yields `True`, so we have done everything correctly.

But wait!—What exactly did we do?—What is `determinisation()`? And why do we need it? The answer to the second question is that the result of the operation `+` is not deterministic.^(ix) And `determinisation()` is an algorithm which makes this automaton deterministic. Some fact: For each automaton an equivalent deterministic automaton can be created. But be aware that the determinisation of an automaton can increase the number of states exponentially. That is the reason why it is not done automatically, and you have to call it yourself.

2.3 It's Counting Time

As interesting as the previous tasks were, we want more mathematics and make a small excursion now: It is counting time! We ask: How many NAFs of a given length are there?

Once we have an automaton accepting the NAF, SageMath should be able to tell us the answer. And indeed, this can be achieved by typing

^(ix) A nondeterministic automaton can have more than one possible path for an input sequence. An input is accepted if there is at least one path with this input sequence as labels leading to a final state.

```
var('n')
N = NAF_language.number_of_words(n)
```

The answer is

$$\frac{4}{3} \cdot 2^n - \frac{1}{3} (-1)^n,$$

which seems like magic.—How does this trick work? This will be revealed in Section 4.6.

This is the end of our section on automata. We come back to automata a bit later in Section 3.4 because there is a further possibility creating a NAF-automaton: We use the output of a transducer which transforms any digit expansion with digits $-1, 0$ and 1 into a non-adjacent form and make an automaton out of it.

So, what comes next?—We already have methods that tell us whether some input is a non-adjacent form and how many there are, but we also want to calculate these digit expansions. This brings us to the transducers of the following sections.

3 Three Kinds of Calculating the Non-Adjacent Form

We already heard an introduction to non-adjacent forms at the beginning of Section 2, and building various automata accepting these digit expansions went quite well. So let us now tackle the question: How do I calculate a non-adjacent form?

3.1 Creating a Transducer from Scratch

In [20, Figure 2], a transducer for converting the binary expansion of an integer n into its non-adjacent form is given. We reproduce it here as Figure 2 and directly translate it into SageMath. We write

```
NAF1 = Transducer([( 'I', 0, 0, None), ( 'I', 1, 1, None),
                    (0, 0, 0, 0), (0, 1, 1, 0),
                    (1, 0, 0, 1), (1, 2, 1, -1),
                    (2, 1, 0, 0), (2, 2, 1, 0)],
                  initial_states=[ 'I'], final_states=[0],
                  input_alphabet=[0, 1])
```

to construct this transducer with states 'I' (the string I), 0, 1 and 2 (the integers 0, 1 and 2, respectively). The list of 4-tuples defines the transitions of the transducer. For example, $(1, 2, 1, -1)$ is a transition from state 1 to state 2 with input 1 and output -1 . Here, input means reading a (more precisely, the next) digit of the binary expansion of n . The output is a digit of the non-adjacent form, written step-by-step. Note that we read and write the expansions from the least significant digit to the most significant one and we start at the digit corresponding to $2^0 = 1$.

This approach required us to manually model the digit conversion as a transducer (or, in this particular instance, to find a reference in the available literature). Shouldn't there be an easier method using the full power of SageMath? For sure there is.—And we will do so in later examples to demonstrate various approaches for constructing transducers.

3.2 The Non-Adjacent Form of Fourteen

Let us compute the non-adjacent form of, for example, our lucky number fourteen.^(x) We intend to use the transducer above, so, for convenience, we set `NAF = NAF1`. Remember that you can see the transducer NAF in Figure 2.

As a first try, we type

```
NAF_of_14 = NAF(14.digits(base=2))
```

^(x) You may ask why 14 is our lucky number. In fact, it is not! But it is not that bad. This number is, beside the actual 13, our second lucky number. The reason of preferring 14 over 13 is simple and of educational character: The digit expansion (used in this tutorial) of 13 is too symmetric. This may lead to confusion whether this expansion is read from left to right or the other way round.

and get

```
ValueError: Invalid input sequence.
```

An error message?—Huh?—So did we make a mistake in our construction? Fortunately not; the transducer has just not finished yet: With the input `[0, 1, 1, 1]`, which is the binary expansion of 14 from the least significant to the most significant digit, we would stop in the nonfinal state with label 2, as we can see by typing

```
NAF.process(14.digits(base=2))
```

which results in `(False, 2, [0, -1, 0])`. Here, the first component indicates whether the input is accepted or not; the second component is the label of the state where we stopped. The third component of this triple is the output of the transducer irrespective of whether the state is final or not.

By adding enough trailing zeros to the expansion of 14, we reach a final state and we ensure that all carries are processed. We type

```
NAF_of_14 = NAF(14.digits(base=2) + [0, 0, 0])
```

and get the output `[0, -1, 0, 0, 1, 0]`. This list corresponds to the digits of the non-adjacent form of 14, starting with the digit corresponding to 1 at the left, and then continuing with the digits corresponding to 2, 4, 8, 16, and 32.

But do we really want to think about trailing zeros? This should be done by SageMath. And that is possible by

```
NAF = NAF.with_final_word_out(0)
```

This function constructs a final output for every state. The final output of a state is appended to the “normal” output if we stop reading the input in this state. Transducers with final output are called *subsequential*, cf. [34]. The method `with_final_word_out()` computes the final output by reading as many zeros as necessary to reach a final state (if possible). The corresponding output is then the final output.

Now, we compute the non-adjacent form of 14 again by

```
NAF_of_14 = NAF(14.digits(base=2))
```

without thinking about how many trailing zeros we have to add. And the result `[0, -1, 0, 0, 1]` is still the same as before (except for one trailing zero).

3.3 Calculating the Non-Adjacent Form with Less Thinking

A different approach to construct a transducer calculating the non-adjacent form is via a transition function.

To get this function, we think about the following algorithm rewriting the binary expansion to the NAF: We start by determining the least significant digit n_0 of the non-adjacent form of the integer n . This can be decided by looking at the two least significant digits of the binary expansion: If n is even, then the digit of the non-adjacent form is zero. If n is odd, it is 1 or $\bar{1}$, depending on n modulo 4. As the next step of this algorithm, we have to compute the non-adjacent form of $\frac{1}{2}(n - n_0)$.

We can reformulate this as a transition function. As we have to look at two consecutive input digits, we simply read and store the very first input digit by inserting an additional rule for the initial state. This leads to the following code:

```
def NAF_transition(state_from, read):
    if state_from == 'I':
        write = None
        state_to = read
        return (state_to, write)
    current = 2*read + state_from
    if current % 2 == 0:
        write = 0
```

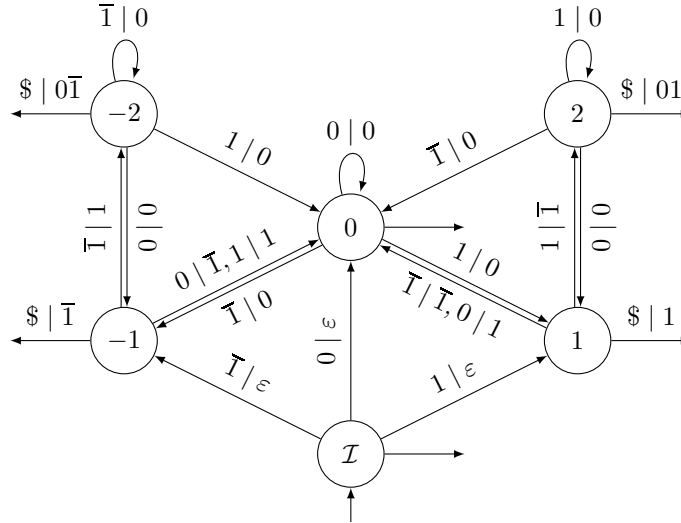


Figure 4: A transducer transforming any binary digit expansion with digits -1 , 0 and 1 into a non-adjacent form.

```

elif current % 4 == 1:
    write = 1
else:
    write = -1
state_to = (current - write) / 2
return (state_to, write)

```

Here, $\%$ is the remainder of the integer division in SageMath.

The transducer defined by this transition function can be built by

```

NAF2 = Transducer(NAF_transition,
                  initial_states=['I'],
                  final_states=[0],
                  input_alphabet=[0, 1]).with_final_word_out(0)

```

We can check whether the two transducers are the same by

```
NAF == NAF2
```

which, luckily, yields `True`.

If we think again about our algorithm at the beginning of the section, we see that we can also use a different input alphabet, e.g. $[-1, 0, 1]$. Then we obtain a transducer which can transform any binary digit expansion with digits -1 , 0 and 1 into a non-adjacent form. The code is

```

NAF_all = Transducer(NAF_transition,
                    initial_states=['I'],
                    final_states=[0],
                    input_alphabet=[-1, 0, 1]).with_final_word_out(0)

```

and you can see a picture of this transducer in Figure 4. Comparing `NAF` in Figure 2 and `NAF_all` in Figure 4 suggests that `NAF` is contained in `NAF_all` as a subgraph because such an inclusion holds for their digit sets as well: $\{0, 1\} \subseteq \{-1, 0, 1\}$.

3.4 Back to Automata

Now we can live up to our promise from Section 2: We can use the output of the transducer `NAF_all` to construct the automaton accepting non-adjacent forms once more.^(xi)

Now, we simply “forget” the input of every transition and only consider the output labels, which we can do by

```
NAF_language4 = NAF_all.output_projection().minimization()
```

Again we can check the equivalence by `NAF_language4.is_equivalent(NAF_language)` which yields `True`.

So far so good; we have played around with automata and transducers a lot. But now it is time to get in touch with some other mathematical areas. In the following section, we will consider a minimization problem.

3.5 What Makes the Non-Adjacent Form Better than Other Expansions?

To answer this question, first take a look at Figure 4. Forget everything you have read up to now for a moment and use some imagination. Does the object in this figure remind you of anything?—A spiderweb?—Well, let us rephrase the question: Does the object in this figure remind you of any well-known mathematical object?—Go one or two steps back.—Better now?—Do you see a graph? More precisely a directed graph with labels?—Very well.

Since our mind could do this switching to a different world, SageMath should be able to do this as well. We type

```
def weight(word):
    return sum(ZZ(d != 0) for d in word)
G = NAF_all.digraph(
    edge_labels=lambda t: weight(t.word_in) - weight(t.word_out))
```

and voilà, here is a graph `G`. In the code above, we use the convention that the integer values of `True` and `False` are 1 and 0 respectively. The edge labels of `G` will be explained in a moment.

Now back to the question posed in the title of this part. Let us first ask what “better” means. The answer can be stated as follows: A *minimal digit expansion* has the lowest possible number of nonzero digits (aka the *Hamming weight*) among all possible expansions of the same number with base 2 and digits -1 , 0 and 1 . Now, the question is: Is the non-adjacent form a minimal digit expansion? And this is exactly the point where graphs come into play.

We have chosen the labels of the directed graph `G` such that they are the difference between the weights of the input and the output of the transition. Assume for a moment that there is a path in the graph `G` from the initial state \mathcal{I} to some final state with a negative sum of labels. In terms of the transducer `NAF_all` this means that we have read some input with lower number of nonzeros than the corresponding output (which is a non-adjacent form). Therefore, the non-adjacent form is not a minimal digit expansion; we have found an expansion (the input) with lower weight.

So much for this thought experiment; we start a search for negative paths, i.e, look at the shortest paths from \mathcal{I} to any state. If all these paths have a nonnegative weight, then the non-adjacent form is always minimal. Such a search can be done by the Bellman–Ford algorithm (cf. for example [10]), which is used by SageMath when we call

```
paths = G.shortest_path_lengths('I', by_weight=True)
```

It returns

```
{0: 0, 1: 1, 2: 1, 'I': 0, -2: 1, -1: 1}
```

So, for example, the shortest path from state \mathcal{I} to state -2 has weight 1 in the graph. This means that an input of the transducer `NAF_all` (taking exactly this path from \mathcal{I} to -2) has a weight one larger than its output.

All these weights are nonnegative. So we can be relieved now because we know that the non-adjacent form is always a good choice.

^(xi) We choose `NAF_all` rather than `NAF2` (or any of the other transducers), because `NAF2` only allows positive numbers as input.

3.6 A Third Construction of the Same Transducer

We are now back in our construction half-marathon. The non-adjacent form can also be generated in the following way. We start with the binary expansions of $\frac{3n}{2}$ and of $\frac{n}{2}$. We subtract each digit of $\frac{n}{2}$ from the corresponding digit of $\frac{3n}{2}$. This leads to a digit expansion of n with digits $\{-1, 0, 1\}$ in base 2. One can prove that this digit expansion is the non-adjacent form of n (cf. [8], see also [36, Theorem 10.2.4]).

Note that in the following, we will add a summand $0 \cdot \frac{1}{2}$ to expansions like (2). This means that we write

$$14 = (100\bar{1}0.0)_2.$$

It will turn out that this is convenient, since we are working with halves a lot in our example in the following sections.

For this construction we need a few simple transducers (as, for example one for multiplying by 3 and one for performing subtraction), which we combine later appropriately. We will also reuse these machines in a later example for the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form in Section 4.

So let us start with the times-3-transducer, i.e., one that takes the binary expansion of a number n as input and outputs $3n$ (in binary). We do this, as above, by a transition function. We define

```
def f(state_from, read):
    current = 3*read + state_from
    write = current % 2
    state_to = (current - write) / 2
    return (state_to, write)
```

to compute the next output digit (`write`) and the new carry (encoded in `state_to`) from the input digit (`read`) and the previous carry (`state_from`) in the multiplication-by-3-algorithm. From this transition function we get the following transducer:

```
Triple = Transducer(f, input_alphabet=[0, 1],
                    initial_states=[0],
                    final_states=[0]).with_final_word_out(0)
```

Eager as we are, we test this construction by

```
three_times_fourteen = Triple(14.digits(base=2))
```

and get `[0, 1, 0, 1, 0, 1]`, which equals 42. Hooray!

Back to business; our goal is to calculate binary- $3n$ minus binary- n . To do so, we need a transducer which acts as identity (for the binary- n -part), i.e., writes out everything that is read in. Here,

```
Id = Transducer([(0, 0, 0, 0), (0, 0, 1, 1)],
                 initial_states=[0], final_states=[0],
                 input_alphabet=[0, 1])
```

does the trick. We also get the above by

```
prebuiltId = transducers.Identity(input_alphabet=[0, 1])
```

where we just have to specify the alphabet `[0, 1]`.

As a next step (before we are heading to subtraction), we want a transducer which produces pairs of the digits of $3n$ and of n simultaneously. This can be achieved with

```
Combined_3n_n = Triple.cartesian_product(Id).relabelled()
```

Let us test this machine by

```
fortytwo_and_fourteen = Combined_3n_n(14.digits(base=2))
```

It returns [(0, 0), (1, 1), (0, 1), (1, 1), (0, None), (1, None)], which seems to be correct.

We further construct a transducer computing the component-wise difference: Its input is a pair like the output of `Combined_3n_n` and the output is the difference of the two entries. We generate the transducer by

```
Minus = transducers.operator(
    lambda read_3n, read_n: ZZ(read_3n) - ZZ(read_n),
    input_alphabet=[None, -1, 0, 1])
```

where the `lambda`-function specifies our operator. Here we use that `ZZ(None)` is 0.

Of course, there is not only a prebuilt identity transducer, but also a prebuilt transducer for component-wise difference, available as

```
prebuiltMinus = transducers.sub([-1, 0, 1])
```

But unfortunately, it can only work with numbers, and we also want to subtract `None`. The final outputs are the reason: Sometimes, one component is `None`. For example, the final output of state 1 is

```
final_word_out = Combined_3n_n.state(1).final_word_out
```

which yields [(1, None)].

Finally, by

```
NAF3 = Minus(Combined_3n_n).relabelled()
```

we obtain a transducer computing the non-adjacent form of $3n - n = 2n$. This means, `NAF3` is built as the composition of `Minus` and `Combined_3n_n`, which could also have been called by using the method `composition()`.

Let us test this construction. For example,

```
NAF_of_14 = NAF3(14.digits(base=2))
```

returns [0, 0, -1, 0, 0, 1]. This is, once again, the non-adjacent form expansion of 14, see (2), but now starting with the digit corresponding to $\frac{1}{2}$ (which is obviously 0) at the left, and then continuing with the digits corresponding to 1, 2, 4, 8 and 16.

Now we have finished our warm-up and are ready for the main example, which will be dealing with $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form.

4 An Example: Three-Half–One-Half-Non-Adjacent Form

4.1 What Is the Three-Half–One-Half-Non-Adjacent Form?

We have (or, at least, we can calculate) the non-adjacent forms of $\frac{3n}{2}$ and of $\frac{n}{2}$. Inspired by the construction of the NAF presented in Section 3.6 (as a remainder: we subtracted two binary expansions), we build the difference of these two NAFs as well. Thus, we define the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form of an integer n as the digit expansion obtained by subtracting each digit of the NAF of $\frac{n}{2}$ from the corresponding digit of the NAF of $\frac{3n}{2}$. This leads to a digit expansion of n with digits $\{-2, -1, 0, 1, 2\}$ in base 2. For example, the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form of—guess which number comes now—14 is

$$14 = (10101.0)_2 - (100\bar{1}.0)_2 = (1\bar{1}102.0)_2.$$

In a first step, we want to calculate this new expansion; see the following sections. On the one hand, we are lazy and want to reuse as much as possible from the finite state machines we have already constructed. On the other hand, we are motivated to use our new knowledge on working with those automata and transducers. So the idea will be to combine several of these known transducers appropriately.

4.2 Combining Small Transducers to a Larger One

We first combine the transducers `Triple` and `NAF` to obtain a transducer to compute the non-adjacent form of $3n$. For convenience, we choose `NAF = NAF3`, because there we do not have to consider an empty output of a transition.

We use

```
NAF3n = NAF(Triple)
```

which builds the composition of the two transducers involved and therefore gives us a gadget to get the non-adjacent form of $3n$.

Next, we construct a transducer which builds the non-adjacent forms of $3n$ and n simultaneously by

```
Combined_NAF_3n_n = NAF3n.cartesian_product(NAF).relabelled()
```

The function `cartesian_product()` sounds familiar—we used it in Section 3.6 already. It constructs a transducer which writes pairs of digits.

Finally, by reusing `Minus`, we construct

```
T = Minus(Combined_NAF_3n_n).relabelled()
```

This transducer finally computes the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form. To get some information like the number of states of the finite state machine, we type `T` in `SageMath` and see

```
Transducer with 9 states
```

Let us continue with the example from the beginning. To compute this new digit expansion of 14, we type

```
expansion_of_14 = T(14.digits(base=2))
```

The output is

```
[0, 0, 2, 0, 1, -1, 1]
```

which is the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form of 14 starting with the digit corresponding to $\frac{1}{4}$. This starting digit has the following reasons: The output of the transducer `NAF` starts with the digit corresponding to $\frac{1}{2}$ when reading n . We use the non-adjacent form of $\frac{n}{2}$, which thus starts at the digit corresponding to $\frac{1}{4}$.

4.3 Getting a Picture

Up to now, we constructed a couple of (or one might call it “many”) finite state machines, but we barely saw one. Okay, to be fair, we have drawn an automaton in Section 2.1.—But how did this work exactly?

With `T.plot()` we get a first graphical representation of the transducer. This was easy, but we are not fully satisfied. For example, the labels of the transitions are missing. And, maybe we want to rearrange the states a little bit to obtain less crossings of the transitions. This can be achieved by the following: `view(T)` gives a second graphical representation of the transducer. Maybe the arrangement of the states is not nicer than before, but we will improve this a lot.

We first choose the coordinates of the states by

```
T.set_coordinates({
    0: (-2, 0.75),
    1: (0, -1),
    2: (-6, -1),
    3: (6, -1),
    4: (-4, 2.5),
    5: (-6, 5),
    6: (6, 5),
    7: (4, 2.5),
    8: (2, 0.75)})
```

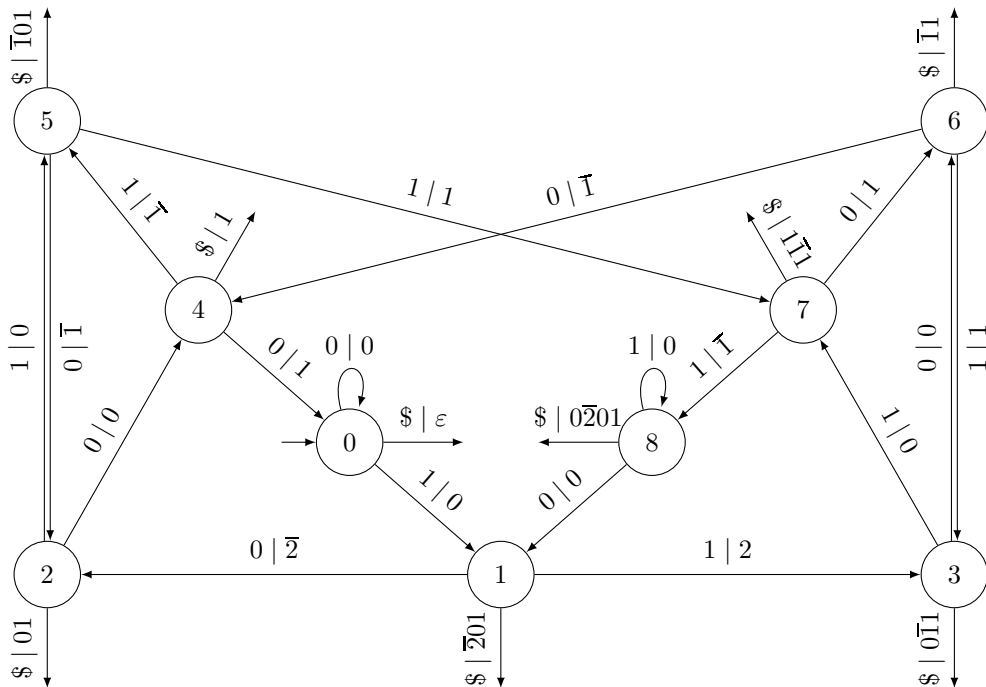


Figure 5: Transducer T to compute the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form of n .

Furthermore, for transition labels, we prefer “ $\bar{1}$ ” over “ -1 ”, so we choose the appropriate formatting function. Additionally, we choose the directions of the arrows with the final outputs.

```
T.latex_options(format_letter=T.format_letter_negative,
                 accepting_where={
                     0: 'right',
                     1: 'below',
                     2: 'below',
                     3: 'below',
                     4: 60,
                     5: 'above',
                     6: 'above',
                     7: 120,
                     8: 'left'},
                 accepting_show_empty=True)
```

Now, the output of `view(T)` in SageMath looks like Figure 5. The $\$$ -symbol signals the end of the input sequence. Further customization of the underlying TikZ-code is possible, see the documentation of `latex_options()`.

On the other hand, by typing `latex(T)` we get this TikZ-code for the transducer, which can be used to include a figure of the finite state machine in a L^AT_EX-document, like it was done for this tutorial. To succeed, we need to use the package `tikz` and to include the line `\usetikzlibrary{automata}` in the preamble of the L^AT_EX-document.

4.4 Accepting More Strange Animals

As for the non-adjacent form (Sections 3.4), we can construct an automaton accepting all $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent forms by

```
R = T.output_projection().minimization()
```

It has 17 states, so we do not show it here. However, we can still compute the number of $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent forms of length n (see also Section 2.3). A simple

```
N = R.number_of_words(n, CyclotomicField(3))
```

results in

$$\frac{1}{9} \left(-e^{\frac{2}{3}i\pi}\right)^n \left(e^{\frac{2}{3}i\pi} + 2\right) - \frac{1}{9} \left(e^{\frac{2}{3}i\pi} + 1\right)^n \left(e^{\frac{2}{3}i\pi} - 1\right) + \frac{1}{9} \cdot 2^n + \frac{1}{18} (-1)^n + \frac{1}{2}.$$

What was that `CyclotomicField(3)` about? It was a hint for `SageMath` to compute in that number field instead of computing in the field of algebraic numbers. This is more efficient and produces nicer output.

Let us persuade ourselves that the answer is correct by considering the first few values.

```
bool(sum(N.subs(n=i) for i in srange(6)) == len(list(R.language(5))))
```

yields `True`. In fact, `R.language(5)` iterates over all words accepted by `R` of length at most 5.

4.5 (Heavy) Weights

Our original motivation to study the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form comes from analyzing its (Hamming) weight, i.e., the number of nonzero digits. We want to compare the Hamming weights of the different digit expansions: standard binary expansion, non-adjacent form and $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form.

Using the ideas of the constructions above, this is also not difficult. We construct a transducer computing the weight of the input by

```
def weight(state_from, read):
    write = ZZ(read != 0)
    return (0, write)
Weight = Transducer(weight, input_alphabet=srange(-2, 2+1),
                    initial_states=[0], final_states=[0])
```

The transducer `Weight` writes a 1 for every nonzero input, which means that the weight is encoded in unary in the output string.

There also exists a prebuilt transducer which we could use instead of our own construction. It is available via

```
prebuiltWeight = transducers.weight(srange(-2, 2+1))
```

Composing the weight-transducer with the one calculating the $\frac{3}{2}$ - $\frac{1}{2}$ -NAF by

```
W = Weight(T)
```

we end up with a transducer with 9 states computing the Hamming weight of this new digit expansion of n (given in binary). For instance,

```
W(14.digits(base=2))
```

yields `[0, 0, 1, 0, 1, 1, 1]`, which means the weight is 4.

The transducer `W` can be further simplified by

```
W = W.simplification()
```

Why did we not use `minimization()` as we did in Section 2? The reason is that we work with transducers now, in contrast to Section 2, where we used automata. There is not always a unique minimal transducer for a given transducer (cf. [9]). But we can at least simplify the transducer to obtain one with a smaller number of states.

If you wonder, why there is the word “heavy” in the title of this part, read on until the end of the example.

4.6 Also Possible: Adjacency Matrices

We want to asymptotically analyze the expected value of the Hamming weight of our new digit expansion for all positive integers less than 2^k , where k is a fixed large number.

We dive a bit deeper into the mathematics involved now. One way to perform the asymptotic analysis is by means of the adjacency matrix of the transducer. By

```
var('y')
def am_entry(trans):
    return y^add(trans.word_out) / 2
A = W.adjacency_matrix(entry=am_entry)
```

we obtain a matrix where the entry at (k, l) is $\frac{1}{2}y^h$ if there is a transition with output sum h from state k to l and 0 otherwise. The adjacency matrix generated by this command is

$$A = \begin{pmatrix} 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \\ \frac{1}{2}y & 0 & 0 & 0 & 0 & 0 & \frac{1}{2}y & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2}y^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2}y^2 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & \frac{1}{2}y & 0 \\ 0 & 0 & 0 & \frac{1}{2}y & 0 & 0 & 0 & 0 & \frac{1}{2}y \\ 0 & \frac{1}{2}y & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

For $y = 1$, this is simply the transition probability matrix. Its normalized left eigenvector to the eigenvalue 1 gives the stationary distribution. We write

```
dim = len(W.states())
I = matrix.identity(dim)
(v_not_normalized,) = (A.subs(y=1) - I).left_kernel().basis()
v = v_not_normalized / v_not_normalized.norm(p=1)
```

and obtain $(1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9, 1/9)$.

To obtain the average Hamming weight of the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form, we compute the expected output vector in each state as

```
expected_output = derivative(A, y).subs(y=1) * vector(dim*[1])
```

and obtain $(0, 1, 1, 0, 1, 0, \frac{1}{2}, 1, \frac{1}{2})$. Note that the derivative here simply computes the expected output for every transition. We could also have called `adjacency_matrix()` with a suitably modified entry function.

The expected density is therefore

```
v * expected_output
```

which yields $\frac{5}{9}$. This means that the main term of the average number of nonzero digits in $\frac{3}{2}$ - $\frac{1}{2}$ -NAFs of length k is $\frac{5}{9}k$.

4.7 More on the Hamming Weight by Letting SageMath Do the Work

We have the impression that the analysis of the previous section could be done (or, rephrased, we want that this should be done) more automatically. Indeed, we can let SageMath do the work for us, and it does it very well: It not only outputs the mean of the Hamming weight, but also its variance and more.

By

```
var('k')
moments = W.asymptotic_moments(k)
```

we obtain a dictionary whose entries are the expectation and the variance of the sum of the output of the transducer, and the covariance of the sum of the output and the input of the transducer (cf. [18]). The probability model is the equidistribution on all input sequences of a fixed length k .

The expected value of the Hamming weight of the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form is

$$\frac{5}{9}k + \mathcal{O}(1).$$

as k tends to infinity.

This function can also give us the variance of the Hamming weight of the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form, which is

$$\frac{44}{243}k + \mathcal{O}(1).$$

Of course we could do a lot more beautiful stuff. We could construct a bivariate generating function. From this, we could obtain more terms and better error terms of the asymptotic expansion of the expected value, the variance and higher moments. We could also prove a central limit theorem. And everything by using the full power of `SageMath`. But, again, we do not want to go into details here. We refer to the book by Flajolet and Sedgewick [11] for details on the asymptotic analysis of digit expansions and other sequences.

4.8 What Does This Mean for This Brand New Digit Expansion?

In the past several sections, we were able to calculate the average Hamming weight of the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form asymptotically by the help of the finite state machines package in `SageMath`. But what does this result tell us?

So let us compare this digit expansion with the standard binary expansion and the classical non-adjacent form. The expected value of the Hamming weight of the standard binary expansion can be calculated by

```
expectation_binary = Id.asymptotic_moments(k)['expectation']
```

which gives

$$\frac{1}{2}k + \mathcal{O}(1).$$

Of course, it was not necessary to use the transducer `Weight` here (since we only have digits 0 and 1). The expected value of the weight of the NAF can be obtained with the code

```
expectation_NAF = Weight(NAF).asymptotic_moments(k)['expectation']
```

which produces the weight

$$\frac{1}{3}k + \mathcal{O}(1),$$

cf. also [26]. Note that in this particular construction (only digits -1 , 0 and 1), we could have used the prebuilt transducer

```
Abs = transducers.abs([-1, 0, 1])
```

instead of the weight-transducer.

Both values are (asymptotically) less than the

$$\frac{5}{9}k + \mathcal{O}(1)$$

of the $\frac{3}{2}$ - $\frac{1}{2}$ -non-adjacent form, which means that this expansion has much more nonzero digits on average and therefore is much “heavier”. So, from a point of view of minimizing the Hamming weight, this new expansion is disappointing: It uses more digits but realizes a larger weight.

5 More Questions and Answers

5.1 Does This Automata-Package Have a History?

Indeed it has. One of the authors wrote a similar unpublished package for Mathematica [38], which is used in the articles [4, 5, 6, 13, 14, 15, 16, 19, 20, 21, 22, 23, 24]. But this implementation was of limited scope and the rise of SageMath made it clear that it is ripe for a redesigned version. Having the finite state machine module readily available in a publicly available and continuously maintained system also leads to more transparency in the computational parts of publications.

5.2 What about Other Finite State Machine Packages?

There are quite a lot of implementations for finite state machines available. One of the fastest libraries is OpenFST [27], for which a Python interface [28] exists, too. But since it is written in C/C++, it does not work well with the mathematical objects defined in SageMath. Vcsn (Vaucanson) [37] is written in C++ and offers a Python interface as well. Other non-Python modules are, for example, [2, 7, 35]. There also exist a couple of Python packages, e.g. [3, 12, 29], which can also be found on the Python-Wiki.^(xii) Some of those are specialized (and thus not flexible enough) and implement only partial support for both automata and transducers. It seems that some of them are even out-dated and not developed any further.

5.3 How Fast is This Package?

To get a feeling, let us have a look at the following concrete example: We construct a minimal automaton forbidding the subwords $10^n 1$ for all $1 \leq n < 15$. We can achieve this in the same manner as in Section 2.2—there we have forbidden adjacent nonzeros—by

```
S = [automata.ContainsWord('o' + n*'z' + 'o', input_alphabet=['o', 'z'])
      for n in range(1, 15)]
P = sum(S[1:], S[0])
A = P.determinisation().complement()
M = A.coaccessible_components().minimization()
```

While P has only 147 states, the automaton A consists of 245 760 states. Due to this large number of states, it takes SageMath roughly three minutes to create A out of P. In comparison, the compiled automaton^(xiii) of the C-library Vcsn [37] achieves this in slightly less than one tenth of the time. The resulting minimal automaton M has 16 states; its creation out of A is not time-critical.

Of course, speed comparisons between an interpreted, dynamic programming language such as Python and an optimized C-library can only have one outcome. One main design goal of the SageMath package is flexibility and a good interplay with mathematical objects. In fact, all labels of states and transitions are allowed to be arbitrary SageMath-objects (and not only letters or strings as in the example shown here). Another design goal was to have a very low entry barrier for using the package due to its inclusion in SageMath.

5.4 How Complex is Everything?

Many algorithms for automata have an exponential worst case running time. (Think of determinisation!) However, we tried to choose efficient algorithms whenever possible. For instance, SageMath's package implements Moore's algorithm for minimization of deterministic automata,

^(xii) The Python-Wiki can be found at <https://wiki.python.org/moin/FiniteStateMachine>.

^(xiii) The equivalent code of the example in Section 5.3 using the Python bindings of Vcsn [37] is

```
import vcsn
context = vcsn.context("lal_char(oz), b")
any_word = context.expression('\z').automaton().complete().complement()
S = [any_word * context.expression('o' + n*'z' + 'o').automaton() * any_word
      for n in range(1, 15)]
P = sum(S[1:], S[0])
A = P.complete().determinize().complement()
M = A.trim().minimize()
```

which has a polynomial worst case complexity [33, Sec. I.3.3.3]. Additionally, the user may also choose Brzozowski’s algorithm. This algorithm has an exponential worst case complexity but performs well in empirical experiments [1], and it works with nondeterministic automata, too.

5.5 What will the Future Bring Us?

Of course, nobody knows—but there is room for improvement and extensions. Performance of the package might be improved by Cythonizing^(xiv) the source code or by calling specialized packages for time-critical operations.

References

- [1] Marco Almeida, Nelma Moreira, and Rogério Reis, *On the performance of automata minimization algorithms*, CiE 2008: Abstracts and extended abstracts of unpublished papers, 2008.
- [2] *The automata standard template library*, <http://astl.sourceforge.net>, 2013.
- [3] *automata 0.1.4*, <https://pypi.python.org/pypi/automata>, 2013.
- [4] Roberto Avanzi, Clemens Heuberger, and Helmut 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.
- [5] ———, *Arithmetic of supersingular Koblitz curves in characteristic three*, Cryptology ePrint Archive, Report 2010/436, 2010.
- [6] ———, *Redundant τ -adic expansions I: Non-adjacent digit sets and their applications to scalar multiplication*, *Des. Codes Cryptogr.* **58** (2011), 173–202.
- [7] *dk.brics.automaton 1.11-8*, <http://www.brics.dk/automaton/>, 2011.
- [8] Sze-hou Chang and Nelson T. Tsao-Wu, *Distance and structure of cyclic arithmetic codes*, Proc. Hawaii International Conference on System Sciences, vol. 1, 1968, pp. 463–466.
- [9] Christian Choffrut, *Minimizing subsequential transducers: a survey*, *Theoret. Comput. Sci.* **292** (2003), no. 1, 131–143, Selected Papers in honor of Jean Berstel.
- [10] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver, *Combinatorial optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Inc., New York, 1998.
- [11] Philippe Flajolet and Robert Sedgewick, *Analytic combinatorics*, Cambridge University Press, Cambridge, 2009.
- [12] *FSA – Finite State Automaton processing in Python*, <http://www.osteele.com/software/python/fsa/>, 2004.
- [13] Peter J. Grabner, Clemens Heuberger, and Helmut Prodinger, *Counting optimal joint digit expansions*, *Integers* **5** (2005), no. 3, A9.
- [14] Clemens Heuberger, *Minimal expansions in redundant number systems: Fibonacci bases and greedy algorithms*, *Period. Math. Hungar.* **49** (2004), 65–89.
- [15] ———, *Redundant τ -adic expansions II: Non-optimality and chaotic behaviour*, *Math. Comput. Sci.* **3** (2010), 141–157.

^(xiv) Cython (semi-)automatically translates Python code to C code and then compiles it.

- [16] Clemens Heuberger, Rajendra Katti, Helmut Prodinger, and Xiaoyu Ruan, *The alternating greedy expansion and applications to left-to-right algorithms in cryptography*, Theoret. Comput. Sci. **341** (2005), 55–72.
- [17] Clemens Heuberger, Daniel Krenn, and Sara Kropf, *Finite state machines, automata, transducers*, <http://trac.sagemath.org/15078>, 2013, module in Sage 5.13.
- [18] Clemens Heuberger, Sara Kropf, and Stephan Wagner, *Variances and covariances in the central limit theorem for the output of a transducer*, European J. Combin. **49** (2015), 167–187.
- [19] Clemens Heuberger and James A. Muir, *Minimal weight and colexicographically minimal integer representations*, J. Math. Cryptol. **1** (2007), 297–328.
- [20] Clemens Heuberger and Helmut Prodinger, *Analysis of alternative digit sets for nonadjacent representations*, Monatsh. Math. **147** (2006), 219–248.
- [21] ———, *The Hamming weight of the non-adjacent-form under various input statistics*, Period. Math. Hungar. **55** (2007), 81–96.
- [22] ———, *On α -greedy expansions of numbers*, Adv. in Appl. Math. **38** (2007), 505–525.
- [23] ———, *Analysis of complements in multi-exponentiation algorithms using signed digit representations*, Internat. J. Found. Comput. Sci. **20** (2009), 443–453.
- [24] Clemens Heuberger, Helmut Prodinger, and Stephan G. Wagner, *Positional number systems with digits forming an arithmetic progression*, Monatsh. Math. **155** (2008), 349–375.
- [25] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman, *Introduction to automata theory, languages, and computation*, Addison-Wesley series in computer science, Addison-Wesley, 2001.
- [26] François Morain and Jorge Olivos, *Speeding up the computations on an elliptic curve using addition-subtraction chains*, RAIRO Inform. Théor. Appl. **24** (1990), 531–543.
- [27] *OpenFst Library 1.5.0*, <http://openfst.org>, 2015.
- [28] *pyopenfst*, <https://github.com/tmbdev/pyopenfst>, 2014.
- [29] *python-automata 1.0*, <https://code.google.com/p/python-automata/>, 2007.
- [30] George W. Reitwiesner, *Binary arithmetic*, Advances in Computers, Vol. 1, Academic Press, New York, 1960, pp. 231–308.
- [31] The SageMath Developers, *SageMath Mathematics Software (Version 6.10)*, 2015, <http://www.sagemath.org>.
- [32] William A. Stein et al., *Sage Mathematics Software (Version 5.13)*, The Sage Development Team, 2013, <http://www.sagemath.org>.
- [33] Jacques Sakarovitch, *Elements of automata theory*, Cambridge University Press, Cambridge, 2009, Translated from the 2003 French original by Reuben Thomas.
- [34] Marcel-Paul Schützenberger, *Sur une variante des fonctions séquentielles*, Theoret. Comput. Sci. **4** (1977), no. 1, 47–57.
- [35] *SFST 1.4.7a*, <http://www.cis.uni-muenchen.de/~schmid/tools/SFST/>, 2015.
- [36] Jacobus Hendricus van Lint, *Introduction to coding theory*, Graduate Texts in Mathematics, vol. 86, Springer, 1992.
- [37] *Vcsn 2.1*, <https://www.lrde.epita.fr/wiki/Vcsn>, 2015.
- [38] Wolfram Research, Inc., *Mathematica (Version 5.2)*, 2005.