Mathematical Theory, Computing Implementation and Digital Electronics Simulation of Kaprekar's Routine.

1st SCHOTT Pierre
Institut Supérieur d'Electronique de Paris - Isep
10 rue de Vanves, Issy-les-Moulineaux, 92130, France
pierre.schott@isep.fr, pierrot.schott@laposte.net
ORCID: 0000-0002-8456-4430

2nd LEHMANN Andrew
Institut Supérieur d'Electronique de Paris - Isep
10 rue de Vanves, Issy-les-Moulineaux, 92130, France
andrew.lehmann@isep.fr
ORCID: 0000-0002-5398-8230

***Abstract*—The purpose of this 'Innovative Practice' article is to propose a multidisciplinary project spanning discrete mathematics, computer programming and electronics through the study of Kaprekar's routine. The theoretical component consists of teaching the concepts of arithmetic in different bases and cycle searches in groups of finite cardinality. Students will then realise this algorithm computationally, learning notions of iterative and modular programming. Finally they will implement it with combinatorial and sequential digital electronics with circuit simulation software.**

Kaprekar's routine is as follows: choose a number, then form the two numbers from the rearrangement of the digits in descending and ascending order and take their difference. Repeat this procedure with the result of the subtraction until you recover a previous number, and hence the process is a repeating cycle, or the result settles on a constant called the Kaprekar constant. The result of the routine depends on both the base and the number of digits. For example, in base ten, an initial choice of any 4 digit number will always result in the constant 6174 whereas a 5 digit number will result in a cycle of 3 or 4 numbers. This simple routine can be studied from multiple approaches, giving rise to a rich multidisciplinary student experience.

From the mathematical approach, we can easily find these constants in base 10 on the internet, but less so in base n. We present the demonstration for 3-digit numbers in base 10. Using a computer programming approach, we have developed a Python script to find these constants and cycles in bases 2 to 10. We describe the algorithms of 11 functions contained in this script as a possible teaching template. Taking a digital electronics approach, we developed several circuits and sub-circuits using the open-source software package Logisim. We also present the automated creation of circuits from truth tables using Karnaugh tables. In addition, this project naturally contains measures against student plagiarism. The Kaprekar constants or cycles depend on the bases used, and hence the same project can be assigned to several groups of students who cannot copy from one another.

In short, the algorithm for finding Kaprekar's constants and cycles is very simple, yet the mathematical demonstrations, computer programming implementation and digital electronics simulation are far less trivial. But in the end, this project would produce a prototype that even primary school children can use in a little game!

***Index Terms*—Electrical engineering education, hybrid learning, electronics simulations, numerical algorithms**

I. Introduction

The push for innovation and openness in modern society has given rise to a paradigm of creativity in the technology sector. Management styles that are flexible and project-based are seen as better able to respond to rapidly changing environments. Some educational researchers [1] are contributing to the debate by asking the following questions: What about the creativity paradigm at universities? What are the implications for scientific work and teaching? The present article takes the view that if we want to promote creativity among our students, let's be creative ourselves!

In the French school system, we typically introduce small "unitary" notions (like bricks of knowledge) as we go along, and only at the end of a course is the essential notion finally presented! In this way, during the learning process, students who need to know the purpose of each little notion (and if there is a purpose at all!) do not engage with the study. The consequence is that students are unable to pile up bits of knowledge for lack of motivation, and ultimately fail to learn the essential notion we are aiming for.

We can also teach in the opposite direction: show, describe, and explain the final notion immediately; but then some students stop listening to the intermediate stages because they feel they have understood the only important notion of the course. They may therefore gain a conceptual understanding but never learn the details in the depth required to display independent skills. To try to address the weaknesses of both approaches, we can combine them: identify 3-4 key intermediate notions that have a clear purpose, introduce the first intermediate notion and build on it little by little, and repeat the operation.

In this 'Innovative Practice' article we present a project that combines this mixed pedagogical approach with the creativity paradigm: an approach with intermediate notions (mathematics project, computing project, electronics project) for an overarching project theme that is not a classic topic in one of the subjects used (and/or taught). This project is to study a curious mathematical process called the Kaprekar algorithm,

in which we take any number and recursively calculate the difference of the two numbers formed by rearranging the chosen number's digits in descending and ascending order. The routine is mathematically simple, yet the results of the process are surprising, and combined with considering how the process works with different number bases presents a problem theme that is not generally encountered in standard early university mathematics courses.

In section II we detail the Kaprekar routine and define Kaprekar constants and cycles. Then we give the explicit mathematical demonstration giving the Kaprekar constant for 3-digit numbers in base 10 in section III. This demonstration reveals the lack of a general proof for all cases, motivating the project to search for these constants or cycles for different digits in different bases with a computer program. A possible programming implementation with Python is described in section IV, where we detail the functions used and how they relate to each other. In section V we present a digital electronics implementation using software for simulating electronic circuits. We conclude with some teaching ideas for extending these projects.

II. Kaprekar constants and cycles

We present the Kaprekar algorithm, where one calculates the difference between numbers formed by rearranging the digits of a number in descending and ascending orders. If the repeated application of this process on the result is equal to a unique number, the ending number is called a Kaprekar constant after the Indian mathematician D. R. Kaprekar [2].

In [3], Martin Gardner shows that no Kaprekar constant exists for numbers with 1, 2, 5 and 7 digits, there are two 6 digit constants, and he gives the constants for 8 (97508421), 9 (864197532) and 10 (9753086421) digit numbers. However, while these numbers are stable by Kaprekar's routine they are not the only outcome for other starting numbers with the same number of digits. For $P = 8$ for example, there is also the following cycle

$$86526432 \longmapsto 64308654 \longmapsto 83208762 \longmapsto 86526432$$

The algorithm presented in this section generalises the process to search for all possible constants and cycles even in non base-10 number systems. A general search opens up some motivating questions:

- for which digits do we end up with a Kaprekar constant, a cycle, or neither?
- will we find unique constants or cycles, or can there be more than one for a given P?
- how does all this depend on the base of the number?

A. Kaprekar algorithm

Given a number base P and an integer written in this base with N digits, denoted

$$(A)_P = a_N a_{N-1} \cdots a_1 a_0, \tag{1}$$

we apply the following:

1) If all the digits of A are identical then no calculation is made.
2) Otherwise apply the following algorithm:
   a) rearrange the digits of A to find:
      i) the digits in descending order, denoted
      $$(M)_P = M_N M_{N-1} \cdots M_1 M_0 \tag{2}$$
      ii) the digits in ascending order, denoted
      $$(m)_P = m_N m_{N-1} \cdots m_1 m_0 \tag{3}$$
   b) calculate the difference $M - m$.

B. Example application of the algorithm

Let's take the number 794 in base 10 that we denote $(794)_{10}$. We apply the previously presented algorithm:

1) The digits of A are not all identical, so we can make the calculations.
2) For the number $(794)_{10}$:
   a) rearrange the digits to find:
      i) the digits in descending order, denoted $(M)_P = (974)_{10}$
      ii) the digits in ascending order, denoted $(m)_P = (479)_{10}$
   b) Calculate the difference $M - m = (495)_{10}$.

C. How do we obtain a Kaprekar constant?

We repeat the algorithm proposed in II-A on its results. If this process converges on a number, we call this number a Kaprekar constant. These constants depend on the number of digits in the initial number and of course also on the base.

Let's go back to the example used in II-B:

1) The initial number was $(794)_{10}$
2) The first iteration of the Kaprekar algorithm gives the number $(495)_{10}$.
3) The second iteration of the Kaprekar algorithm gives the initial number $(495)_{10}$.

So the number $(495)_{10}$ is a Kaprekar constant. We will show later that this is indeed the unique constant for all initial choices of 3 digit numbers in base 10, as long as the digits are not all identical.

In some cases there is no such constant, such as base 10 numbers with 5 digits. But there's a property that's just as interesting!

D. When there are no constants, there exist Kaprekar cycles

When the search algorithm presented in II-C does not converge on a Kaprekar constant, then there will always exist one or several cycles. That is, a series of J numbers which are going to repeat as we continue the search algorithm.

Let's take as an example the number $(79)_{10}$. The first iteration of the algorithm gives the number $(97)_{10} - (79)_{10} = (18)_{10}$. Then by iterating the algorithm over the results we obtain in turn $(81)_{10} - (18)_{10} = (63)_{10}$, $(63)_{10} - (36)_{10} = (27)_{10}$, $(72)_{10} - (27)_{10} = (45)_{10}$, $(54)_{10} - (45)_{10} = (09)_{10}$,

$(90)_{10} - (09)_{10} = (81)_{10}$ and the numbers 18 and 81 give the same result by the algorithm! So a cycle has appeared:

$$63 \longmapsto 27 \longmapsto 45 \longmapsto 09 \longmapsto 81 \longmapsto 63$$

We represent Kaprekar's routine starting at every 2-digit number in figure 1. The thick black line shows the above cycle that the routine enters no matter the starting number, as long as that number has unique digits.

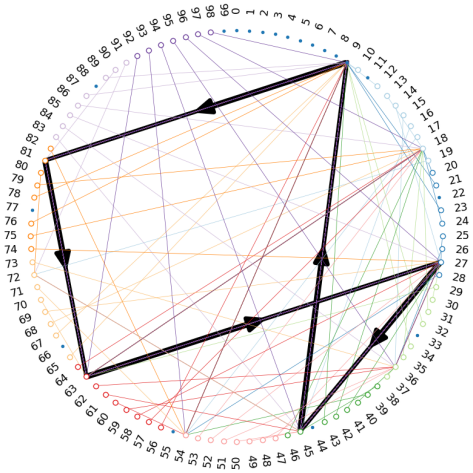

Fig. 1. Representation of Kaprekar's routine starting from all 2 digit numbers. The thick black line represents the cycle that all paths eventually enter.

III. Mathematical demonstration

A. Demonstration for a 3 digit number in base 10

Here we present a demonstration written by french secondary school students [4]. Let n be an integer with three digits a, b and c, not all equal. Assume that a, b and c are arranged in ascending order, i.e. that $a \geq b > c$ or $a > b \geq c$. Then let $abc - cba = efg$. We obtain the following system of equations:

$$\begin{cases} g &= (10+c) - a \\ f &= (10+b) - (1+b) \\ e &= a - (c+1) \end{cases} \tag{4}$$

which has solution:

$$\begin{cases} e+g &= a\text{–}(c+1) + (10+c)\text{–}a = -1+10 = 9 \\ f &= (10+b)\text{–}(1+b) = 10\text{–}1 = 9 \end{cases} \tag{5}$$

With this result, we have only a few cases to study. These cases are all multiples of 99. There are 10 cases in all, which in fact boil down to 5 pairs: 099 or 990, 198 or 891, 297 or 792, 396 or 693 and 495 or 594. We apply the algorithm to all of these cases and end up with the Kaprekar constant 495, except for the 99 case which ends up with 0.

B. As many demonstrations as there are constants to be found

In the demonstration for 4-digit numbers in base 10 proposed by [5], we find a system of inequalities to be solved which involves solving 6 different systems of equations. The only solution (other than 0) to these 6 systems is the number $(6174)_{10}$ as the unique Kaprekar constant we are looking for.

So we see that there are as many mathematical demonstrations as there are possibilities for combining the value of a base and the number of digits! So we are going to repeat the demonstration in such a way as to give an idea of the demonstrations.

IV. A programming implementation

Though the Kaprekar process is remarkably simple, it presents a pedagogically rich idea as the basis of a programming project. The learning outcomes range from simple concepts like understanding the base system of numbers and how to practically convert numbers between bases, to more advanced programming concepts like how to structure a code around a central function that must be repeatedly used. There is also the opportunity to make a comparison between iterative and recursive techniques to code same function.

Here we give one possible code implementation for finding Kaprekar constants and cycles for numbers of any number of digits and any base N between 2 and 10. An instructor could choose to constrain students to this particular workflow, shown in figure 2, so that the student concentrates on coding the particular functionalities of each sub-algorithm, or more advanced students can be asked to design an entire workflow from scratch.

For our implementation, we have developed a Python script without the usage of any libraries outside of the Python Standard Library. The key idea of our implementation is to represent numbers in base N by integers (and not by lists!). So the number $(1011)_2$ is represented by the integer $(1011)_{10}$ and not by its true decimal equivalent $(11)_{10}$.

A. 11 functions of around 220 lines

The program we have written calculates all the Kaprekar constants and cycles for all numbers containing P digits (with no restrictions on P) in any base N (N between 2 and 10). It is written in Python, using iterative functions, without classes and with automated memory management by the computer. It contains 11 functions, one main program and a total of around 220 lines. Its architecture is shown in figure 2 and we hope that by reading it you'll understand the algorithm behind it.

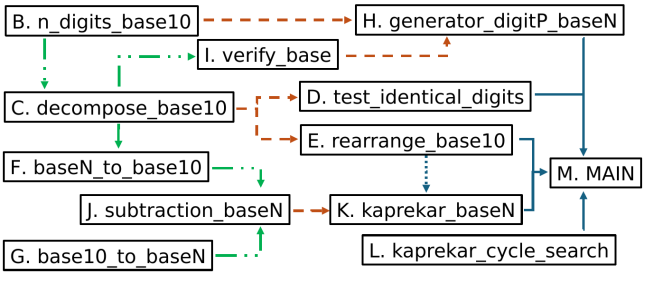

Fig. 2. Flowchart diagrams of the program's 11 functions to find Kaprekar constants and cycles.

B. n_digits_base10

This function returns the number of digits of an integer.

ALGORITHM: ***The function compares the input number to different values of*** 10^i***,*** i ***being the number of digits of the integer.***

Example: for inputs $(4265)_{10}$ or $(1011)_2$ the function returns a 4.

C. decompose_base10

This function returns a list containing the digits of an integer and the number of digits that make up the integer.

ALGORITHM: ***This function uses the function "n_digits_base10" to find the number of digits in the input integer. It then performs a Euclidean division of the input integer by*** 10^i***, where*** i ***is decreasing starting from the number of digits in the number.***

Example: for input $(4265)_{10}$ the function returns 4 and [4,2,6,5], for input $(1011)_2$ it returns 4 and $[1, 0, 1, 1]$.

D. test_identical_digits

This function tests whether an integer is composed only of identical digits; if so, the boolean returned is set to 'true'.

Example: for the input $(1011)_2$ it returns 'False' and for the input $(1111)_2$ it returns 'True'.

E. rearrange_base10

This function returns the smallest possible number and the largest possible number achievable by rearranging the digits of an integer. It does not test whether the number consists of a single digit.

ALGORITHM: ***The function decomposes the input number into digits, then sorts the list from smallest to largest and finally calculates the numbers as*** $nb = \sum a_i \times 10^i$**.**

Example: for the input $(4265)_{10}$, the function returns $(6542)_{10}$ and $(2456)_{10}$.

F. baseN_to_base10

This function converts a base-N number into its base-10 representation. Note that the 2 numbers are stored as integers! The function inputs are the number and the base.

ALGORITHM: ***The function utilises the following property:*** $(nb)_{10} = \sum a_i * (base)^i$**.**

Example: for the input $(1011)_2$, $base = 2$ and the function returns $(11)_{10}$.

G. base10_to_baseN

This function is the inverse of the "*baseN_to_base10*" function. It converts a base-10 number into its base-N representation.

ALGORITHM: ***The function determines the number of digits required to represent the number in base*** N***, then by successive division by*** $(base)^i$***, the new digits are calculated.***

Example: for the input $(11)_{10}$ to be converted to base 2, the function returns $(1011)_2$.

H. generator_digitP_baseN

This function generates all base-N numbers containing P digits. An option allows only P-digit numbers to be generated, i.e. numbers starting with one or more 0s are discarded.

ALGORITHM: ***Let the desired base be*** N ***and the number of digits*** P***. We generate a list containing all the integers between 0 and*** 10^P***. For each element in the list, the function checks whether this number belongs to base*** N ***and, depending on the option chosen, whether or not it begins with 0.***

Example: for $P = 2$ and $N = 3$ with the option of only P-digit numbers, the function returns the following list of numbers, always encoded by integers! $(10)_3$, $(11)_3$, $(12)_3$, $(20)_3$, $(21)_3$ and $(22)_3$. If we don't select the option, then this last list is completed by the numbers $(00)_3$, $(01)_3$ and $(02)_3$.

I. verify_base

This function returns a boolean value of 'True' if the number belongs to the required base, 'False' otherwise. It is used by the function "*generator_digitP_baseN*".

ALGORITHM: ***The input number is transformed into a list of digits, then for as long as the digits are strictly smaller than the base value, the function tests the next digit.***

Example: for the input $(17652)_7$ and base 7 the function returns 'False'. For the same input but base 8 the function returns 'True'.

J. subtraction_baseN

This function subtracts 2 numbers expressed in base N. It will be used if the result is positive, but if the result is negative, then the subtraction is correct.

ALGORITHM: ***Each number in base*** N ***is converted into base 10 and then subtracted in base 10. The result is then converted back to base*** N**.**

Example: for inputs $(651)_7$ and $(156)_7$, the function gives the resulting difference $(462)_7$.

K. kaprekar_baseN

This function calculates the result obtained by the Kaprekar algorithm defined in II-A for an initial number expressed in base N.

ALGORITHM: ***In turn, the function uses functions already described: calculation of the 2 extremum numbers, then subtraction of these 2 numbers. To ensure that the result is not clearly false, we then test whether the result obtained belongs to the same base using the function "verify_base".***

Example: for input integer $(516)_7$ the result of this function is $(462)_7$.

L. kaprekar_cycle_search

This function returns a boolean if it finds a cycle in the list of numbers obtained by iterations of the Kaprekar algorithm, as well as the length of the cycle and a list containing the cycle. The function tests the previous K elements between 1 and K_{max} to find a cycle.

ALGORITHM: ***The function retrieves the number of elements in the list, and if it is less than*** K_{max}***, then*** K_{max} ***is modified to this length. For*** K ***that can vary from*** 1 ***to*** K_{max}***, for*** i ***that can vary from the "start" of the list to the end of the list, the function tests whether*** $element[i-K]$ ***is equal to*** $element[i]$***. If it is equal, then the function has found a cycle of length K whose elements lie between*** $element[K-i]$ ***and*** $element[i]$***.***

Example: for initial integer $(97)_{10}$, the list returned by the Kaprekar algorithm is $[(97)_{10},\ (18)_{10},\ (63)_{10},\ (27)_{10},\ (45)_{10},\ (09)_{10},\ (81)_{10}, \cdots (63)_{10},\ (27)_{10},\ (45)_{10},\ (09)_{10},\ (81)_{10}]$.

1) For a K_{max} of 2, 3 or 4 the boolean returned is False;
2) For a K_{max} of 5 or more the booleen returned is True, the length is 5 and the cycle list is $[(63)_{10},\ (27)_{10},\ (45)_{10},\ (09)_{10},\ (81)_{10}]$

M. Main

With all the functions we've developed, we can easily find all the Kaprekar constants and cycles for all bases between 2 and 10 and for any number of digits belonging to $\mathbb{N}$. Once these numbers have been found, we can visualize the results in the form of lists, figures or summary tables.

What hasn't yet been developed is the ability to count the number iterations of Kaprekar's algorithm required to arrive at the beginning of a cycle (or constant) associated with the base used and the number of digits of the initial number.

V. A DIGITAL ELECTRONICS IMPLEMENTATION

Exactly as with the programming project, the Kaprekar algorithm presents a pedagogically rich theme for an electronics project. Here we present a possible project to design a simulated circuit for computing the Kaprekar constant starting from any 3-digit number in base 10. This project illustrates, for example, how truth tables and Karnaugh maps are practically implemented in electronics and how simulation software is used to design circuits.

To extend this project, an Arduino-based hardware realization could be made, which would further interweave algorithmics and digital electronics. Calculations would be carried out using a program implemented on an Arduino board, while visualization would use digital components.

A. Description of the Logisim software

Logisim[1] is a program for simulating combinatorial and sequential digital electronic circuits. It is relatively light (6.65 Mb) and free to use. It allows you to synthesize circuits from truth tables, and even find the truth table from a circuit.

B. Circuit implementation

The complete circuit diagram for implementing the Kaprekar algorithm for 3-digit numbers in base 10 using Logisim is shown in figure 3. We have also created a similar circuit for 4-digit numbers in base 10, which is even more imposing.

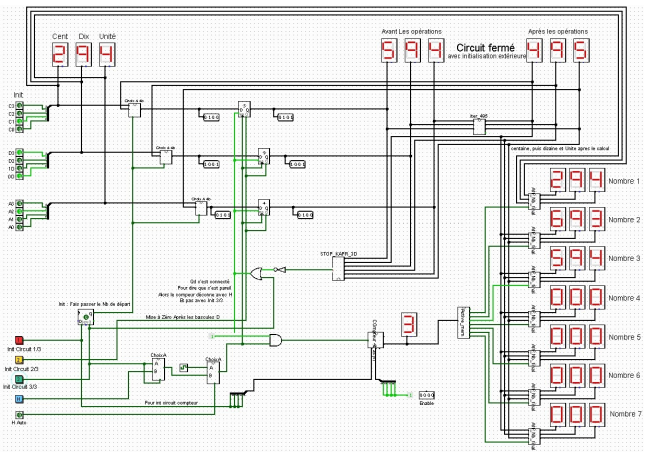

Fig. 3. Electronic circuit in Logisim looking for the Kaprekar constant for 3-digit numbers in base 10, with the following example $(294)_{10}$.

We see that for the initial number $(294)_{10}$ (top left), the Kaprekar algorithm gives the numbers $(693)_{10}$ then $(594)_{10}$ (stored bottom right) then $(495)_{10}$ (top right) and which is not yet stored. At the bottom, we see the number 3, the number of times the Kaprekar algorithm has been used.

It would be too long and tedious to present all the sub-circuits that make up this overall circuit. Unlike the presentation of computer code, we're going to do things the other way round: we'll start with the final application and then present the sub-circuits. So we're only going to present a few circuits and sub-circuits!

C. Functional description of the final circuit

The circuit is composed of 5 principal parts:

1) At the top left is the initial number (here $(294)_{10}$).;

[1] http://www.cburch.com/logisim/index.html

2) in the middle: circuits that allow you to choose whether the initial number is transmitted or the Kaprekar number calculated at phase i, if this is not identical to the Kaprekar number calculated at phase (i-1); **this is the sequential part of the circuit**.
3) At bottom right is the display of the different numbers calculated during the search for the Kaprekar cycle or constant (here $(294)_{10}$, $(693)_{10}$, $(594)_{10}$) with the number of times the algorithm has already been used (here 3);
4) At top right is the circuit that performs the Kaprekar algorithm presented in IV-L (here with the numbers $(594)_{10}$ which gives $(495)_{10}$).
5) Bottom left consists of two sub-sections:
   a) the initialization part of the circuit (reset all numbers to zero, the counter to zero and then pass the initial number to the input of the circuit performing the Kaprekar algorithm)
   b) the circuit clock, which automatically repeats the Kaprekar algorithm until the Kaprekar constant is found. When this is reached, all circuits are disabled, although the clock continues to present rising and falling edges.

We therefore have a closed-loop system controlled by the circuit clock; this system being the calculation of the Kaprekar algorithm.

D. The circuit performing the Kaprekar algorithm

The circuit performing Kaprekar's algorithm is shown in figure 4. On the left, the chosen number is $(294)_{10}$, which is transformed into 2 numbers ($(942)_{10}$ and $(249)_{10}$). The subtraction of these 2 numbers is $(693)_{10}$.

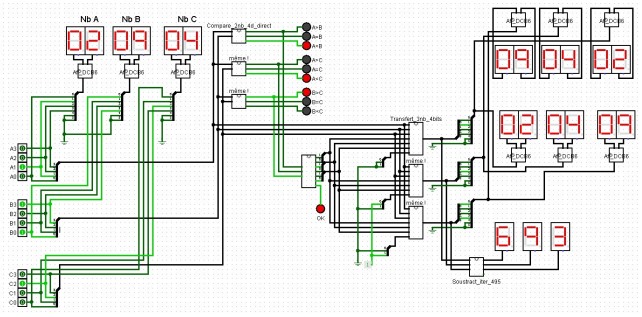

Fig. 4. Logisim electronic circuit simulating the Kaprekar algorithm for 3-digit numbers in base 10.

We can see that there are only 5 sub-circuits for this calculation, presented from left to right.

1) **CIRCUIT :** *Compare_2nb_4d_direct*
2) **CIRCUIT :** *classe_3nb_4d*
3) **CIRCUIT :** *Transfert_Nb_bonne_place_4b_3nb*
4) **CIRCUIT :** *soustract_iteration_495_dcb*
5) **CIRCUIT :** *stop_kapr_3d*

E. Compare_2nb_4d_direct: circuit comparing 2 numbers

This circuit determines whether the number A in hexadecimal is greater than, equal to or less than the number B. To synthesize it, the truth table is defined in Logisim, which then automatically generates the logic circuit using Karnaugh simplifications. Fortunately, this circuit is made up of some 500 logic gates, which is why it is not shown here.

1) As inputs we have 2 times 4 bits denoted A_3, A_2, A_1, A_0, B_3, B_2, B_1 and B_0
2) As output we have *A_sup_B* (which equals 1 if $A > B$, 0 if not), *A_egal_B* (which equals 1 if $A = B$, 0 if not and *A_inf_B* (which is not quite the complement of A_sup_B because it is 1 if $A < B$ and 0 otherwise.)

In figure 4, this circuit has been used 3 times and the outputs are visualized by the middle LEDs. We note that the first circuit, which compares 2 and 9, has 001 as its output, corresponding to $2 < 9$.

F. classe_3nb_4d: circuit sorting 3 numbers by comparing 2 numbers

This circuit classifies 3 numbers A, B and C in hex. We don't compare the 12 bits that make up these 3 numbers, but we do retrieve 3 pieces of information: $A > B$?, $A > C$? and $B > C$?; output from the *Compare_2nb_4d_direct* circuits.

To synthesize the circuit, the truth table is defined in Logisim, partially shown in the left of figure 5. Logisim then generates the logic circuit using Karnaugh simplifications, shown in the right of figure 5.

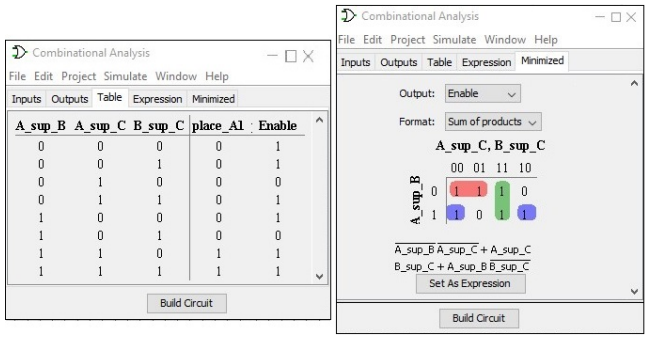

Fig. 5. For a circuit which classifies 3 numbers in hex: (Left) truncated Logisim truth table and (Right) Karnaugh table and Logisim simplification.

G. Transfert_Nb_bonne_place_4b_3nb: circuit transferring a 3-digit base-10 number to a desired channel

This circuit is the equivalent of a MUX with:

1) as input:
   a) the three 4-digit numbers A, B and C
   b) the order of the numbers A, B and C coded by 00 for the smallest number, 01 for the middle number and 10 for the largest number.
   c) the place of interest 00 or 01 or 10.
2) as output: the desired number

The circuit is shown in figure 6, using the number $(294)_{10}$ as an example.

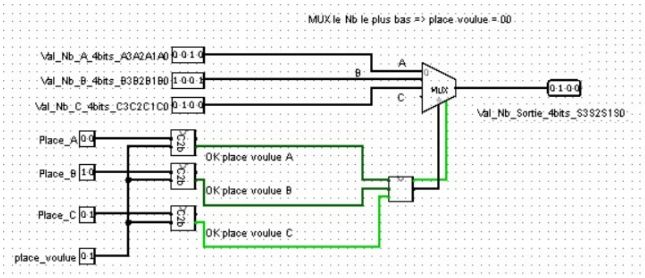

Fig. 6. Logisim circuit for transferring the correct number in relation to its place in a 3-number sequence.

This circuit is used 3 times to select the smallest digit, then the middle digit and then the largest digit; this allows us to form the largest and smallest number made up of the 3 digits A, B and C.

H. soustract_iteration_495_dcb: circuit subtracting two 4-digit numbers in base 10

This circuit performs the subtraction between 2 numbers in base 10, each containing 3 digits. Of course, it consists of several sub-circuits, which in turn consist of several sub-circuits. However, this remains an extremely common and classic problem!

We're not interested in the sign, as we'll always have a positive number when subtracting thanks to Kaprekar's algorithm. So we could simplify this subtractor! However, we prefer to synthesize a subtractor that calculates all cases.

We use the property of subtraction in binary that a subtraction is a two's complement of the addition of 2 numbers.

1) Synthesize a subtractor between 2 hex numbers, $A = a_3a_2a_1a_0$ and $B = a_3a_2a_1a_0$, shown in figure 7.

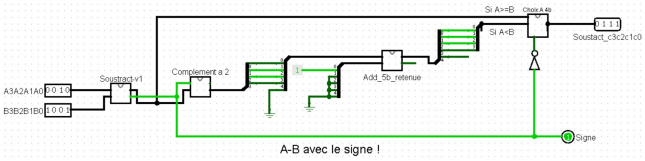

Fig. 7. Logisim circuit subtracting 2 hex-coded numbers (here $(2)_{10} - (9)_{10} = (-7)_{10}$.

To do this, we need to synthesize:

a) a circuit that adds 2 bits a_i and b_i and a potential remainder r_i. This circuit outputs the least significant bit of $a_i + b_i + r_i$, denoted c_i, and the most significant bit, denoted r_{i+1};
b) a 5-bit adder by propagating the remainder;
c) a "two's complement" circuit for a 5-digit binary number;

2) Synthesize a subtractor between two base-10 numbers encoded in binary-coded decimal (BCD) using Kaprekar's algorithm. To do this, we use the previous circuit 3 times. However, only one number is used as input, which is the smallest base-10 number $(DEF)_{10}$, and this circuit calculates $(FED)_{10} - (DEF)_{10}$. It then calculates $D - F$, then $E - E$ with a possible remainder and finally $F - D$ with a possible remainder. This circuit is shown in figure 8.

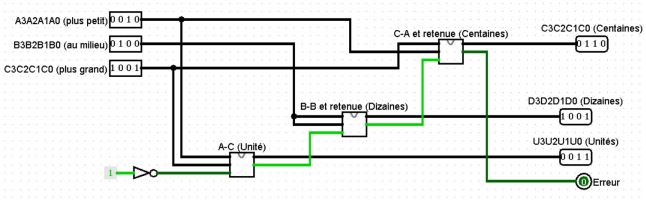

Fig. 8. Logisim circuit performing subtraction in Kaprekar's algorithm for the number $(294)_{10}$; input 2 then 4 then 9 coded in hex.

I. stop_kapr_3d: circuit stopping the Kaprekar process

This circuit stops the Kaprekar process. When this same process arrives at a single constant (as is the case here for 3-digit numbers), this circuit is a very simple combinatorial circuit comparing only the 2 numbers: the one at step i and the one at step $i + 1$.

J. Same project different implementation: weighted binary code

To avoid the temptation for students to simply copy/paste from one of their classmates, we can introduce a weighted binary code associated with each student. This way, they can work together, or even explain their solution to each other, but each student will still have to complete all the circuits. Indeed, as each binary code used will be different, it will be necessary to at least synthesize all the transcoders going from a weighted code to a functionality!

For this project, a weighted code using weights 1-1-2 and 5 is perfectly suitable (with choice conditions on the number and place of bits to discriminate 2 binary words that have the same decimal equivalent).

VI. Conclusion and going further

We have presented in detail a multi-disciplinary project centered on the simple mathematics idea of the Kaprekar routine. The project interweaves a non-standard mathematics calculation, programming and electronics simulations, allowing a student to deepen their understanding of these topic by by application to a single theme. Here we describe natural extensions to the project for instructors who would like to go further in one of the parts of the project.

A. Simulation of a cycle detection sub-circuit

We have introduced the use of flip-flops only as memory (which can be seen here as switches). If we wish to continue teaching/using sequential logic, we need to use a cycle. For numbers containing 2 digits, Kaprekar's process doesn't result in a constant but in a cycle, as shown in the left of figure 9. There are 2 circuit designs for this cycle:

- Let's say we use 2 cycles: one for the units and the other for the tens as shown in the right of figure 9. For

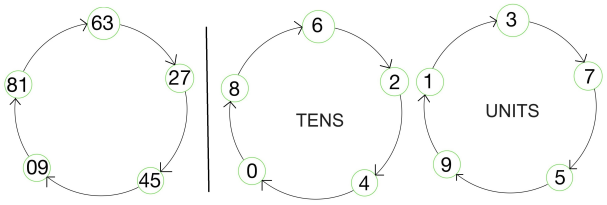

Fig. 9. (Left) Kaprekar cycle for 2-digit base-10 numbers. (Right) The left cycle broken into two cycles of tens and units.

each cycle, we have to design a combinatorial circuit controlling the inputs of each flip-flop in order to switch from one state to another. However, each number is represented in BCD by only 4 bits, and the cycle contains 5 states. So each of the 2 combinatorial circuits will have 4 inputs (i.e. 16 possible binary words) and 5 of them will be useful. Figure 10 shows the generic implementation of a 4-bit synchronous cycle based on JK flip-flops, using Clear and Preset only to force the circuit to a given state.

- Let's say the cycle we've created uses 2-digit numbers. We need to design a combinatorial circuit controlling the inputs of each flip-flop in order to move from one state to another. Represent each number in BCD. 8 bits are needed to code each number, and the cycle contains only 5 states. So the combinatorial circuit will have 8 inputs (i.e. 256 possible binary words) and only 5 of them will be useful. But in fact this amounts to the design of 2 distinct 4-bit cycles!

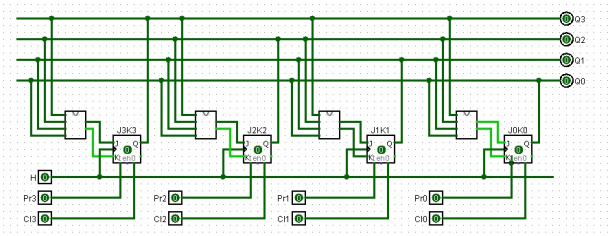

Fig. 10. 4 bit cycle: generic synchronous implementation based on JK flip-flops.

Once the Kaprekar cycle has been completed, we'll synthesize the "STOP_KAPR_2D" circuit, like the "STOP_KAPR_3D" circuit, which will output a single bit indicating whether the number obtained at iteration i (denoted N_i) belongs to the cycle or not. Thus we need to:

- Synthesize a counter from 0 to 4 in pure binary whose clock frequency (H_2) must be 5 times higher than the circuit's general clock frequency;
- Compare N_i with the number of cycles generated in step j;
- At the end of all comparisons, set the output bit to 1 if the number N_i belongs to the cycle, 0 otherwise.

B. Recursive programming

An instructor can add a lesson in recursive programming by rewriting the function "generator_digitP_baseN" presented in section IV-H in a recursive style. Recall that this function generates all base-N numbers containing P digits. A recursive version can be written as follows:

ALGORITHM: ***Let N be the base used. If the number of digits P is greater than 1, then the function will call itself with the number of digits $P-1$. When the number of digits P is finally 1, after a few calls, the function returns the list $[0,1,2,\ldots,N-1]$ if the chosen option is all elements, even those starting with 0; otherwise the list is $[0,1,2,\ldots,N-1]$. Then, for each element l_i of the list, the program generates N elements worth l_i j where j is successively $0,1,\cdots,N-1$. At the end of the calls, the function will have generated the set of base-N numbers containing P digits.***

Example: for $P=2$ and $N=3$ with the option of only P-digit numbers, the function returns the following list of numbers, always encoded by integers! $(10)_3$, $(11)_3$, $(12)_3$, $(20)_3$, $(21)_3$ and $(22)_3$.

C. Circuit realisation with Arduino

To complete the project, students could design a practical implementation. For example, using an Arduino board, 7-segment displays and push buttons, the sutdents could develop a prototype that calculates only the Kaprekar constants in base 10 and the number of times the subtraction operation is performed. Before ordering the parts, they could simulate their model using an Arduino emulator. After a successful demonstration, they could then build a model "for real" (and also face the problem that it never works as the simulation predicted!

In short, with these ideas for further development, each student would have seen a project through from A to Z: from reading the specifications to practical realization, via a theoretical demonstration study, a computer implementation, an electronic simulation and realization.

ACKNOWLEDGMENT

The authors would like to thank Louis-Joseph Brossollet (Director, strategic projects and student experience at ISEP) for introducing (to us physicists!) this mathematical recreation: the Kaprekar routine. We'd also like to thank Lionel Trojman (Director of Research at ISEP) for supporting us on this path of pedagogical innovation, and of course the main actor... Mr. Kaprekar.

REFERENCES

[1] https://cerisy-colloques.fr/universitecreativite2023
[2] D. R. Kaprekar, "An interesting property of the number 6174," Scripta Mathematica, vol. 15, pp. 244–245, 1955
[3] M. Gardner, "The magic numbers of Dr. Matrix," Buffalo, N.Y. : Prometheus Books, 1985
[4] https://www.mathenjeans.fr/sites/default/files/comptes-rendus/kaprekar_clg_alainfournier_orsay_2018-2019.pdf
[5] Y. Nishiyama, "The Weirdness of number 6174," International Journal of Pure and Applied Mathematics, vol. 80, No.3, pp363-373, 2012