RANDOM GENERATION OF A GENOCCHI PERMUTATION

Lala Arimonjy Ramelina[1], Falimanana Randimbindrainibe[2]

[1] *Assistant lecturer, Ecole Doctorale Science et Technique de l'Industrie et de l'Innovation, University of Antananarivo, Antananarivo, Madagascar*
[2] *Professor, Ecole Doctorale Science et Technique de l'Industrie et de l'Innovation, University of Antananarivo, Antananarivo, Madagascar*

ABSTRACT

A recursive method to generate Genocchi permutations is yet to be proposed. On the one hand, this work aims to provide a random generation of a Genocchi permutation and on the other hand, to call upon a free software to allow the use of this permutation in applied aspects such as the queue management or any other related perspectives.

Keywords: *Genocchi numbers, Permutations, Genocchi permutations*

1. INTRODUCTION

The numbers of Genocchi are defined as series of numbers that are useful for some advanced enumerations such as Dumont permutations; in which each even number is followed by a smaller number; then each odd entry is followed by a larger number (an ascent) or ends the string [1]. Researches on Genocchi permutations have missed to offer a recurrent method for generating them. However, Dumont [1] has demonstrated that they can be enumerated from Genocchi numbers. The aim of this work is to propose a random generation of these permutations using the finesse of the Python language (free software) in order to apply this issue for instance in queue management.

1.1 Genocchi numbers

Genocchi numbers can appear in diverse fields such as number theory, asymptotic analysis, differential topology and modular form theory. Yet, their combinatorial properties still require further work.
The following formula is a recall that Genocchi numbers are defined by the relation [1]:

$$\frac{2t}{e^t+1} = t + \sum_{n\geq 1} (-1)^n G_{2n} \frac{t^{2n}}{(2n)!}.$$

It is observed that the relations between Bernoulli numbers B_{2n} and tangent numbers T_{2n-1} are described respectively by following equations and the first values of these three series of numbers are indicated in the Table 1.

$$\frac{t}{e^t-1} = 1 - \frac{1}{2}t + \sum_{n\geq 1} (-1)^{n+1} B_{2n} \frac{t^{2n}}{(2n)!}$$

and

$$\tan t = \sum_{n\geq 1} T_{2n-1} \frac{t^{2n-1}}{(2n-1)!}.$$

Thus, we obtain

$$G_{2n} = 2(2^{2n} - 1)B_{2n}$$

and

$$2^{2n-2}G_{2n} = nT_{2n-1} .$$

Table -1: First values of the three series of numbers from Bernoulli numbers and tangent numbers and Genocchi numbers

n	1	2	3	4	5	6	7
G_{2n}	1	1	3	17	155	2073	38227
B_{2n}	$\frac{1}{6}$	$\frac{1}{30}$	$\frac{1}{42}$	$\frac{1}{30}$	$\frac{5}{66}$	$\frac{691}{2730}$	$\frac{7}{6}$
T_{2n-1}	1	2	16	272	7936	353792	223582256

1.2 Calculation of Genocchi numbers

Dumont and Viennot [1] have developed a straightforward method for calculating Genocchi numbers. The exterior of the stair-shaped graph is filled with the number 0 (see Figure 1). The number 1 is placed on the innermost left side. A number in an even index column is obtained by adding the number to its left and the number above it. Similarly, a number in an odd index column is obtained by adding the number to its left and the number below it. Genocchi numbers are the numbers at the top of the staircase. Note that Dumont gave the first combinatorial interpretations of Genocchi numbers in terms of Genocchi permutations.

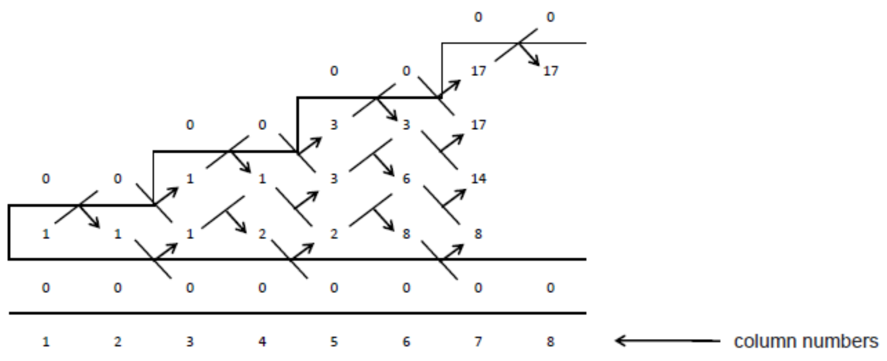

Fig -1: Stair-shaped graph showing the calculation of a Genocchi number

2. PERMUTATION ALGORITHMS

A permutation is defined by the rearrangement of n elements, the number of permutations is then given by $n!$. [2]
Establishing $\mathcal{S}_n$, the set of permutations of $\{1,2,\cdots,n\}$, for any permutation σ of $\mathcal{S}_n$, this last is defined by:

$$\sigma = \begin{pmatrix} 1 & 2 & \cdots & n \\ \sigma(1) & \sigma(2) & \cdots & \sigma(n) \end{pmatrix}$$

For the next step, we will adopt the following linear structure

$$\sigma = \sigma(1)\sigma(2)\cdots\sigma(n).$$

To generate a permutation, several algorithms can be used. Three main algorithms are cited, named Insertion algorithm, Heap algorithm and Steinhaus-Johnson-Trotter or SJT algorithm.

2.1 Insertion algorithm

First, a permutation of length $n-1$ is set, after searching for all potential permutation, n is inserted in the possible spaces of each permutation of length $n-1$ to have all permutations of length n. [2]

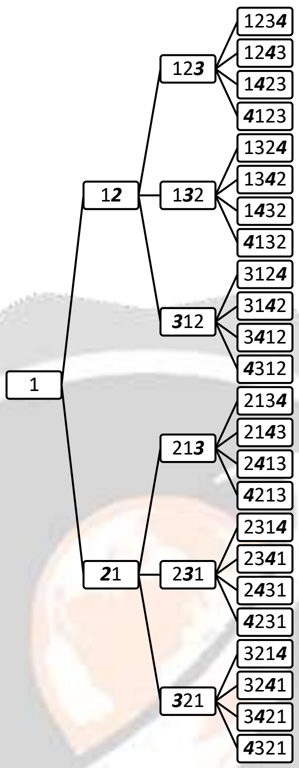

Fig -2: Structure of a permutation by insertion for four elements

2.2 Heap Algorithm

Given a permutation of length n, the first element is fixed and the rest is permuted. In that sense, the first element is also fixed and the others are permuted. The same process is repeated until only two elements remain (length 2) and they can be simply exchanged. [2]

2.3 Steinhaus-Johnson-Trotter Algorithm

Classical algorithm permutations were based on a lexicographical order (alphabetical or ascending for numbers) and they proceed through a propagation way. The Steinhaus-Johnson-Trotter algorithm (like the Heap algorithm) delivers a permutation in a particular order [2]. This algorithm works by direct routing among all permutations.
First of all, the elements are written in ascending order for instance: 1, 2, 3.
Each element is given the same direction at the beginning: $\overleftarrow{1}, \overleftarrow{2}, \overleftarrow{3}$.
A number is categorized as mobile if the number pointed to by its arrow is smaller: here 3 and 2 are mobile. The following operations with several steps can then be conducted.

- Step 1: choose the largest moving element and swap with the neighbor pointed by its arrow $\overleftarrow{1}, \overleftarrow{3}, \overleftarrow{2}$;
- Step 2: check whether there are other, larger mobile in the rank. If so, change their direction (in the case above, there is none)
- Step 3: continue with the largest element under treatment $\overleftarrow{3}, \overleftarrow{1}, \overleftarrow{2}$;
  When the mobile element can no longer move, then go back to step 1 and repeat the whole process. $\overleftarrow{3}, \overleftarrow{2}, \overleftarrow{1}$;
- Step 4: since 3 is greater than 2, then the direction of 3 will be reversed $\overrightarrow{3}, \overleftarrow{2}, \overleftarrow{1}$;
- Step 5: the element 3 becomes again the largest mobile $\overleftarrow{2}, \overrightarrow{3}, \overleftarrow{1}$;
- Step 6: the process provides a new permutation possibility $\overleftarrow{2}, \overleftarrow{1}, \overrightarrow{3}$;
- Step 7: the process ends when there are any mobile elements.

In brief, the whole process remains into setting the elements in an ascending order. As long as there is a moving element, it is always possible to find the largest moving element M; switch this number M and its pointed neighbor; and change the direction of all numbers greater than M.

3. GENOCCHI PERMUTATIONS

A permutation $\sigma \in \mathcal{S}_{2n}$ is called a Genocchi permutation [3] if $\sigma(2i-1) > 2i-1$, $\sigma(2i) \leq 2i$ for $i \in \{1,2,\cdots,n\}$. The sets of Genocchi permutations of {1,2,⋯,2n} will be then denoted as $\mathcal{G}_{2n}$.
As an example, we set a pemutation σ represented by the following equation :

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 4 & 2 & 7 & 1 & 6 & 5 & 8 & 3 \end{pmatrix}$$

It can be seen that $\sigma \in \mathcal{G}_8$, and this is represented by the graph below.

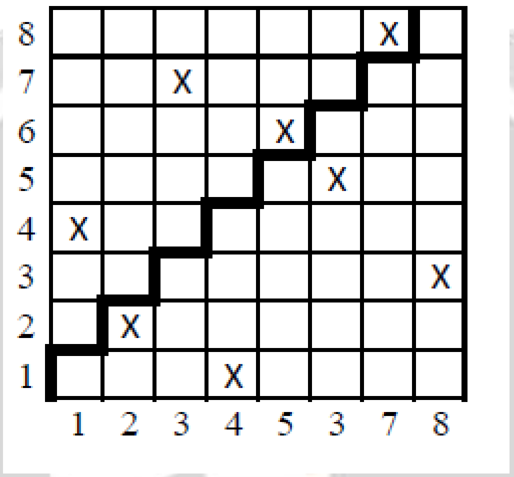

Fig -3: Graphic representation of a Genocchi permutation

Note that, for any permutation $\sigma \in \mathcal{G}_{2n}$, there is always $\sigma(2n-1) = 2n$ and Dumont has demonstrated that card$(\mathcal{G}_{2n}) = G_{2n+2}$

3.1 Generation of a Genocchi permutation

Let's propose a method to build a Genocchi permutation σ of size $2n$. It is sufficient to find the first $2n-2$ terms since $\sigma(2n-1)$ is always equal to $2n$ and $\sigma(2n)$ would be the number that was set apart during the permutation construction.
Considering the following construction that includes two row matrixes where:

- The matrix $C = [1,2,\ldots,2n-2]$ in which the first $2n-2$ terms of σ will be randomly chosen;
- The matrix B is empty at the beginning and will be filled progressively by the elements of σ.

As an instance, we will create a Genocchi permutation of size 8 with the following scripture $\sigma(1)\sigma(2)\sigma(3)\sigma(4)\sigma(5)\sigma(6)\sigma(7)\sigma(8)$. For that, we certainly have $\sigma(7) = 8$ and $\sigma(8)$ is the last number, it is noticed that for this last one, there is no flexibility in the choice for the number. Understanding that $\sigma(2i-1) > 2i-1$ and $\sigma(2i) \leq 2i$ for i ranging from 1 to n, we start by constructing the odd index elements from the largest to the smallest, then the even index elements from the smallest to the largest.
In details, below is the proposed procedure for generating the Genocchi permutation.

- The choice of $\sigma(5)$ comes from $C = [1,2,3,4,5,6,7]$ with the condition that $\sigma(5) > 5$. Given that $\sigma(5) = 6$ is selected, the matrix B becomes $B = [6]$ and 6 will no longer appear in C. We then obtain $C = [1,2,3,4,5,7]$.
- If $\sigma(3) = 7$, 7 is inserted at the beginning of B, both B and C become respectively $B = [7,6]$, $C = [1,2,3,4,5]$.
- If $\sigma(1) = 2$, B becomes $B = [2,7,6]$ and C will be $C = [1,3,4,5]$;

- As $\sigma(2) \leq 2$, let's choose $\sigma(2) = 1$. We insert $\sigma(2)$ at the 2nd place in B, consequently we have respectively for B and C, $B = [2,1,7,6]$ and $C = [3,4,5]$.
- If we set $\sigma(4) = 4$, we insert $\sigma(4)$ in the 4th place in B and the two matrixes will be respectively, $B = [2,1,7,4,6]$ and $C = [3,5]$.
- If $\sigma(6) = 3$, the two matrixes will therefore be represented respectively as the following $B = [2,1,7,4,6,3]$ and $C = [5]$.
- Finally, we just need to insert $\sigma(7) = 8$ and $\sigma(8) = 5$. The linear representation of σ will then be 21746385.

3.2 Generation of a Genocchi permutation using Python

A random generation of a Genocchi permutation is possible using free software like Python. The flexibility in the setting provides more freedom to program the permutation algorithms. The principle detailed in section 3.1 is kept, yet some syntaxes are necessary to run the software and they are listed below following by their meaning.

- *c=list(range(1,2*n))* means the matrix $C = [1,2,\ldots,2n-1]$ is initialized. In this setting, the last element of C is $2n-1$.
- *random.shuffle(c)*: this instruction is performed to randomly shuffle the elements of C. In the rest of the program, the first element of C is always chosen after it has been randomly shuffled.
- *d=t[0]*: this command imports the value $t[0]$ into the variable d. The t matrix is temporarily a copy of the last C matrix. This copy is issued from the command *t=c.copy( )*.
- *b.insert(0,d):* this command inserts the value of d at the beginning of the matrix B indexed by 0. For example, an instruction of type *b.insert(4,d)* means the insertion of the value d in row $4 - 1 = 3$. The size of the matrix B increases depending on each input and this is the advantage of this free programming languages software.
- *c.remove(d):* it means removing the item of value equal to the value d. The size of the matrix C decreases in size after this instruction.
- *b.append(c[0]):* this command adds element $c[0]$ to the last row of the matrix B.

With Python, we have the following program:

```
n=int(input('Give n = ?  '))
c=list(range(1,2*n))
random.shuffle(c)
b=[]
i=2*n-3
while i>=0:
    t=c.copy()
    d=t[0]
    t.remove(d)
    while d<=i:
        random.shuffle(t)
        d=t[0]
        t.remove(d)
    b.insert(0,d)
    c.remove(d)
    del d
    i-=2
x=2
while x<2*n:
    t=c.copy()
    d=t[0]
    t.remove(d)
    while d>x:
        random.shuffle(t)
        d=t[0]
        t.remove(d)
    b.insert(x-1,d)
    c.remove(d)
    del d
    x+=2
b.append(2*n)
b.append(c[0])
print('A Genocchi permutation of size ',2*n,' is : ')
```

```
        print(b)
```

Fig -4: A program to generate randomly a Genocchi permutation using free software Python

To illustrate the workflow in randomly generating Genocchi permutation, the following permutation (size equals 8) is set to [2, 1, 6, 4, 7, 3, 8, 5]. Same as the previous example, we provide 10 Genocchi permutations of size $2 \times 4 = 8$ randomly chosen by the program as cited below:

[2, 1, 4, 3, 6, 5, 8, 7]
[2, 1, 4, 3, 7, 6, 8, 5]
[3, 2, 5, 4, 6, 1, 8, 7]
[6, 1, 4, 2, 7, 5, 8, 3]
[3, 1, 5, 4, 6, 2, 8, 7]
[2, 1, 5, 3, 7, 6, 8, 4]
[2, 1, 5, 4, 7, 6, 8, 3]
[3, 1, 6, 2, 7, 5, 8, 4]
[3, 1, 4, 2, 6, 5, 8, 7]
[3, 2, 7, 1, 6, 5, 8, 4]

It is noticed that there is no instance of repetition in that data of 10 permutations. This case shows the efficiency of generating pseudo-random numbers by Python. Recall that for $n = 4$, the number of Genocchi permutations of size 8 is 155. By varying the integer n from 5 to 10, we can have the result below:

[6, 1, 4, 2, 9, 5, 8, 7, 10, 3]
[3, 1, 5, 2, 6, 4, 11, 8, 10, 9, 12, 7]
[4, 2, 5, 3, 6, 1, 10, 8, 13, 9, 12, 11, 14, 7]
[2, 1, 5, 3, 12, 4, 8, 7, 13, 9, 14, 10, 15, 6, 16, 11]
[3, 2, 5, 1, 6, 4, 9, 7, 10, 8, 12, 11, 14, 13, 17, 16, 18, 15]

In these examples, all the elements are distinct 2 by 2 and the conditions are always verified. For a fairly large numbers, there are no particular difficulties shown by the program except for the execution time, which is a bit extended.
This parameter time can be calculated in advance. In that sense, the "time" module with the instruction *"time.time()"* is used to give the execution time of a series of instructions expressed in seconds. If the starting time of the execution is defined by *t1= time.time()*, and the end by *t2= time.time()*, the duration is given by *t2-t1*.
For a comparative analysis according to the computer processor and the size of input data, the table below summarizes the execution time in seconds of generation of a Genocchi permutation.

Table -2: Summary of the execution time (in second) needed by the software in performing Genocchi permutation for a range of large numbers depending on the computer processor

Permutation Size	100	200	300	400	500	600	700	800	900	1000
Pentium dual core 32 bits	0.036	0.14	0.312	0.577	0.952	1.607	1.685	2.621	3.806	4.368
Intel core i5 64 bits	0.016	0.109	0.266	0.285	0.447	0.688	1.095	1.409	1.866	2.134

A graph illustrates the execution time in function of the size of the Genocchi permutation.

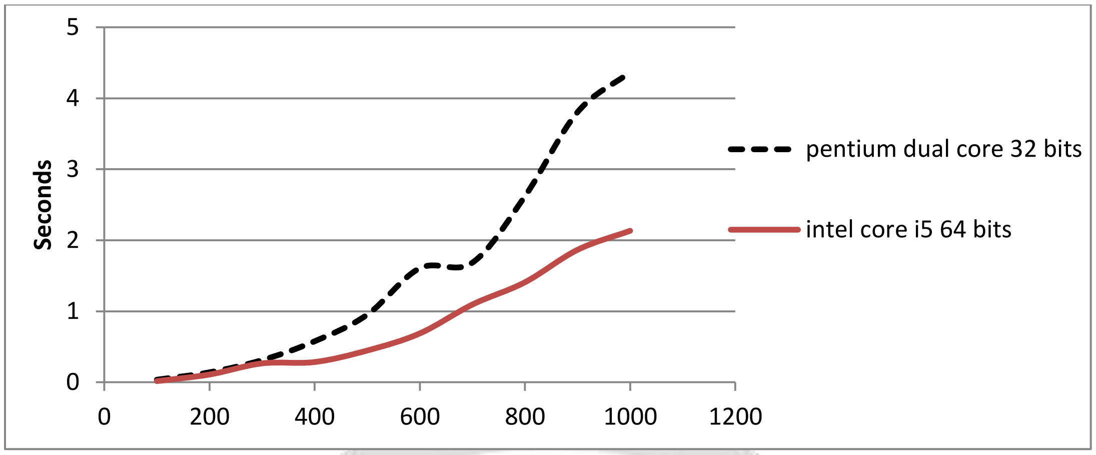

Fig -5: Graph showing the evolution of time duration in generating Genocchi permutation according to different computer processor

The continuing line curve represents the result obtained using "intel core 15 64-bit" processor which is obviously faster in execution than a "pentium dual core 32-bit". In both cases, the curves are generally increasing, but represent irregularities. The latter are due to the randomness of the generation of integers according to precise conditions: the number of loops performed is also random. We thus propose average durations for each permutation size to have much more precision and instead of giving a value for one size, it would be preferable to give the average of the 100 durations for one size. The table summarizing the time execution of the program is thus below as well as the corresponding graph.

Table -3: Summary of the execution time (in second) needed by the software in performing Genocchi permutation for 100 durations in one size depending on the computer processor

Permutation Size	100	200	300	400	500	600	700	800	900	1000
pentium dual core 32 bits	0,021	0,112	0,279	0,556	0,916	1,404	2,001	2,752	3,573	4,601
intel core i5 64 bits	0,01	0,051	0,132	0,265	0,442	0,671	0,965	1,387	1,75	2,188

For both curves, the evolution is more regular. It is recalled that the average time to generate a Genocchi permutation of length 2000 is 10.515 seconds, which means the processor "intel core i5 64-bit" requires 1051.5 seconds or 17 minutes 31.5 seconds to perform the calculation.

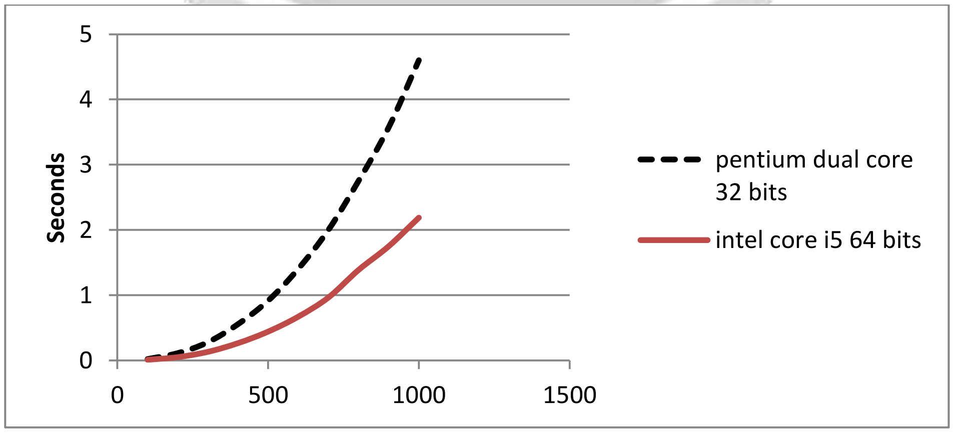

Fig -6: Graph showing the evolution of time duration in generating Genocchi permutation according to different computer processor with some averaging (average of 100 durations for one size)

4. CONCLUSION

In summary, this work suggests a random generation of the Genocchi permutations. If previous researches have calculated the number of permutations from Genocchi numbers, they have failed in providing randomly their combination. Thus, this study has highlighted a random generation of a Genocchi permutation. This work has promoted the utilization of free software in generating permutation. The choice of the software Python remains in its flexibility in programming and more automatism in the randomization. However, the time execution can be challenging depending on the computer processor type. The present work has provided – a random generation of a Genocchi permutation - has promoted its use in applied aspects such the queuing management.

5. REFERENCES

[1]. D. Dumont and G. Viennot, "*A combinatorial interpretation of the Seidel generation of Genocchi numbers*", Ann. Discrete Math. 6 (1980) 77-87.
[2]. E. J. Hartung, H. P. Hoang, T. Mütz and A. Williams, "*Combinatorial generation via permutation langage. I. Fundamentals*", Efficient Generation of Combinatorial Objects using Generalized Gray Codes, (2019).
[3]. A. Randrianarivony and J. Zeng, "*Some equidistributed statistics on Genocchi permutations*", Electronic journal of combinatorics, 3 (2) (1996).