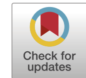
Information Sciences 654 (2024) 119842

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins



# Computable aggregations of random variables

Juan Baz [a], Irene Díaz [b,*], Luis Garmendia [c], Daniel Gómez [d], Luis Magdalena [e], Susana Montes [a]

[a] Department of Statistics, Operational Research and Didactic of Mathematics, University of Oviedo, Calle Federico García Lorca 18, Oviedo, 33007, Spain
[b] Department of Computer Science, University of Oviedo, Campus de Viesques, Gijón, 33004, Spain
[c] Department of Computer Science, Complutense University of Madrid, Madrid, Spain
[d] Departamento de Estadística y Ciencia de los Datos, Universidad Complutense de Madrid, Madrid, Spain
[e] E.T.S. Ingenieros Informáticos, Universidad Politécnica de Madrid, Campus de Montegancedo, Boadilla del Monte, Madrid, 28660, Spain

ARTICLE INFO

*Keywords:*
Aggregation functions
Computable aggregations
Aggregation of random variables
Probability theory

ABSTRACT

Aggregation theory is devoted to the fusing of several values into a unique output that summarizes the given information. Typically, the aggregation process is formalized in terms of an increasing mathematical function that maps the input values to the result, fulfilling some boundary conditions. However, this formalization can be too restrictive for some scenarios. In some cases, the inputs can be seen as observations of random variables, the aggregation result being also a random variable. In others, the aggregation process can be identified as a program that performs the aggregation rather than a mathematical function. In this direction, the concepts of aggregation of random variables and computable aggregation have been defined in the literature. This paper is devoted to the definition of computable aggregation of random variables, which are computer programs, not functions, that aggregate random variables, not numbers. Special attention is given to different possible alternatives to modelize random variables and monotonicity. The implementation of some examples is also provided.

## 1. Introduction

Aggregation theory is a central topic of study in soft computing. The idea behind aggregation theory is to summarize the information of several elements or inputs to a fused output element. Classically, these processes have been modelized by means of aggregation functions [7], which are increasing functions that fulfill some boundary conditions. This concept was initially defined for numbers on the unit interval, then for real numbers, and subsequently for bounded lattices.

In many cases, the aggregation functions are applied over empirical measurements associated to the quantities of interest. In this way, it is quite reasonable to consider the input and output of the aggregation as random variables, which is the typical assumption in Statistics [20]. In this direction, the aggregation of random variables was introduced in [1], extending the monotonicity and the boundary conditions in terms of stochastic orders.

Computable aggregations were first considered in [16], replacing the mathematical function with a computer program that performs the aggregation. This approach is relevant because it allows one to modelize aggregation processes that cannot be expressed

* Corresponding author.
*E-mail addresses:* bazjuan@uniovi.es (J. Baz), sirene@uniovi.es (I. Díaz), lgarmend@fdi.ucm.es (L. Garmendia), dagomez@estad.ucm.es (D. Gómez), luis.magdalena@upm.es (L. Magdalena), montes@uniovi.es (S. Montes).

https://doi.org/10.1016/j.ins.2023.119842
Received 6 July 2023; Received in revised form 2 October 2023; Accepted 30 October 2023

Available online 7 November 2023
0020-0255/© 2023 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

by means of a mathematical function (see [14]), and consequently providing a more general concept of aggregation. In addition, it permits the study of computational properties that are directly related with the implementation of such programs.

The aim of this paper is to merge both concepts in order to define computable aggregations of random variables. Given an Abstract Data Type (ADT) associated with random variables, we aim to have programs that take several objects of the considered ADT and return an object of the same ADT, fulfilling the monotonicity and boundary conditions related to aggregations of random variables.

This approach can be specially useful in the field of computational statistics. Let us give two examples in this regard. Firstly, real-time estimation is a part of Statistics that deals with the estimation of a quantity when we are adding new observations to the dataset, see [19,21]. In this field, both probabilistic and computational properties (mainly the recursivity) are important. Many internal aggregation functions can be used to estimate location parameters, see [2] for an example. In this direction, computable aggregations of random variables allow to study the recursivity (see [12]) and also to work with probability properties, thus they are relevant in the context of real-time estimation. Secondly, M-estimators are functions of random variables that are defined in terms of a minimization problem [5]. This type of problems are related with penalty-based aggregations [4]. There are cases in which do not exist a closed expression of the functions, thus a numerical computation should be performed in order to obtain the result.

The conjunction of these two concepts (computable aggregations and aggregation of random variables) is not straightforward. On the one hand, aggregation of random variables is always linked to a fixed probability space, which is, in most of the cases, not countable. In addition, the definition of aggregation of random variables is not very restrictive, allowing randomness also in the aggregation process or the application of functions which are not directly related with the observations of the random variables such as the expected value. On the other hand, computable aggregations, since they are computer programs, cannot handle to work with infinite structures. At most, infinite structures can be approximated by finite ones. Even in the finite case, the storage space should be considered. In addition, the concept of computable aggregation considers the aggregated elements to be separated objects of the same ADT, while in aggregation of random variables, there is a common structure, the probability space, that allows dependence between them.

In order to overcome these problems, it is necessary to study the type of random variables, how they are related and modeled and the aggregation of random variables of interest. In particular, a first approach considering induced aggregations of independent discrete random variables, working with its distribution functions, is presented. This approach will be the basis for more general frameworks.

It is important to emphasize that the examples proposed in this article have been presented in the Python programming language, but any other programming language could have been used since all considerations are valid for any programming language.

The remainder of the paper is structured as follows. In Section 2, different concepts about aggregation that appear in the literature are presented. We devote Section 3 to the study of the modelization of the random variables, and its benefits and drawbacks for our purposes. Section 4 is focused on the definition of computable aggregations of random variables. We end with the conclusions and final remarks in Section 5.

## 2. Preliminary notions about aggregation

In our search for a definition of computable aggregation of random variables, we devote this section to introducing the main concepts regarding classical aggregation functions, aggregation of random variables, and computable aggregation.

### 2.1. Aggregation functions

Aggregation functions are functions that receive several values in a real interval and return a new value in the same interval, fulfilling some boundary properties and monotonicity. They are relevant in several areas such as decision theory [15,26], fusion of predictions [18,23] or image analysis, and are the basis of fuzzy set theory [27]. Classically, only the unit interval was considered:

**Definition 1.** [7] An aggregation function is a function $A : [0,1]^n \to [0,1]$ satisfying:

- It is non-decreasing (in each variable).
- $A(0,\dots,0) = 0$ and $A(1,\dots,1) = 1$

However, the concept of aggregation function can be extended to any real interval, bounded or not, as follows:

**Definition 2.** [7] Let $I$ be an interval in the real line $\mathbb{R}$. An aggregation function is a function $A : I^n \to I$ satisfying:

- It is non decreasing (in each variable).
- The following boundary conditions are met:

$$\inf_{\vec{x}\in I^n} A(\vec{x}) = \inf I, \ \sup_{\vec{x}\in I^n} A(\vec{x}) = \sup I$$

The case of $I = \mathbb{R}$ is especially relevant when considering random data, since many classical random variables have unbounded support, for instance, the Gaussian distribution [20].

### 2.2. Aggregation of random variables

Aggregation functions are usually used to fuse data [9]. Following the usual approach in Statistics, these data can be modeled using random variables. In this direction, the concept of aggregation of random variables was defined in [1], extending the boundary conditions and the monotonicity using stochastic orders. Before explaining this type of aggregation, let us recall the basic notions about probability spaces, random variables, and related functions.

**Definition 3.** [20] Let $\Omega$ be the sample space set and $\Sigma \subseteq \mathbb{P}(\Omega)$ a set of subsets of $\Omega$. $\Sigma$ is said to be a sigma-algebra if the following conditions are fulfilled:

1. $\Omega \in \Sigma$
2. If $B \in \Sigma$, $\overline{B} \in \Sigma$
3. If $B_1, B_2, \cdots \in \Sigma$, then $\left(\cup_{i=1}^{\infty} B_i\right) \in \Sigma$

A function $P : \Sigma \to [0,1]$ is said to be a probability measure if it satisfies the following properties:

1. $P(B) \geq 0$ for any $B \in \Sigma$
2. $P(\Omega) = 1$
3. If $B_1, B_2, \ldots$ is a disjoint sequence of measurable sets, then:

$$P\left(\cup_{i=1}^{\infty} B_i\right) = \sum_{i=1}^{\infty} P\left(B_i\right)$$

The trio $(\Omega, \Sigma, P)$ is known as a probability space.

A random variable is a measurable function from a probability space to the real numbers, i.e., the preimage of any Borel set of $\mathbb{R}$ belongs to $\Sigma$ (see [20]). Let us now introduce the cumulative distribution, density, and probability mass functions of a random variable.

**Definition 4.** [20] Given a random variable $X$:

- Its cumulative distribution function is defined as $F(t) = P(w \in \Omega \mid X(w) \leq t)$.
- If $F(t)$ is absolutely continuous, $X$ is said to be a continuous random variable and its density function is defined as $f(t) = \frac{dF(t)}{dt}$.
- If $F(t)$ is a piecewise constant function, $X$ is said to be a discrete random variable and its probability mass function is defined as $P_X(t) = P(w \in \Omega \mid X(w) = t)$

The monotonicity condition should be redefined in order to handle random variables. In this direction, stochastic orders, partial orders defined over the equivalence classes of random variables with the same distribution function [22], are the usual approach to order random variables. The most common stochastic order is the Stochastic Dominance [22], defined as $X \leq_{FSD} Y \iff F_X(t) \leq F_Y(t)$ for any $t \in \mathbb{R}$. For random vectors, finite collections of random variables, the most usual stochastic order is the Strong Stochastic Dominance or usual stochastic order $\leq_{SSD}$ (see [11,22]), which compares probabilities over upper sets. It is equivalent to the Stochastic Dominance when working with univariate random vectors. Let us now introduce the concept of aggregation of random variables.

**Definition 5.** [1] Let $(\Omega, \Sigma, P)$ be a probability space and let $I$ be a real interval. Then, $L_I^n(\Omega)$ is defined as the set of random vectors from $(\Omega, \Sigma, P)$ with support in $I^n$, that is, $L_I^n(\Omega) = \left\{ \vec{X} : \Omega \to I^n \mid \vec{X} \text{ is measurable} \right\}$.

**Definition 6.** [1] Let $(\Omega, \Sigma, P)$ be a probability space and $I$ be a real non-empty interval. An aggregation function of random variables (with respect to the Strong Stochastic Dominance) is a function $A : L_I^n(\Omega) \to L_I(\Omega)$ satisfying:

- For any $\vec{X}, \vec{Y} \in L_I^n(\Omega)$ such that $\vec{X} \leq_{SSD} \vec{Y}$, $A(\vec{X}) \leq_{FSD} A(\vec{Y})$ (non decreasing).
- The following boundary conditions are met:

$$\inf_{\vec{X} \in L_I^n(\Omega)} A(\vec{X}) = \inf L_I(\Omega), \quad \sup_{\vec{X} \in L_I^n(\Omega)} A(\vec{X}) = \sup L_I(\Omega)$$

The Strong Stochastic Dominance can be changed to any stochastic order if necessary. It has been proved in [1] that this order allows the composition of usual aggregation functions and random vectors to be aggregations of random variables. This type of aggregation of random variables is known as induced as the following proposition states.

**Proposition 1.** *[1] Let $I$ be a real interval, and let $\hat{A} : I^n \to I$ be a measurable aggregation function. Consider the function $A : L_I^n(\Omega) \to L_I(\Omega)$ such that for any random vector $\vec{X} \in L_I^n(\Omega)$, it holds $A(\vec{X}) = \hat{A} \circ \vec{X}$. Then, $A$ is an aggregation function of random variables with respect to the Strong Stochastic Dominance.*

*The function $A$ is referred to as the aggregation function of random variables induced by $\hat{A}$.*

In general, more complex and less intuitive aggregations of random variables can be considered, for instance, those that are defined by considering a mixture of the distributions of the aggregated variables. Nevertheless, we will focus only on the case of induced aggregation of random variables, since they are the most simple case and the one that is usually considered in data analysis. As an example, in statistics the mean, median, maximum, and minimum are widely used [20], as well as OWA operators [25] as particular cases of L-statistics [8].

**Remark 1.** Note that the definition proposed in [1] interprets the process of aggregation of random variables as a function that transforms a multidimensional random variable, $\vec{X}$, into a one-dimensional variable, $A(\vec{X})$. Under this assumption, the aggregation process allows us to aggregate (among other things) $n$ random variables that have dependency among them. Therefore, the components of $\vec{X}$ should be necessarily defined in the same probability space.

However, it is important to mention that this general idea of aggregation does not match the approach of the classical theory of aggregation operators (also applies for computable aggregations). Classical aggregation considers aggregation as a process in which we aggregate a vector of $n$ objects into another object of the same nature, that is, an aggregation should be a function of $n$ random variables $A(X_1, \dots, X_n)$ instead of a function that takes a $n$-dimensional random variable as input.

Notice that the distribution of $\vec{X}$ cannot be determined univocally by the distribution of its components $X_1, \dots, X_n$ [17] because it is necessary the underlying structure of the probability space. However, if the random variables are defined in different probability spaces, then the dependence structure does not exist. If we follow the idea of classical aggregation, the inputs could be not related objects of the same type, that is, independent random variables. In this case, a common probability space can be considered as the Cartesian product of the initial probability spaces [3].

The independence framework also allows us to work directly with the marginals, which will be important in Section 3. In addition, the assumption of independence is quite common when working with data, inherited from the independence of measurements [20]. Taking into account the latter considerations, in this paper we will focus on the case of independent random variables.

### 2.3. Computable aggregations

According to Definitions 1 and 2 (and Remark 1), aggregation functions are basically mappings that transform $n$ objects defined in a certain space ($[0,1]$, $I$, $\mathbb{R}$, or even $L_I(\Omega)$) into a single object in that same space.

These definitions would restrict us to only work with aggregation processes that can be expressed in terms of a (deterministic) function, not allowing for example those aggregation processes involving sampling components. In addition, even if our aggregation process is defined as a function, in the end it will almost surely be implemented on a computer. This implementation will not change the output of the aggregation process (which should correspond to that of the mapping), but will modify the process itself, conditioning questions as important as its complexity. These two situations show how important could be to consider the aggregation process from a different point of view, the one provided by Computable aggregations.

As said before, computable aggregations were first considered in [16], replacing the mapping with a computer program performing the aggregation.

**Definition 7.** [16] **(Computable aggregation).** Let $L < T >$ be a non-empty and finite list of $n$ elements with type $T$. A **computable aggregation** is a program $P$ that transform the list $L < T >$ into an element of $T$.

This approach solves the first question mentioned above by also implementing aggregation processes that cannot be expressed by means of a mathematical function (see [14]). Moreover, being the aggregation process directly a program, that is, an implementation of the process, studying the computational properties simply involves analyzing the program.

In summary, the main idea is to go beyond the pure input-output relation and analyze the process as a whole where implementation also matters.

It is important to notice that the previous definition does not mention monotonicity or boundary conditions, so could be closer to that of fusion functions understood as a method to get an output of the same nature of the considered inputs without further restrictions. The reason for this is that these properties are closely related to the structure and properties of $T$, and consequently could only be analyzed within the specific space.

In the present paper, $T$ will be an abstract data type representing a random variable, and consequently the concepts of monotonicity and boundary conditions should be adapted to that framework.

## 3. Different alternatives to represent random variables

As it was introduced in Definition 7, one of the main requirement for a program to be a computable aggregation is that the type of elements of the input list must be the same as the type of output of the program. There are several ways to model a random variable

in a computational environment, and its choice will be particularly relevant in the definition of computable aggregation of random variables. In this section, we explore three different alternatives.

In any case, before delving into the developed representations for random variables, it must be noted that computable aggregators are formally defined as programs that simulate the aggregation process of a list of objects to produce another object of the same nature. Taking into account that we are dealing with programs that have to be executed to generate the aggregation, we have the classical limitations that present programs when we try to represent mathematical concepts that cannot be formally defined in the context of a computer or a program. Specifically, we refer to two concepts widely used in mathematics: the concept of infinity and the concept of continuity. From a computational standpoint, both concepts cannot be represented exactly, and taking into account this, we will begin in this paper with the discrete and finite random variables case, whose representation can be made reliably.

Of course we can have programs that "model" continuous random variables as a normal or an exponential or also we can have programs that try to model the concept of infinity, but obviously, these representations are made always from a discrete and finite point of view, so this is the reason why we are going to start with the discrete and finite case.

### 3.1. Representation based on its definition

The very first way to represent a random variable on a computer is with a function that implements the mathematically measurable function from a probability space to the real numbers. In particular, we need to have a fixed probability space $(\Omega, \Sigma, P)$ stored on our computer and then a computer function $(X)$ that maps any value of $\Omega$ to $\mathbb{R}$ that satisfies the measurability property. In order to illustrate this type of computational object, let us give an example concerning Bernoulli's random variable with $p = 0.4$.

**Example 1. The ADT describing a random variable by its formal definition (above), and an instance of Bernoulli random variable, with $p = 0.4$, in accordance with this ADT (below).**

```
1 #types
2 from typing import Set, Dict
3
4 Omega = set[str]
5 Sigma = set[set[str]]
6 Prob  = dict[Sigma, float]
7 X     = dict[Omega, float]
8 RV_pure = [Omega, Sigma, Prob, X]
```

```
1 p=0.4
2 omega = {'heads','tails'}
3 sigma = {{},{'heads'},{'tails'},{'heads','tails'}}
4 prob  = {{}:0,{'heads'}:p,{'tails'}:1-p,{'heads','tails'}:1}
5 x     = {{'heads'}:1,{'tails'}:0}
6 rv_pure_x = [omega, sigma, prob, x]
```

This is the most faithful representation of a random variable for implementing the function itself, but it entails several problems. The first is that the measurability condition is not easy to check in a computer program. This can be solved by considering $\Sigma$ as the parts of $\Omega$, $\mathcal{P}(\Omega)$, but it can lead to a $\sigma$-algebra with a large number of elements. Furthermore, it is not clear how to model an usual probability space as the unit interval with the Borel $\sigma$-algebra and the Lebesgue measure.

### 3.2. Representation based on its distribution

The second alternative is to represent the random variable by means of the implementation of its cumulative distribution, density, or probability mass functions. This makes sense for a wide collection of applications in which we are only interested in the distribution of the random variable, but not in the underlying structure of a measurable function from a probability space to the real numbers. In addition, it is quite simpler than in the previous case.

Between the two alternatives, the use of distribution functions allows us to have a common structure for continuous and discrete random variables. On the other hand, the density function and the probability mass function are, in some cases, easy to handle. For discrete random variables, it is quite easy to move from the distribution function and the probability mass function, but for continuous random variables, it is necessary to perform a numeric differentiation or integration [20]. Let us give an example of the same random variable as in the previous subsection implemented using its distribution function and its probability mass function.

**Example 2. ADT to model a discrete finite random variable by its mass function.**

```
1 #types
2 from typing import List, Dict
3
4 Domain = list[float]
5 Mass_function = dict[Domain, float]
6 RV_mass = [Domain, Mass_function]
```

**Example 3. A Bernoulli random variable with $p = 0.4$ by its mass function definition.**

```
# The domain can be obtained x.values()
p=0.4
domain = [0,1]
mass_function = {1:p, 0:1-p}
rv_mass = [domain, mass_function]
```

**Example 4. ADT to model a random variable by its distribution function.**

```
#types
from typing import Callable

RV_distribution_function = Callable[float,float]
```

**Example 5. A Bernoulli random variable with $p = 0.4$ by its distribution function.**

```
def rv_distribution_function (x: float ) -> float:
    y = 0
    if 0 <= x < 1:
        y = 0.4
    elif x > 1:
        y = 1
    return y

# visualizing
import matplotlib.pyplot as plt
import numpy as np
xline =np.linspace(-1, 2)
yline = np.array([rv_distribution_function(xi) for xi in xline])
plt.plot(xline, yline)
```

The main drawback of this representation is that we are further to the initial concept of random variable; in fact, there exist different random variables with the same distribution function. Another problem is that the possible dependence between some random variables cannot be considered just by specifying their distribution functions.

### *3.3. Representation based on its simulation*

The last alternative is to identify the random variable with a generator of pseudo-random numbers. This case is the farthest from the initial concept of a random variable. We do not have the implementation of the random variable, not even of its cumulative distribution, density or probability mass functions, we have just a way to obtain realizations of the aforementioned random variable.

However, in some cases, this approach suffices for applied purposes. In particular, a simulation can be a good approximation of the behavior of a good number of random systems [6,10]. Furthermore, the distribution of a combination of random variables is generally not easy to compute explicitly. We want to remark that the distribution function of the associated random variable is assured to be able to be approximated by realizations of the generator by the convergence of the empirical distribution function [24]. We end this section by providing an example analogous to Examples 1 and 4 but with this modelization. (See Fig. 1.)

**Example 6. ADT to model a random variable by its simulation.**

```
#types
from typing import Callable

RV_empirical = Callable [None,float]
#....  note that the float is in a given domain Dx
```

**Example 7. A Bernoulli random variable with $p = 0.4$ example by its empirical simulation.**

```
import random
p = 0.4
def rv_empirical_bern() -> int:
    if random.random() < p:
        return 0
    return 1
```

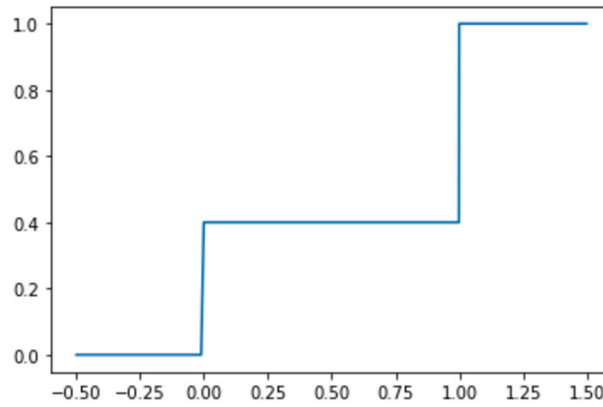



**Fig. 1.** Distribution function in Example 7.

### *3.4. Discussion about the representation for computable aggregations of random variables*

In the definition of computable aggregation of random variables, how we represent the random variables is crucial. Subsequently, a discussion on the viability of the latter three alternatives is provided.

The first alternative is representing the random variable by its definition. This approach has the benefit of implementing the random variable as its mathematical definition. Furthermore, the dependence between the input random variables of the input of the aggregation can be taken into account. Finally, usual aggregation functions are easily implemented with this modelization, we just need to apply them element by element in the sample space. On the other hand, the number of elements in the $\sigma$-algebra can increase exponentially with the number of elements in the sample space. Therefore, storing the sample space, the $\sigma$-algebra and the probability measure can be unaffordable for sample spaces with large cardinality in terms of memory. Moreover, the case of continuous probability spaces seems intractable. In conclusion, although the representation of random variables by their definition possesses good properties, we have currently set aside this approach, at least for the moment.

Another alternative is to represent the random variables by a generator of pseudo-random numbers. It has some advantages, such as low memory requirement and an easy way to implement computable aggregation. However, we would want to impose a monotonicity condition to the computable aggregation of random variables, and there is no natural manner to define orders between generators of pseudo-random numbers. A possible approach may be to execute the generators a certain number of times and then use statistical criteria to order them, but the conclusion may change between realizations, thus the resultant order will be non-deterministic. In conclusion, this approach seems interesting, but a proper way to order generators of pseudo-random numbers must be firstly established.

In our view, the representation by its distribution appears to be the most suitable option to consider when defining a computable aggregation of random variables. It avoids the memory problems that the first representation has, and the stochastic orders can be used to define the monotonicity. The main drawback is that, in general, the distribution of an aggregation function given the distribution of the inputs is not easy to compute. Another issue is the impossibility of introducing the dependence between the random variables in the input of the computable aggregation.

## 4. Computable aggregations of random variables

This section focuses on the definition of computable aggregations of random variables. Taking into account previous considerations and with the aim of formally defining a computable aggregation of random variables, we will focus on the representation of a random variable $X$ based on its distribution function $F_X$ or its density function $f_X$.

Let us observe that, for each of these two cases, we are going to use different Abstract Data Types (ADT) (denoted as $X^F$ and $X^f$) that correspond, respectively, to the use of distribution functions and density/mass to represent a random variable $X$. In conclusion, a computable aggregation of random variables is a computable aggregation in which the considered ADT is one between the distribution or density/mass function.

### *4.1. Monotonicity and boundary conditions*

As said before, Definition 7 does not mention monotonicity or boundary conditions since these properties are closely related to the structure and properties of $T$, and consequently they could only be analyzed within the specific space. In the present paper, once defined $T$ as $L_I(\Omega)$, the concepts of monotonicity and boundary conditions will be adapted to this framework.

In order to study the monotonicity of a computable aggregation $P$, it is very important to distinguish between deterministic and non deterministic computable aggregations (see for more details [14]). Due to the reasons previously mentioned, in this paper we will focus on the deterministic case. Consequently, given a list $l \in < T = L_I(\Omega) >$ of random variables, $l = (X_1, \ldots, X_n)$, the output of the computable aggregation applied to $l$, $P(l)$, will always produce the same random variable as result (i.e., the aggregated value

does not change for different executions of the program $P$). Taking into account this and Propositions 1 and 3 in [14], it is possible to find a function $A : L_I(\Omega) \times \dots \times L_I(\Omega) \longrightarrow L_I(\Omega)$, such that for any list of random variables $l$, $P(l) = P(X_1, \dots, X_n) = A(X_1, \dots, X_n)$, so monotonicity and boundary conditions can be directly translated to the monotonicity studied in [1] for the case of independent random variables.

### 4.2. Implementing the computable aggregation

Once monotonicity and boundary conditions have been defined for computable aggregations of random variables, in this section we provide the algorithm framework to adapt the aggregation of random variables defined in [1] to the context of computable aggregators. We want to remark that, as a consequence of Proposition 1, the monotonicity and boundary conditions are hold for all the here-presented computable aggregations for random variables.

Given an aggregation function $Ag : R^n \longrightarrow R$ and a list of random variables $X_1, \dots, X_n$, it is necessary to design a program that from a list of ADT $\left\{ X_1^f, \dots, X_n^f \right\}$ produces as output an Abstract Data Type $X_{Ag}^f$ that corresponds to the random variable $X_{Ag}$. By $X_{Ag}$ we are denoting the random variable associated with the process of aggregating the random variables $X_1, \dots, X_n$ by means of the $Ag$ function.

It is important to note that formally, the function $Ag$ is a function that goes from $R^n \longrightarrow R$, so if $X_1, \dots, X_n$ are random variables with $X_i : \Omega \longrightarrow R$ the formula $Ag(X_1, \dots, X_n)$ is not formally defined (or should be understood as an abuse of notation).

Assuming that all random variables come from the same probability space $\Omega$, it is possible to define the random variable $X_{Ag}$ as a function $X_{Ag} : \Omega^n \longrightarrow R$, where $X_{Ag}(\omega_1 \dots, \omega_n) = Ag(X_1(\omega_1), \dots, X_n(\omega_n))$. From now on, we will assume this notation.

**Example 8.** Taking into account previous considerations, the random variable $X_{Ag}$ associated with the aggregation function $Ag(u, v) = (u + v)/2$ for the Bernoulli random variables $X_1, X_2$ defined in $\Omega = \{c, x\}$ as $X_1(c) = X_2(c) = 1$ and $X_1(x) = X_2(x) = 0$, and with $P(X_1 = 1) = p_1$, $P(X_2 = 1) = p_2$ could be denoted as

$$X_{Ag} == Ag(X_1, X_2) = (X_1 + X_2)/2$$

formally defined as $X_{Ag} : \Omega^2 \longrightarrow R$:

$$X_{Ag}(c, c) = (X_1(c) + X_2(c))/2 = 1,$$
$$X_{Ag}(c, x) = (X_1(c) + X_2(x))/2 = 0.5,$$
$$X_{Ag}(x, c) = (X_1(x) + X_2(c))/2 = 0.5,$$
$$X_{Ag}(x, x) = (X_1(x) + X_2(x))/2 = 0.$$

So, the support of this random variable is $D_{X_{Ag}} = \{0, 0.5, 1\}$ and the mass function can be computed assuming the independence among $X_1$ and $X_2$ as follows:

$$f_{X_{Ag}}(0) = P(X_1 = 0, X_2 = 0) = P(X_1 = 0)P(X_2 = 0) = (1 - p_1)(1 - p_2),$$
$$f_{X_{Ag}}(0.5) = P(X_1 = 1, X_2 = 0) + P(X_1 = 0, X_2 = 1) = p_1(1 - p_2) + (1 - p_1)p_2 \text{ and}$$
$$f_{X_{Ag}}(1) = P(X_1 = 1, X_2 = 1) = p_1 p_2.$$

From this example we can observe two things that should be noted: The first is that, given a set of random variables $X_1, \dots, X_n$ with the same support $D_x$, the support of the random aggregated variable $D_{X_{Ag}}$ could be different (in this case, we go from $D_x = \{0, 1\}$ to $D_{X_{Ag}} = \{0, 0.5, 1\}$). For idempotent aggregations, it is assured that if $D_x \subseteq [a, b]$ for a real interval $[a, b]$, then $D_{X_{Ag}} \subseteq [a, b]$. The second one is that, from a computational point of view, the aggregation function defined for the aggregation of some $n$ particular random variables only needs to be defined from $D_x^n$ to $R$ (instead of a general function from $R^n$ to $R$). This fact, is especially relevant from a computational point of view and especially interesting in the case of discrete random variables, since the aggregation function could be implemented in an efficient way (not necessarily as an explicit function) as, for example, with a dictionary.

Once all of these considerations have been made, the problem of defining a computable aggregation of random variables is equivalent to the following two problems:

- Given the density/mass functions $f_1, \dots, f_n$ that correspond to the random variables $X_1, \dots, X_n$ and given $Ag : D_x^n \longrightarrow R$ aggregation function, we have to design an algorithmic procedure or program that builds the density/mass function $f_{Ag}$ associated with the random variable $X_{Ag}$.
- Given the distribution functions $F_1, \dots, F_n$ that correspond to the random variables $X_1, \dots, X_n$ and given $Ag : D_x^n \longrightarrow R$ aggregation function, we have to design an algorithmic procedure or program that builds the distribution function $F_{X_{Ag}}$ associated to $X_{Ag}$.

**Example 9.** Let $X_1$, $X_2$ be two Bernoulli variables with probability $p_1$ and $p_2$ respectively. And let $Ag_1 : D = \{0, 1\}^2 \subset R^2 \longrightarrow R$ be the average aggregator, i.e. $Ag_1(x_1, x_2) = \frac{(x_1 + x_2)}{2}$ and let $Ag_2 : D = \{0, 1\}^2 \subset R^2 \longrightarrow R$ be the maximum aggregator, i.e. $Ag_2(x_1, x_2) = max\{ x_1, x_2\}$.

Since $Ag_1(0,0)=0$; $Ag_1(1,0)=0.5$, $Ag_1(0,1)=0.5$, $Ag_1(1,1)=1$, it is easy to see that the support of $X_{Ag_1}$ is $\{0,0.5,1\}$ and the mass function can be described as follows:

- $f_{X_{Ag_1}}(0)=f_{X_1}(0)f_{X_2}(0)=(1-p_1)(1-p_2)$.
- $f_{X_{Ag_1}}(0.5)=f_{X_1}(0)f_{X_2}(1)+f_{X_1}(1)f_{X_2}(0)=(1-p_1)(p_2)+(p_1)(1-p_2)$.
- $f_{X_{Ag_1}}(1)=f_{X_1}(1)f_{X_2}(1)=p_1p_2$.

Since $Ag_2(0,0)=0$; $Ag_2(1,0)=1$, $Ag_2(0,1)=1$, $Ag_2(1,1)=1$, it is easy to see that the support of $X_{Ag_2}$ is now $\{0,1\}$ and the mass function can be described as follows:

- $f_{X_{Ag_2}}(0)=f_{X_1}(0)f_{X_2}(0)=(1-p_1)(1-p_2)$.
- $f_{X_{Ag_2}}(1)=f_{X_1}(0)f_{X_2}(1)+f_{X_1}(1)f_{X_2}(0)+f_{X_1}(1)f_{X_2}(1)=1-(1-p_1)(1-p_2)$.

Let us first formalize the aggregation of $n$ random variables and then analyze it from an algorithmic point of view, defining a procedure to deal with the computable aggregation in the discrete case. Notice that, in this case, the composition of any aggregation function and the random variables is always a measurable function, thus a (discrete) random variable.

**Definition 8.** Let $X_1,\dots,X_n$ be $n$ discrete random variables with support $D_x$ and described in terms of their mass functions $\left(X_1^f,\dots,X_n^f\right)$, let $Ag$ be an aggregation function $Ag: D_x^n \longrightarrow R$, and let $U$ be the set of aggregated values ($U=\{u\in R \mid \exists(x_1,\dots,x_n)\in D_x^n$ with $Ag(x_1,\dots,x_n)=u\}$). We compute the aggregated variable $X_{Ag}$ described in terms of its mass function as:

$$f_{X_{Ag}}: U \longrightarrow [0,1]$$

with

$$f_{X_{Ag}}(u)=\sum_{(x_1,\dots,x_n)\ /\ Ag(x_1,\dots,x_n)=u}\prod_{k=1}^{n} f_{X_k}(x_k).$$

From an algorithmic point of view this function can be computed in different ways, and it is quite important to notice that in this discrete case, the set of aggregated values ($U$) may not be known in advance, and may differ from $D_x$.

A first option for this algorithm is the following one.

**Algorithm 1** *(Aggregate through mass functions).*

**Input:** A list of ADT $\left(X_1^f,\dots,X_n^f\right)$ and the aggregation function ($Ag$).

**Step 1:** Obtain $U$ from $D_x$ and $Ag$ ($D_x$ should be equal for all variables but we can generalize this considering different domains.).

- Initialize $U$ as empty,
- For each $x_1$ in $D_x$, for each $x_2$ in $D_x$, …, for each $x_n$ in $D_x$ do:
  Compute the aggregated value for the corresponding input ($u=Ag(x_1,x_2,\dots,x_n)$), and if the obtained value ($u$) was not in $U$ then add it to $U$ ($U:=U+\{u\}$).

**Step 2:** Compute $f_{X_{Ag}}$ for each $u$ in $U$.

- For each $u$ in $U$ do:
  - Initialize $f_{X_{Ag}}(u):=0$
  - For each $x_1$ in $D_x$, for each $x_2$ in $D_x$, …, for each $x_n$ in $D_x$ do: if $Ag(x_1,x_2,\dots,x_n)=u$ then $f_{X_{Ag}}(u):=f_{X_{Ag}}(u)+\prod_{k=1}^{n} f_{X_k}(x_k)$.

**Output:** An Abstract Data Type $X_{Ag}^f$ representing the aggregated random variable by means of a mass function $f_{X_{Ag}}: U \longrightarrow [0,1]$.

**Example 10. Code to implement aggregation through mass functions, Algorithm 1.**

```
1  #types
2  from typing import *
3
4  Omega = set[str]
5  RV = dict[Omega, float]
6  RV_domain = list[float]
7  RV_mass_function = dict[float,float]
8  #.... note that the dict.keys() is a given RV_domain
9
10 Agg = Callable [list[float],float]
11 AggRV = Callable [list[RV],RV]
```

```
12
13 def agg(list: list[float]) -> float:
14     val = 0
15     #....  the agg code
16     return  val
17
18 # RV Computable  Aggregation  based  on  agg  function
19
20 import itertools
21
22 def aggRV(rvList:list[RV_mass_function]) -> RV_mass_function:
23     # Computing  the  cartesian  product  of  the  rv  domains
24     keysList = []
25     for keys in  rvList:
26         keysList.append(keys)
27     # The cart. prod. of the rv domains is itertools.product(*keysList)
28
29     #  Step 1. Computing  the aggregated RV domain
30     domain = []
31     for dom in itertools.product(*keysList):
32         if agg(dom) not in domain:
33             domain.append(agg(dom))
34     print("Domain is " + str(list(domain)) )
35
36     # Step 2. Computing the aggregated RV mass function
37     rvAgg = {}
38     for i in domain:
39         sum = 0
40         for dom in itertools.product(*keysList):
41             if agg(dom) == i:
42                 prod = 1
43                 for index in range(len(dom)):
44                     prod *= rvList[index][dom[index]]
45                 sum += prod
46         rvAgg[i] = sum
47     return rvAgg
```

It is clear that with this approach, we have to go through $D_x^n$ as many times as $|U| + 1$. The first to obtain $U$, plus one for each element in $U$. The following examples show the program to obtain the computable aggregation of different random variables using max, min or mean aggregation functions.

**Example 11. A computable aggregation based on the max, min and mean aggregation functions of three Bernoulli random variables.**

```
1 # Some agg callable functions
2
3 def max (list):
4     m = 0
5     for i in list:
6         if i > m:
7             m = i
8     return m
9
10 def min (list):
11     m = 10000
12     for i in list:
13         if i < m:
14             m = i
15     return m
16
17 def avg (list):
18     m = 0
19     for i in list:
20         m += i
21     return m/len(list)
22
23 # printing functions
24
25 def printRV(rv):
26     for i in rv.keys():
27         print(round(i,2),'->', round(rv[i],2))
```

```
28
29 def printRVList(rvList):
30     for rv in rvList:
31         printRV(rv)
32         print()
```

```
1  # Executing a computable aggregation of a list of three bernoullis with several agg
2
3  rvBernuilli_mass_function1 = {0:1/2, 1:1/2}
4  rvBernuilli_mass_function2 = {0:0.4, 1:0.6}
5  rvBernuilli_mass_function3 = {0:0.7, 1:0.3}
6
7  rvList = [rvBernuilli_mass_function1  \
8            ,rvBernuilli_mass_function2 \
9            ,rvBernuilli_mass_function3]
10
11 print('\nRandom variables')
12 printRVList(rvList)
13 print('Random variable Min')
14 agg = min
15 printRV(aggRV(rvList))
16 print('\nRandom variable Arithmetc Mean')
17 agg = avg
18 printRV(aggRV(rvList))
19 print('\nRandom variable Max')
20 agg = max
21 printRV(aggRV(rvList))
22
23 '''
24 The output is:
25
26 Random variables
27 0 -> 0.5
28 1 -> 0.5
29
30 0 -> 0.4
31 1 -> 0.6
32
33 0 -> 0.7
34 1 -> 0.3
35
36 Random variable Min
37 Domain is [0, 1]
38 0 -> 0.91
39 1 -> 0.09
40
41 Random variable Arithmetc Mean
42 Domain is [0.0, 0.3333333333333333, 0.6666666666666666, 1.0]
43 0.0 -> 0.14
44 0.33 -> 0.41
45 0.67 -> 0.36
46 1.0 -> 0.09
47
48 Random variable Max
49 Domain is [0, 1]
50 0 -> 0.14
51 1 -> 0.86
```

**Example 12. Execution of three different computable aggregation based on Min, Max and Average for a list of $n$ equal Random Variables with $Dx = \{1, \dots, 6\}$**

```
1  # Example aggregating 3 rv Dice
2
3  n = 3
4  omega = set(['1','2','3','4','5','6'])
5  rvDice = {'1':1,'2':2,'3':3, '4':4, '5':5, '6':6}
6  rvDice_mass_function = {1:1/6, 2:1/6, 3:1/6, 4:1/6, 5:1/6, 6:1/6}
7
8  from random import random
9  rvList = []
10 for i in range(n):
11     rvList.append(rvDice_mass_function)
```

```
12
13 print('\nRandom variables')
14 printRV(rvDice_mass_function)
15 print()
16 print('Random variable Min')
17 agg = min
18 printRV(aggRV(rvList))
19 print('\nRandom variable Arithmetc Mean')
20 agg = avg
21 printRV(aggRV(rvList))
22 print('\nRandom variable Max')
23 agg = max
24 printRV(aggRV(rvList))
25
26 '''
27 The output is:
28
29 Random variables
30 1 -> 0.17
31 2 -> 0.17
32 3 -> 0.17
33 4 -> 0.17
34 5 -> 0.17
35 6 -> 0.17
36
37 Random variable Min
38 Domain is [1, 2, 3, 4, 5, 6]
39 1 -> 0.42
40 2 -> 0.28
41 3 -> 0.17
42 4 -> 0.09
43 5 -> 0.03
44 6 -> 0.0
45
46 Random variable Arithmetc Average
47 Domain is [1.0, 1.3333333333333333, 1.6666666666666667, 2.0,
48            2.3333333333333335, 2.6666666666666665, 3.0,
49            3.3333333333333335, 3.6666666666666665, 4.0,
50            4.333333333333333, 4.666666666666667, 5.0,
51            5.333333333333333, 5.666666666666667, 6.0]
52 1.0 -> 0.0
53 1.33 -> 0.01
54 1.67 -> 0.03
55 2.0 -> 0.05
56 2.33 -> 0.07
57 2.67 -> 0.1
58 3.0 -> 0.12
59 3.33 -> 0.12
60 3.67 -> 0.12
61 4.0 -> 0.12
62 4.33 -> 0.1
63 4.67 -> 0.07
64 5.0 -> 0.05
65 5.33 -> 0.03
66 5.67 -> 0.01
67 6.0 -> 0.0
68
69 Random variable Max
70 Domain is [1, 2, 3, 4, 5, 6]
71 1 -> 0.0
72 2 -> 0.03
73 3 -> 0.09
74 4 -> 0.17
75 5 -> 0.28
76 6 -> 0.42
77 '''
```

Let us consider now an alternative algorithm where Step 2 is completed in a single process for all elements in $U$.

**Algorithm 2** *(Aggregate through mass functions, option 2).*

**Input:** A list of ADT $\left(X_1^f, \ldots, X_n^f\right)$ and the aggregation function ($Ag$).

**Step 1:** Obtain $U$ from $D_x$ and $Ag$.

- Initialize $U$ as empty,
- For each $x_1$ in $D_x$, for each $x_2$ in $D_x$, …, for each $x_n$ in $D_x$ do: Compute the aggregated value for the corresponding input ($u = Ag(x_1, x_2, \ldots, x_n)$), and if the obtained value ($u$) was not in $U$ then add it to $U$ ($U := U + \{u\}$).
- Initialize $f_{X_{Ag}}$. For each $u$ in $U$ do $f_{X_{Ag}}(u) := 0$.

**Step 2:** Compute $f_{X_{Ag}}$ in a single process for every $u$ in $U$.
For each $x_1$ in $D_x$, for each $x_2$ in $D_x$, …, for each $x_n$ in $D_x$ do:

- Assign $u := Ag(x_1, x_2, \ldots, x_n)$.
- Update the mass function for $u$: $f_{X_{Ag}}(u) := f_{X_{Ag}}(u) + \prod_{k=1}^{n} f_{X_k}(x_k)$.

**Output:** An Abstract Data Type $X^f_{Ag}$ representing the aggregated random variable by means of a mass function $f_{X_{Ag}} : U \longrightarrow [0, 1]$.

**Example 13** *(Implementation of Algorithm 2.)*. This algorithm walks along the cartesian product of the supports of the input random variables twice: Once to compute the aggregated random variable support and another one to compute their probabilities. For this new interpretation, we can simply replace in the previous code the definition of *aggRV* with the following one:

```
1  agg = min #assing any aggregation operator
2  import itertools
3  def aggRV(rvList:list[RV_mass_function]) -> RV_mass_function:
4      # Computing a list of rv domains
5      keysList = []
6      for keys in  rvList:
7          keysList.append(keys)
8
9      # Computing the aggregated rv domain
10     domain =[]
11     for dom in itertools.product(*keysList):
12         if agg(dom) not in domain:
13             domain.append(agg(dom))
14     print("Domain is " + str(list(domain)) )
15
16     # Computing the aggregated rv mass function
17     rvAgg= {}
18     for dom in itertools.product(*keysList):
19         prod = 1
20         ag = agg(dom)
21         for index in range(len(dom)):
22             prod *= rvList[index][dom[index]]
23         if ag in rvAgg.keys():
24             rvAgg[ag] += prod
25         else:
26             rvAgg[ag] = prod
27     return rvAgg
```

And finally we can even reduce the whole process to a single step where $U$ is updated while discovering it.

**Algorithm 3** *(Aggregate through mass functions, option 3)*.
**Input:** A list of ADT $\left(X^f_1, \ldots, X^f_n\right)$ and the aggregation function ($Ag$).
**Step 1:** Obtain $f_{X_{Ag}}$ from $D_x$, $[X^f_1, \ldots, X^f_n]$, and $Ag$.

- Initialize $U$ as empty.
- For each $x_1$ in $D_x$, for each $x_2$ in $D_x$, …, for each $x_n$ in $D_x$ do:
  - Compute the aggregated value for the corresponding input ($u := Ag(x_1, x_2, \ldots, x_n)$).
  - If the obtained value ($u$) was not in $U$ then add it to $U$ ($U := U + \{u\}$) and initialize its mass function by doing $f_{X_{Ag}}(u) := \prod_{k=1}^{n} f_{X_k}(x_k)$; else update the mass function as $f_{X_{Ag}}(u) := f_{X_{Ag}}(u) + \prod_{k=1}^{n} f_{X_k}(x_k)$.

**Output:** An Abstract Data Type $X^f_{Ag}$ representing the aggregated random variable by means of a mass function $f_{X_{Ag}} : U \longrightarrow [0, 1]$.

**Example 14** *(Implementation of Algorithm 3)*. This algorithm walks along the cartesian product of the supports of the input random variables just once to compute the aggregated random variable support keys and at the same time setting or adding to finally compute their probabilities.

```
1  def aggRV(rvList:list[RV_mass_function]) -> RV_mass_function:
2      # Computing a list of rv domains
3      keysList = []
4      for keys in  rvList:
5          keysList.append(keys)
6      rvAgg= {}
7      # Computing the aggregated rvAgg mass function
8      for dom in itertools.product(*keysList):
9          prod = 1
10         ag = agg(dom)
11         for index in range(len(dom)):
12             prod *= rvList[index][dom[index]]
13         if ag in rvAgg.keys():
14             rvAgg[ag] += prod
15         else:
16             rvAgg[ag] = prod
17     return rvAgg
```

We end this section by remarking that for the particular cases of the maximum and minimum, the aggregation can be easily implemented by considering the well-known relationship between the distribution functions. In particular, if $F_1, \dots, F_n$ are the distribution functions of the independent random variables $X_1, \dots, X_n$ and $F_{\max}$ and $F_{\min}$ are the distribution functions of their maximum and minimum, one has that:

$$F_{\max} = \prod_{i=1}^{n} F_i, \qquad F_{\min} = 1 - \prod_{i=1}^{n} (1 - F_i)$$

## 5. Conclusion and final remarks

In this work, the concept of computable aggregation is extended to the case of random variables. Computable aggregations were initially introduced to aggregate real numbers by replacing the mathematical function defining the aggregation, with a program that performs the aggregation process [16]. There are different reasons for justifying this extension. First of all, it is possible to deal with more complex and realistic aggregation process that cannot be represented by means of mathematical functions. Second, modeling an aggregation process by means of its implementation allows us to explore some computational properties not directly related to the aggregation itself but to its implementation (recursivity, complexity, parallelization, etc.) [12,13].

The extension to random variables represents an important challenge from both applications and computational points of view, since the modeling/representation of a random variable as the input of a program is not a trivial process. The importance of aggregation processes over random variables was introduced with the idea of modeling at least those situations in which the information to be aggregated is obtained as a measurement process over a sample population. In the classical approach, the aggregation function can be seen as a function that, given a vector/set of random variables [1], returns a random variable as the result of the aggregation process. In order to extend the concept of computable aggregation to this case, we propose four scenarios and representations: aggregate random variables based on their density functions, their distribution functions, their simulation functions, or the concept of random variable itself.

Focusing on the case of representation by mass functions of discrete random variables, we have proposed three different algorithms that make it possible to formalize a computable aggregation of random variables in the discrete case in a general way.

It is important to emphasize that one of the advantages provided by this new approach is that it allows us to explore different properties associated with aggregation processes that could not be easily explored from a functional definition of the aggregation process. In this regard, one of the most relevant properties is that related to algorithmic complexity associated with computational aggregators from all aspects: temporal complexity (commonly known as algorithmic complexity, referring to the computational time of the algorithm/procedure), memory complexity, and spatial complexity (storage).

The first of these, enabling the classification of computational aggregators according to their complexity, in the same line as the aggregation procedures were formalized based on their algorithmic complexity, addresses one of the most significant aspects within what is known as “green algorithms”, which is the ability to classify algorithms based on their complexity so that we can distinguish between “sustainable” and non-sustainable algorithms. For example, among the four proposed modeling approaches for random variables (based on their density functions, distribution functions, simulation functions, or the concept of the random variable itself), it is clear that the last one presents some computational issues in terms of information storage and consequently, very likely, in terms of algorithmic complexity. Similarly, the three algorithms presented can be arranged to obtain the density/mass function of the aggregated variable from the mass functions of the variables to be aggregated.

We want to remark two relevant points about the usage of the Phyton programming language. On one hand, the advantage of representing the mass functions by means of dictionaries instead of functions for the discrete case, which improves the efficiency and the calculation of the inverses; on the other hand, and although Phyton is an untyped programming language, random variables have been represented as lists or sets of typed variables, defining the aggregation process as a program that takes a list of a specific abstract data type and produces as output an object of the same type. In this way, the computable aggregation process represents the same “spirit” of classical aggregation process that takes a list of elements for generating an element of the same class of the input list.

It is also important to mention that we have studied in a first approach, the simplest case: discrete random independent variables and induced aggregations of random variables, in which the mass function of the aggregated random variable can be obtained as the sum of the product which is an operation that can be easily implemented in any programming language. Additionally, the boundary and monotonicity conditions with respect to the Strong Stochastic Dominance are assured to be fulfilled by Proposition 1.

How to extend the problem to the continuous case is a question that deserves to be explored in a future since it could be done in an approximated way (integrals can be approximated from a computational point of view by sums) or if the program that we are using allows us to deal with symbolic programming or integrals could be done directly. In addition, if needed, the dependence between random variables could be introduced by means of copulas [17]. In any case, the theory and implementation of these extensions will be adaptations of the here-proposed concepts and methods.

## CRediT authorship contribution statement

**Juan Baz:** Conceptualization, Formal analysis, Investigation, Methodology, Validation, Writing – original draft, Writing – review & editing. **Irene Díaz:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Writing – review & editing. **Luis Garmendia:** Conceptualization, Formal analysis, Investigation, Methodology, Resources, Software, Validation, Writing – review & editing. **Daniel Gómez:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Writing – original draft, Writing – review & editing. **Luis Magdalena:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Writing – review & editing. **Susana Montes:** Conceptualization, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

No data was used for the research described in the article.

## Acknowledgements

J. Baz is partially supported by Programa Severo Ochoa of Principality of Asturias (BP21042). L. Garmendia and D. Gómez are supported by Government of Spain (grant PID2021-122905NB-C21). L. Magdalena is supported by Government of Spain (grants PID2020-112502RB-C41 and PID2021-122905NB-C22), and Comunidad de Madrid (Convenio Plurianual con la UPM en la lınea de actuación Programa de Excelencia para el Profesorado Universitario). J. Baz, S. Montes and I. Díaz are been supported by the Ministry of Science and Innovation (PDI2022-139886NB-l00).

## References

[1] J. Baz, I. Díaz, S. Montes, The choice of an appropriate stochastic order to aggregate random variables, in: Building Bridges Between Soft and Statistical Methodologies for Data Science, Springer, 2022, pp. 40–47.
[2] J. Baz, D. García-Zamora, I. Díaz, S. Montes, L. Martínez, Flexible-dimensional EVR-OWA as mean estimator for symmetric distributions, in: International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, Springer, 2022, pp. 11–24.
[3] V.I. Bogachev, Measure Theory, vol. 1, Springer Science & Business Media, 2007.
[4] T. Calvo, G. Beliakov, Aggregation functions based on penalties, Fuzzy Sets Syst. 161 (10) (2010) 1420–1436.
[5] D. De Menezes, D.M. Prata, A.R. Secchi, J.C. Pinto, A review on robust m-estimators for regression analysis, Comput. Chem. Eng. 147 (2021) 107254.
[6] B. Ermentrout, A. Mahajan, Simulating, analyzing, and animating dynamical systems: a guide to XPPAUT for researchers and students, Appl. Mech. Rev. 56 (4) (2003) B53.
[7] M. Grabisch, J.-L. Marichal, R. Mesiar, E. Pap, Aggregation Functions, vol. 127, Cambridge University Press, 2009.
[8] J. Hosking, 8 l-estimation, in: Handbook of Statistics, vol. 17, 1998, pp. 215–235.
[9] S. James, An Introduction to Data Analysis Using Aggregation Functions in R, Springer, 2016.
[10] A. Joseph, Markov Chain Monte Carlo Methods in Quantum Field Theories: A Modern Primer, Springer Nature, 2020.
[11] M. Kopa, B. Petrová, Strong and weak multivariate first-order stochastic dominance, Available at SSRN 3144058, 2018.
[12] L. Magdalena, L. Garmendia, D. Gómez, R.G. del Campo, J.T. Rodríguez, J. Montero, Types of recursive computable aggregations, in: 2019 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE, IEEE, 2019, pp. 1–6.
[13] L. Magdalena, L. Garmendia, D. Gómez, J. Montero, Hierarchical computable aggregations, in: 2022 IEEE International Conference on Fuzzy Systems, FUZZ-IEEE, IEEE, 2022, pp. 1–8.
[14] L. Magdalena, D. Gómez, L. Garmendia, J. Montero, Analysing monotonicity in non-deterministic computable aggregations: the probabilistic case, Inf. Sci. 583 (2022) 288–305.
[15] W.R.W. Mohd, L. Abdullah, Aggregation methods in group decision making: a decade survey, Informatica 41 (1) (2017).
[16] J. Montero, R. González-del Campo, L. Garmendia, D. Gómez, J.T. Rodríguez, Computable aggregations, Inf. Sci. 460 (2018) 439–449.
[17] R.B. Nelsen, An Introduction to Copulas, Springer Science & Business Media, 2007.
[18] M.K. Nungesser, L.A. Joyce, A.D. McGuire, Effects of spatial aggregation on predictions of forest climate change response, Clim. Res. 11 (2) (1999) 109–124.
[19] C. Piao, Z. Li, S. Lu, Z. Jin, C. Cho, Analysis of real-time estimation method based on hidden Markov models for battery system states of health, J. Power Electron. 16 (1) (2016) 217–226.

[20] V.K. Rohatgi, An Introduction to Probability Theory and Mathematical Statistics, John Wiley and Sons, New York, 1976.
[21] R. Roughan, D. Veitch, P. Abry, Real-time estimation of the parameters of long-range dependence, IEEE/ACM Trans. Netw. 8 (4) (2000) 467–478.
[22] M. Shaked, J.G. Shanthikumar, Stochastic Orders, Springer, 2007.
[23] D. Shanmugam, D. Blalock, G. Balakrishnan, J. Guttag, Better aggregation in test-time augmentation, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 1214–1223.
[24] G.R. Shorack, J.A. Wellner, Empirical Processes with Applications to Statistics, SIAM, 2009.
[25] R.R. Yager, Families of OWA operators, Fuzzy Sets Syst. 59 (2) (1993) 125–148.
[26] A. Zahedi Khameneh, A. Kilicman, Some construction methods of aggregation operators in decision-making problems: an overview, Symmetry 12 (5) (2020) 694.
[27] H.-J. Zimmermann, Fuzzy set theory, Wiley Interdiscip. Rev.: Comput. Stat. 2 (3) (2010) 317–332.