Annals of Operations Research manuscript No.
(will be inserted by the editor)

Why Copulas Have Been Successful in Many Practical Applications: A Theoretical Explanation Based on Computational Efficiency

Vladik Kreinovich · Hung T. Nguyen · Songsak Sriboonchitta · Olga Kosheleva

Received: date / Accepted: date

Abstract A natural way to represent a 1-D probability distribution is to store its cumulative distribution function (cdf) $F(x) = \text{Prob}(X \le x)$. When several random variables $X_1, \ldots, X_n$ are independent, the corresponding cdfs $F_1(x_1)$, $\ldots$, $F_n(x_n)$ provide a complete description of their joint distribution. In practice, there is usually some dependence between the variables, so, in addition to the marginals $F_i(x_i)$, we also need to provide an additional information about the joint distribution of the given variables. It is possible to represent this joint distribution by a multi-D cdf $F(x_1, \ldots, x_n) = \text{Prob}(X_1 \le x_1 \,\&\, \ldots \,\&\, X_n \le x_n)$, but this will lead to duplication – since marginals can be reconstructed from the joint cdf – and duplication is a waste of computer space. It is therefore desirable to come up with a duplication-free representation which would still allow us to easily reconstruct $F(x_1, \ldots, x_n)$. In this paper, we prove that among all duplication-free representa-

We acknowledge the partial support of the Center of Excellence in Econometrics, Faculty of Economics, Chiang Mai University, Thailand. This work was also supported in part by the National Science Foundation grants HRD-0734825 and HRD-1242122 (Cyber-ShARE Center of Excellence) and DUE-0926721.

V. Kreinovich
Department of Computer Science, University of Texas at El Paso
500 W. University, El Paso, TX 79968, USA
Tel.: +1-915-747-6951
Fax: +1-915-747-5030
E-mail: vladik@utep.edu

H. T. Nguyen
Department of Mathematical Sciences, New Mexico State University
Las Cruces, NM 88003, USA and
Faculty of Economics, Chiang Mai University
Chiang Mai, Thailand

S. Sriboonchitta
Faculty of Economics, Chiang Mai University
Chiang Mai, Thailand

O. Kosheleva
University of Texas at El Paso, El Paso, TX 79968, USA

tions, the most computationally efficient one is a representation in which marginals are supplements by a copula.

This result explains why copulas have been successfully used in many applications of statistics: since the copula representation is, in some reasonable sense, the most computationally efficient way of representing multi-D probability distributions.

Keywords Copula · Computational complexity · Multi-D probability distribution

1 Introduction

In many practical problems, we need to deal with joint distributions of several quantities, i.e., with multi-D probability distributions. There are many different ways to represent such a distribution in a computer. In many practical applications, it turns out to be beneficial to use a representation in which we store the marginal distributions (that describe the distribution of each quantity) and a copula (that describe the relation between different quantities; definitions are given below). While this representation is, in many cases, empirically successful, this empirical success is largely a mystery.

In this paper, we provide a theoretical explanation of this empirical success, by showing that the copula representation is, in some reasonable sense, the most computationally efficient.

The structure of this paper is as follows: In Section 2, we explain why representing probability distributions is important for decision making: (consistent) decision making requires computing expected utility, and to perform this computation, we need to have the corresponding probability distribution in the computer represented in a computer. In Section 3, with this objective in mind, we consider the usual representations of probability distributions: what are the advantages and limitations of these representations. Section 3 ends with the main problem that we consider in this paper: what is the best representation of multi-D distributions? The formulation of this problem in Section 3 is informal. The problem is formalized in Section 4. This formalization enables us to prove that copulas are indeed the most effective computer representation of multi-D distributions.

Comment. This paper is an extended version of the conference paper [4].

2 Why it is important to represent probability distributions: since this is necessary for decision making

In this section, we explain why it is important to represent probability distributions in a computer.

Probability distributions are ubiquitous. To understand why representing probability distributions is important, let us recall that one of the main objectives of science and engineering is to predict the future state of the world – and to come up with decisions which lead to the most preferable future state.

These predictions are based on our knowledge of the current state of the world, and on our knowledge of how the state of the world changes with time. Our

knowledge is usually approximate and incomplete. As a result, based on our current knowledge, we cannot predict the *exact* future state of the world, *several* future states are possible based on this knowledge. What we can predict is the set of possible states, and the *frequencies* with which, in similar situations, different future states will occur. In other words, what we can product is a *probability distribution* on the set of all possible future states.

This is how many predictions are made: weather predictions give us, e.g., a 60% chance of rain; economic predictions estimate the probability of different stock prices, etc.

Need to consider random variables. Information about the world comes from measurements. As a result of each measurement, we get the values of the corresponding physical quantities. Thus, a natural way to describe the state of the world is to list the values of the corresponding quantities $X_1, \ldots, X_n$.

From this viewpoint, the probability distribution on the set of all possible states means a probability distribution on the set of the corresponding tuples $X = (X_1, \ldots, X_n)$.

How to represent probability distributions: an important question. Due to ubiquity of probability distributions, it is important to select an appropriate computer representation of these distributions, a representation that would allow us to effectively come up with related decisions.

Thus, to come up with the best ways to represent a probability distribution, it is important to take into account how decisions are made.

How decisions are made: a reminder. In the idealized case, when we are able to exactly predict the consequences of each possible decision, decision making is straightforward: we select a decision for which the consequences are the best possible. For example:

- an investor should select the investment that results in the largest return,
- a medical doctor should select a medicine which leads to the fastest recovery of the patient, etc.

In reality, as we have mentioned, we can rarely predict the exact consequence of different decisions; we can, at best, predict the probabilities of different consequences of each decision. In such real-life setting, it is no longer easy to select an appropriate decision. For example:

- if we invest money in the US government bonds, we get a small guaranteed return;
- alternatively, if we invest into stocks, we may get much higher returns, but we may also end up with a loss.

Similarly:

- if we prescribe a well-established medicine, a patient will slowly recover;
- if instead we prescribe a stronger experimental medicine, the patient will probably recover much faster, but there is also a chance of negative side effects which may drastically delay the patient's recovery.

Researchers have analyzed such situations. The main result of the corresponding decision theory is that a consistent decision making under such probability uncertainty can be described as follows (see, e.g., [2,6,9]):

- we assign a numerical value u (called *utility*) to each possible consequence, and then
- we select a decision for which the expected value $E[u]$ of utility is the largest possible.

Since we want a representation of a probability distribution that would make decision making as efficient as possible, we thus need to select a representation that would allow us to compute the expected values of different utility functions as efficiently as possible.

In the next section, we use this motivation for representing probability distributions to explain which computer representations are most adequate.

Comment. From the strictly mathematical viewpoint, every decision making is based on comparing expected values of the utility functions. However, in many practical situations, the corresponding problem becomes much simpler, because the corresponding utility functions are simple.

For example, in risk analysis, e.g., when considering whether the bridge will collapse during a hurricane, we often do not differentiate between different *positive* situations (i.e., situations in which the bridge remains standing), and we also do not differentiate between different *negative* situations (i.e., situations in which the bridge collapses). In terms of the utility function, this is equivalent to considering a "binary" utility in which we only have two utility levels $u^+ > u^-$. In such situations, the expected utility is equal to $u^+ \cdot (1-p^-) + u^- \cdot p^- = u^+ - (u^+ - u^-) \cdot p^-$, where p^- is the probability of the undesired scenario.

In such cases, comparing different values of expected utility is equivalent to comparing the corresponding probabilities p^-. Thus, instead of computing expected values of the utility function, it is sufficient to simply compute the corresponding probabilities.

This observation will be actively used in the following sections.

3 Different computer representations of probability distributions: analysis from the viewpoint of decision-making applications

Case of 1-D probability distribution: what is needed? In the previous section, we argued that since ultimately, our problem is to make a decision, and consistent decision making means comparing expected values of the utility function, it is reasonable to select such computer representations of probability distributions that would be the most efficient in computing the corresponding expected values.

Let us start with the simplest case of 1-D probability distributions. In view of the above argument, to understand which representation is the most appropriate, we need to describe possible utility functions. To describe such functions, let us start by considering a simple example: the problem of getting from point A to point B.

In general, all else being equal, we would like to get from A to B as fast as possible. So, in the idealized case, if we knew the exact driving time, we should select the route that takes the shortest time. In practice, random delays are possible, so we need to take into account the cost of different delays.

In some cases – e.g., if we drive home after a long flight – a small increase of driving time leads to a small decrease in utility. However, in other cases – e.g., if

we are driving to the airport to take a flight – a similar small delay can make us miss a flight and thus, the corresponding decrease in utility will be huge. In our analysis, we need to take into account both types of situations.

In the situations of the *first type*, utility $u(x)$ is a smooth function of the corresponding variable x. Usually, we can predict x with some accuracy, so all possible values x are located in a small vicinity of the predicted value x_0. In this vicinity, we can expand the dependence $u(x)$ in Taylor series and safely ignore higher order terms in this expansion:

$$u(x) = u(x_0) + u'(x_0) \cdot (x - x_0) + \frac{1}{2} \cdot u''(x_0) \cdot (x - x_0)^2 + \ldots$$

The expected value of this expression can be thus computed as the linear combination of the corresponding moments:

$$E[u] = u(x_0) + u'(x_0) \cdot E[x - x_0] + \frac{1}{2} \cdot u''(x_0) \cdot E[(x - x_0)^2] + \ldots$$

Thus, to deal with situations of this type, it is sufficient to know the first few moments of the corresponding probability distribution.

In situations of the *second type*, we have a threshold x_t such that the utility is high for $x \le x_t$ and low for $x > x_t$. In comparison with the difference between high and low utilities, the differences between two high utility values (or, correspondingly, between two low utility values) can be safely ignored. Thus, we can simply say that $u = u^+$ for $x \le x_t$ and $u = u^- < u^+$ for $x > x_t$. In this case, the expected value of utility is equal to $E[u] = u^- + (u^+ - u^-) \cdot F(x_t)$, where $F(x_t) = \mathrm{Prob}(x \le x_t)$ is the probability of not exceeding the threshold. So, to deal with situations of this type, we need to know the cdf $F(x)$.

1-D case: what are the most appropriate computer representations? Our analysis shows that in the 1-D case, to compute the expected utilities, we need to know the cdf *and* the moments.

Since the moments can be computed based on cdf, as

$$E\left[(x - x_0)^k\right] = \int (x - x_0)^k \, dF(x),$$

it is thus sufficient to have a cdf. From this viewpoint, the most appropriate way to represent a 1-D probability distribution in the computer is to store the values of its cumulative distribution function $F(x)$.

Multi-D case. In the multi-D cases, we similarly have two types of situations. For situations of the first type, when small changes in the values x_i lead to small changes in utility, it is sufficient to know the first few moments.

In the situations of the second type, we want all the values not to exceed appropriate thresholds. For example, we want a route in which the travel time does not exceed a certain pre-set quantity, and the overall cost of all the tolls does not exceed a certain value. To handle such situations, it is desirable to know the following probabilities – that form the corresponding multi-D cdf:

$$F(x_1, \ldots, x_n) \stackrel{\mathrm{def}}{=} \mathrm{Prob}(X_1 \le x_1 \,\&\, \ldots \,\&\, X_n \le x_n).$$

So, in the multi-D case too, computing expected values of utility functions means that we need to compute both the moments and the multi-D cdf. Since the

moments can be computed based on the cdf, it is thus sufficient to represent a cdf.

Situations when we go from 1-D to multi-D case. The above analysis of the multi-D case is appropriate for situations in which we acquire all our knowledge about the probabilities in one step:

- we start "from scratch", with no knowledge at all,
- then we gain the information about the joint probability distribution.

In such 1-step situations, as we have just shown, the ideal representation of the corresponding probability distribution is by its cdf $F(x_1, \ldots, x_n)$.

In many practical situations, however, knowledge comes *gradually*. Usually, first, we are interested in the values of the first quantity, someone else may be interested in the values of the second quantity, etc. The resulting information is provided by the corresponding marginal distributions $F_i(x_i)$.

After that, we may get interested in the relation between these quantities $X_1, \ldots, X_n$. Thus, we would like to supplement the marginal distributions with an additional information that would enable us to reconstruct the multi-D cdf $F(x_1, \ldots, x_n)$.

In principle, we can store this multi-D cdf as the additional information. However, this is not the most efficient approach. Indeed, it is well known that each marginal distribution $F_i(x_i)$ can be reconstructed from the multi-D cdf, as

$$F_i(x_i) = F(+\infty, \ldots, +\infty, x_i, +\infty, \ldots, \infty) = \lim_{T \to \infty} F(T, \ldots, T, x_i, T, \ldots, T).$$

So, if we supplement the original marginals with the multi-D cdf, we thus store duplicate information, and duplication is a waste of computer memory.

Situations when we go from 1-D to multi-D case: resulting problem. In the general multi-D case, we have shown that storing a cdf is an appropriate way of representing a multi-D distribution in a computer. However, in situations when we go from 1-D to multi-D case, this representation is no longer optimal: it involves duplication and is, thus, a waste of computer memory.

Copula-based computer representations: a possible way to solve this problem. To avoid duplication, some researchers and practitioners use copula-based representations of multi-D distributions.

A copula corresponding to a multi-D distribution with cdf $F(x_1, \ldots, x_n)$ is a function $C(x_1, \ldots, x_n)$ for which

$$F(x_1, \ldots, x_n) = C(F_1(x_1), \ldots, F_n(x_n)),$$

where $F_i(x_i)$ are the corresponding marginal distributions; see, e.g., [3, 7, 8]. A copula-related way to represent a multi-D distribution is to supplement the marginals $F_i(x_i)$ with the copula $C(x_1, \ldots, x_n)$.

The above formula then enables us to reconstruct the multi-D cdf $F(x_1, \ldots, x_n)$. This representation has no duplication, since for the same copula, we can have many different marginals.

Remaining problem. A copula-based representation avoids duplication and is, in this sense, better than storing the cdf. But *is the copula-based representation optimal* (in some reasonable sense), or is an even better representation possible?

And if the copula-based representation *is* optimal, is it *the only optimal* one, or are there other representations which are equally good?

These are the questions that we will answer in the next section. Of course, in order to answer them, we need to first formulate them in precise terms.

4 Formalization of the problem and the main result

Analysis of the problem. Let us start by formalizing the above problem. We want a computer representation that will be duplicate-free and computationally efficient. What do we mean by computationally efficient?

As we have argued, for making decisions, we need to know the values of the multi-D cdf $F(x_1, \ldots, x_n)$. Thus, whatever representation we come up with, we need to be able to reconstruct the cdf based on this representation. Thus, to make the representation computationally efficient, we need to make sure that the algorithm for reconstructing the cdf is as fast as possible (i.e., that this algorithm consists of as few computational steps as possible), and that this representation uses as little computer memory as possible.

To find such an optimal representation, we need to have precise definitions of what is an algorithm, what is a computational step, and when is an algorithm computationally efficient (in terms of both computation time and computer space). Let us start with providing an exact definition of an algorithm.

Towards a precise description of what is an algorithm. We want to be able, given the marginals and the additional function(s) used for representing the distribution, to reconstruct the multi-D cdf $F(x_1, \ldots, x_n)$. This reconstruction has to be done by a computer *algorithm*.

An algorithm is a sequence of steps, in each of which we either apply some operation ($+$, $-$, sin, given function) to previously computed values, or decide where to go further, or stop.

In our computations, we can use inputs, we can use auxiliary variables, and we can use constants. In accordance with the IEEE 754 standard describing computations with real numbers, infinite values $-\infty$ and $+\infty$ can be used as well.

It is also possible to declare a variable as "undefined" (in IEEE 754 this is called "not a number", NaN for short). For each function or operation, if at least one of the inputs is undefined, the result is also undefined.

We thus arrive at the following formal definition of an algorithm:

Definition 1.

- *Let F be a finite list of functions $f_i(z_1, \ldots, z_{n_i})$.*
- *Let $v_1, \ldots, v_m$ be a finite list of real-valued variable called* inputs.
- *Let $a_1, \ldots, a_p$ be a finite list of real-valued variables called* auxiliary variables.
- *Let $r_1, \ldots, r_q$ be real-valued variables; they will be called the* results *of the computations.*

An algorithm $\mathcal{A}$ *is a finite sequence of instructions $I_1, \ldots, I_N$ each of which has one of the following forms:*

- *an* assignment *instruction "$y \leftarrow y_1$" or "$y \leftarrow f_i(y_1, \ldots, y_{n_i})$", where:*
  - *y is one of the auxiliary variables or a result variable,*
  - *$f_i \in F$, and*

  - *each y_i is either an input, or an auxiliary variable, or a result, or a real number (including $-\infty$, $+\infty$, and NaN);*
- *an* unconditional branching instruction "go to I_i;
- *a* conditional branching *instruction* "if $y_1 \odot y_2$, then to I_i else go to I_j"*, where:*
  - *each y_i is either an input, or an auxiliary variable, or the result, or a real number (including $-\infty$ and $+\infty$); and*
  - *$\odot$ is one of the symbols $=$, $\neq$, $<$, $>$, $\leq$, and $\geq$;*
- *or a* stopping *instruction* "stop"*.*

Definition 2. *The* result *of applying an algorithm $\mathcal{A}$ to the inputs $a_1, \ldots, a_m$ is defined as follows:*

- *in the beginning, we start with the given values of the inputs, all other variables are undefined;*
- *we then start with instruction I_1;*
- *on each instruction:*
  - *if this is an assignment instruction $y \leftarrow y_1$ or $y \leftarrow f_i(y_1, \ldots, y_{n_i})$, we assign, to the variable y, the new value y_1 or $f_i(y_1, \ldots, y_{n_i})$ and go to the next instruction;*
  - *if this is an unconditional branching instruction, we go to instruction I_i;*
  - *if this is a conditional branching instruction and both values y_1 and y_2 are defined, we check the condition $y_1 \odot y_2$ and, depending of whether this condition is satisfied, go to instruction I_i or to instruction I_j;*
  - *if this a conditional branching instruction, and at least one of the values y_i is undefined, we stop;*
  - *if this a stopping instruction, we stop.*

The values $r_1, \ldots, r_q$ at the moment when the algorithm stops are called the result *of applying the algorithm.*

Toward a formal definition of the number of computational steps. The above definition of an algorithm as a step-by-step procedure leads to the following natural definition of the number of computational steps:

Definition 3. *For every algorithm $\mathcal{A}$ and for each tuple of inputs $v_1, \ldots, v_m$, the number of instructions that the algorithm goes through before stopping is called the* running time *of $\mathcal{A}$ on $v_1, \ldots, v_m$.*

Examples. To illustrate the above definition, let us start with simple algorithms.

1°. The standard algorithm for computing the value $r_1 = v_1 \cdot (1 - v_1)$ requires the use of two arithmetic operations: subtraction $f_1(z_1, z_2) = z_1 - z_2$ and multiplication $f_2(z_1, z_2) = z_1 \cdot z_2$. Here, we can use a single auxiliary variable a_1. The corresponding instructions have the following form:

I_1: $a_1 \leftarrow f_1(1, v_1)$; this instruction computes $a_1 = 1 - v_1$;
I_2: $r_1 \leftarrow f_2(v_1, a_1)$; this instruction computes $r_1 = v_1 \cdot a_1 = v_1 \cdot (1 - v_1)$;
I_3: stop.

For all the inputs, this algorithm goes through two instructions before stopping, so its running time is 2.

2°. Computation of the absolute value $|v_1|$, i.e., v_1 if $v_1 \geq 0$ and $-v_1$ otherwise, requires that we use a unary minus operation $f_1(z_1) = -z_1$. The corresponding instructions have the following form:

I_1: if $v_1 \geq 0$, then go to I_2 else go to I_4;
I_2: $r_1 \leftarrow v_1$;
I_3: stop;
I_4: $r_1 \leftarrow f_1(v_1)$;
I_5: stop.

This algorithm also goes through two instructions before stopping, so its running time is also 2.

3°. Computation of $n! = 1 \cdot 2 \cdot \ldots n$ for a given natural number n requires:

- two arithmetic operations: addition $f_1(z_1, z_2) = z_1 + z_2$ and multiplication $f_2(z_1, z_2) = z_1 \cdot z_2$; and
- a loop, with an additional variable a_1 that takes the values 1, 2, ..., n.

The corresponding instructions have the following form:

I_1: $r_1 \leftarrow 1$;
I_2: $a_1 \leftarrow 1$;
I_3: if $a_1 \leq v_1$, then go to I_4 else go to I_7;
I_4: $r_1 \leftarrow f_2(r_1, a_1)$;
I_5: $a_1 \leftarrow f_1(a_1, 1)$;
I_6: go to I_3;
I_7: stop.

The running time of this algorithm depends on the input v_1.

- When $v_1 = 0$, we go through three instructions I_1, I_2, and I_3 before stopping, so the running time is 3.
- When $v_1 = 2$, we go through I_1, I_2, I_3, I_5, I_6, then again I_3, I_4, I_5, and I_6, and finally I_3 and stop. In this case, the running time is 11.

4°. If we already have the multi-D cdf as one of the basic functions $f_i(z_1, \ldots, z_n) = F(z_1, \ldots, z_n)$, then computing cdf for given inputs requires a single computational step:

I_1: $r_1 \leftarrow f_1(v_1, \ldots, v_n)$;
I_2: stop.

The running time of this algorithm is 1.

5°. Similarly, if we have a copula $f_1(z_1, \ldots, z_n) = C(z_1, \ldots, z_n)$, and we can use the values $v_{n+i} = F_i(x_i)$ as additional inputs, the corresponding algorithm for computing the cdf also has a running time of 1:

I_1: $r_1 \leftarrow f_1(v_{n+1}, \ldots, v_{2n})$;
I_2: stop.

What is a computer representation of a multi-D distribution: towards a formal definition. Now, we are ready to provide a formal definition of a computer representation: it is a representation in which, in addition to the marginals, we have one or more functions that enable us to algorithmically reconstruct the cdf.

Definition 3. *By a* representation of an n-dimensional probability distribution*, we mean a tuple consisting of:*

- *finitely many fixed functions* $G_i(z_1, \ldots, z_{n_i})$*, same for all distributions (such as* $+$*,* $\cdot$*, etc.);*
- *finitely many functions* $H_i(z_1, \ldots, z_{m_i})$ *which may differ for different distributions; and*
- *an algorithm (same for all distributions), that, using the above functions and* $2n$ *inputs* x_1*, ...,* x_n*,* $F_1(x_1)$*, ...,* $F_n(x_n)$*, computes the values of the cdf* $F(x_1, \ldots, x_n)$*.*

Examples.

- In the original representation by a cdf, we have $H_1(z_1, \ldots, z_n) = F(z_1, \ldots, z_n)$.
- In the copula representation, we have $H_1(z_1, \ldots, z_n) = C(z_1, \ldots, z_n)$.

The corresponding algorithms for computing the cdf are described in the previous text.

What is duplication-free: towards a precise definition. In the previous section, we argued that if we represent a distribution by storing both its marginals and its cdf, then this representation contains duplicate information: indeed, based on the cdf, we can reconstruct the marginals.

In precise terms, the original representation by a cdf, when we have $H_1(z_1, \ldots, z_n) = F(z_1, \ldots, z_n)$, is not duplication-free, since we can compute, e.g., the marginal $F_1(v_1)$ by applying the following algorithm:

I_1: $r_1 \leftarrow H_1(v_1, +\infty, \ldots, +\infty)$;
I_2: stop.

It is therefore reasonable to call a representation duplication-free if such a reconstruction is impossible:

Definition 4. *We say that a representation is* duplication-free *if no algorithm is possible that, given the functions* H_i *representing the distribution and the inputs* $x_1, \ldots, x_n$*, computes one of the marginals.*

Example. The copula representation is duplication-free: indeed, for the same copula, we can have different marginals, and thus, it is not possible to compute the marginals based on the copula.

A representation must be computationally efficient: towards precise definitions. First, we want the reconstruction of the cdf to be as fast as possible:

Definition 5. *We say that a duplication-free representation is* time-efficient *if for each combination of inputs, the running time of the corresponding algorithm does not exceed the running time of any other duplication-free algorithm.*

As we have mentioned earlier, in addition to an efficient use of computation *time*, it is also important to make sure that computer *memory* is used efficiently: this is why it makes sense to consider only duplication-free representations.

In general, we store the values of one of several functions of different number of variables. To store a function of m variables, we need to store, e.g., its values on the corresponding grid. If we use g different values of each of the coordinates, then we need to store the values of this function at g^m points, i.e., we need to

store g^m real numbers. Thus, the smaller m, the more efficient we are. This leads to the following definition.

Definition 6.

- *We say that a representation* $H_1(z_1, \ldots, z_{m_1})$, $\ldots$, $H_k(z_1, \ldots, z_{m_k})$ *is* more space-efficient *than a representation* $H'_1(z_1, \ldots, z_{m'_1})$, $\ldots$, $H'_{k'}(z_1, \ldots, z_{m'_{k'}})$ *if* $k \le k'$ *and we can sort the value* m_i *and* m'_i *in such as way that* $m_i \le m'_i$ *for all* $i \le k$.
- *We say that a time-efficient duplication-free representation is* computationally efficient *if it is more space-efficient than any other time-efficient duplication-free representation.*

Main Result. *The only computationally efficient duplication-free representation of multi-D probability distributions is the copula representation.*

Discussion. Thus, copulas are indeed the most efficient way of representing additional information about the multi-D distributions for which we already know the marginals. This theoretical result explains why copulas have been efficiently used in many applications.

Proof.

1°. By definition, a computationally efficient representation should be time-efficient. By definition of time efficiency, this means that for each combination of inputs, the running time of the corresponding algorithm should not exceed the running time of any other duplication-free algorithm.

We know that the copula representation is duplication-free and that its running time is 1 for all the inputs. Thus, for all the inputs, the running time of the computationally efficient algorithm should not exceed 1. Thus, this algorithm can have exactly one non-stop instruction.

2°. This instruction is our only chance to change the value of the output variable r_1, so this instruction must be of assignment type $r_1 \leftarrow f_1(y_1, \ldots, y_{n_1})$. Since we did not have time to compute the values of any auxiliary variables – this is our first and only instruction – the values $y_1, \ldots, y_{n_1}$ must be the original inputs.

3°. The function f_1 cannot be from the list of fixed functions, since otherwise

- we would get the same result for all possible probability distributions, and thus,
- we would not be able to compute the corresponding values of the cdf $F(x_1, \ldots, x_n)$, which are different for different distributions.

Thus, the function f_1 must be one of the functions H_i characterizing a distribution.

4°. This function $f_1 = H_i$ cannot have fewer than n inputs, because otherwise, some variable x_j will not be used in this computation. Thus, the list of functions H_i used to describe a probability distribution must include at least one function of n variables.

5°. We are interested in a computationally efficient duplication-free representation. By definition, this means that this representation must be more space-efficient than

any other time-efficient duplication-free representation. We know one time-efficient duplication-free representation – it is the copula representation, in which we use a single function H_1 of n variables.

The fact that our representation is more space-efficient than this one means that it uses only one function, and this must be a function of n or fewer variables. We have already shown that we cannot have a function of fewer than n variables, so we must have a function of exactly n variables.

6°. The result $F(x_1, \ldots, x_n)$ of applying this function of n variables must depend on all n variables $x_1, \ldots, x_n$. Thus, for each of these variables x_i, either this same value x_i or the value $F_i(x_i)$ must be among its inputs.

7°. If one of the inputs is x_i, i.e., if the corresponding instruction has the form

I_1: $r_1 \leftarrow H_1(y_1, \ldots, y_{i-1}, x_i, y_{i+1}, \ldots, y_n)$;

where each y_i is either x_i or $F_i(x_i)$, then we will be able to compute the corresponding marginal by using the instruction

I_1: $r_1 \leftarrow H_1(Y_1, \ldots, Y_{i-1}, x_i, Y_{i+1}, \ldots, Y_n)$;

where $Y_i = +\infty$ when $y_i = x_i$ and $Y_i = 1$ when $y_i = F_i(x_i)$. Since we assumed that our scheme is duplication-free, this means that such a case is not possible, and thus, all the inputs to the function H_1 are not the values x_i, but the values of the marginals. Thus, the corresponding instruction has the form

I_1: $r_1 \leftarrow H_1(F_1(x_1), \ldots, F_n(x_n))$;

The result of this computation should be the multi-D cdf, so we should have

$$F(x_1, \ldots, x_n) = H_1(F_1(x_1), \ldots, F_n(x_n))$$

for all possible values $x_1, \ldots, x_n$.

This is exactly the definition of the copula, so we indeed conclude that every computationally efficient representation of a multi-D probability distribution is the copula representation. The main result is proven.

5 An additional argument: a similar reasoning explains the empirical success of neural networks

An example can make our explanation more convincing. At first glance, the above reasoning may sound somewhat obscure. Yes, at first glance, it sounds somewhat reasonable, but to be more convincing, it is nice to have an example where a similar reasoning works.

It turns out that such an example does exist: namely, according to [5], a similar reasoning explains the empirical success of neural networks (see, e.g., [1]). Let us give a brief description of this explanation; details are given in [5].

In many practical applications, data processing speed is important. For example, data processing speed is important in real-time control, where we need to process data in time to make a decision – so that, e.g., a fast-flying drone can avoid the obstacle.

Parallel computing is an answer. A natural way to increase the speed of the computations is to perform computations *in parallel* on several processors. To make the computations really fast, we must divide the algorithm into parallelizable steps, each of which requires a small amount of time.

What are these steps?

The fewer variables, the faster. One of the main reasons why control algorithms are computationally complicated is that we must process many inputs. For example, controlling a car is easier than controlling a plane, because the plane (as a 3-D object) has more characteristics to take care of, more characteristics to measure and hence, more characteristics to process. Controlling a spaceship, especially during the lift-off and landing, is even a more complicated task, usually performed by several groups of people who control the trajectory, temperature, rotation, etc. In short, the more numbers we need to process, the more complicated the algorithm. Therefore, if we want to decompose our algorithm into fastest possible modules, we must make each module to process as few numbers as possible.

Functions of one variable are not sufficient. Ideally, we should only use the modules that compute functions of one variable. However, if we only have functions of one variables (i.e., procedures with one input and one output), then, no matter how we combine them, we will always end up with functions of one variable. Since our ultimate goal is to compute the control function $u = f(x_1, \ldots, x_n)$ that depends on many variables $x_1, \ldots, x_n$, we must therefore enable our processors to compute at least one function of two or more variables.

What functions of two variables should we choose?

Choosing functions of two or more variables. Inside the computer, each function is represented as a sequence of hardware implemented operations. The fastest functions are those that are computed by a single hardware operation. The basic hardware supported operations are arithmetic operations $a + b$, $a - b$, $a \cdot b$, and a/b. The time required for each operation, crudely speaking, corresponds to the number of bits operations that have to be performed:

- Addition and subtraction are usually implemented in the same way.
- Multiplication is implemented as a sequence of additions, so it is much slower than addition.
- Division is done by successive multiplications (basically, in the same way as we do it manually), so, it is a much slower operation than multiplication.

So, the fastest possible functions of two variables are addition and subtraction. Thus, the fastest function of two or more variables are the functions that are obtained by adding and subtracting these variables – i.e., linear functions.

Summarizing the above-given analysis, we can conclude that our computer will contain modules of two type:

- modules that compute functions of one variable;
- modules that compute a linear combination of two or several numbers.

How to combine these modules? We want to combine these modules in such a way that the resulting computations are as fast as possible. The time that is required for an algorithm is crudely proportional to the number of sequential steps that it takes. We can describe this number of steps in clear geometric terms:

– at the beginning, the input numbers are processed by some processors; these processors form the *first layer* of computations;
– the results of this processing may then go into different processors, that form the *second layer*;
– the results of the second layer of processing go into the *third layer*,
– etc.

In these terms, the fewer layers the computer has, the faster it is.

So, we would like to combine the processors into the smallest possible number of layers.

Now, we are ready for the formal definitions.

Definitions and the main result. Let us first give an inductive definition of what it means for a function to be computable by a k-layer computer.

Definition 4.

- *We say that a function $f(x_1, \ldots, x_n)$ is* computable by a 1-layer computer *if either $n = 1$, or the function f is linear.*
- *Let $k \geq 1$ be an integer. We say that a function $f(x_1, \ldots, x_n)$ is* computable by a $(k+1)$-layer computer *if one of the following two statements is true:*
  - *$f(x_1, \ldots, x_n) = g(h(x_1, \ldots, x_n))$, where $g(x)$ is a function of one variable, and $h(x_1, \ldots, x_n)$ is computable by a k-layer computer;*
  - *$f(x_1, \ldots, x_n) = a_0 + \sum_{i=1}^{m} a_i \cdot g_i(x_1, \ldots, x_n)$, where all functions $g_i(x_1, \ldots, x_n)$ are computed by a k-layer computer.*

Comment. A computer is a finite-precision machine, so, the results of the computations are never absolutely precise. Also, a computer is limited in the size of its numbers. So, we can only compute a function approximately, and only on a limited range. Therefore, when we say that we can compute an arbitrary function, we simply mean that for an arbitrary range T, for an arbitrary continuous function $f : [-T, T]^n \to R$, and for an arbitrary accuracy $\varepsilon > 0$, we can compute a function that is ε-close to f on the given range. In this sense, we will show that not every function can be computed on a 2-layer computer, but that 3 layers are already sufficient.

Proposition 1. [5] *There exist real numbers T and $\varepsilon > 0$, and a continuous function $f : [-T, T]^n \to R$ such that no function ε-close to f on $[-T, T]^n$ can be computed on a 2-layer computer.*

Proposition 2. *For every real numbers T and $\varepsilon > 0$, and for every continuous function $f : [-T, T]^n \to R$, there exists a function $\widetilde{f}$ that is ε-close to f on $[-T, T]^n$ and that is computable on a 3-layer computer.*

Proof. This proposition follows from the universal approximation property of 3-layer neural networks [1], in which the output y has the form

$$y = \sum_{k=1}^{K} W_k \cdot y_k - W_0,$$

where $y_k = s\left(\sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}\right)$ and $s(z) = \dfrac{1}{1 + \exp(-z)}$. This function can be computed by the following 3-layer computer:

- first, we compute all the linear combinations $z_k \stackrel{\text{def}}{=} \sum_{i=1}^{n} w_{ki} \cdot x_i - w_{k0}$;
- then, we apply a function $s(z)$ of one variable to all the values z_k, computing $y_k = s(z_k)$;
- finally, we compute the linear combination $y = \sum_{k=1}^{K} W_k \cdot y_k - W_0$.

Discussion. So, arguments similar to the ones we used to explain copulas also explain the empirical success of neural networks. Neural networks are known to be very efficient in many practical applications. Thus, the fact that similar arguments explain neural networks can serve as an additional argument in favor of our explanation of the efficiency of copulas.

6 Conclusions and future work

Conclusions. The need for representing multi-D distributions in a computer comes from the fact that to make decisions, we need to be able to compute (and compare) the expected values of different utility functions. So, from all possible computer representations of multi-D distributions, we should select the ones for which the corresponding computations are the most efficient.

In this paper, we have shown that in situations where we already know the marginals, copulas are indeed the most computationally efficient way of representing additional information about the multi-D distributions.

Possible future work. In this paper, we have concentrated on computing the cumulative distribution function (cdf). This computation corresponds to *binary* utility functions – i.e., utility functions that take only two values $u^+ > u^-$. Such binary functions provide a good first approximation to the user's utilities and user's preferences, but to obtain a more accurate description of user's preferences, we need to use utility functions from a wider class.

It is therefore desirable to find out, for wider classes of utility functions, which computer representations are the most computationally efficient for computing the corresponding expected values. The empirical success of copulas leads us to a natural conjecture that for many such classes, the copula-based computer representations will still be the most computationally efficient.

Acknowledgments

The authors are thankful to all the participants of the Fourth International Symposium on Integrated Uncertainty in Knowledge Modelling and Decision Making IUKM'2015 (Nha Trang, Vietnam, October 15–17, 2015) for valuable discussions.

References

1. C. M. Bishop, Pattern recognition and machine learning. Springer, New York (2006)
2. P. C. Fishburn, Utility theory for decision making. John Wiley & Sons Inc., New York (1969)

3. P. Jaworski, F. Durante, W. K. Härdle, and T. Ruchlik (eds.), Copula theory and its applications. Springer Verlag, Berlin, Heidelberg, New York (2010)
4. V. Kreinovich, H. T. Nguyen, S. Sriboonchitta, and O. Kosheleva, Why copulas have been successful in many practical applications: a theoretical explanation based on computational efficiency", In: V.-N. Huynh, M. Inuiguchi, and T. Denoeux (eds.), Proceedings of The Fourth International Symposium on Integrated Uncertainty in Knowledge Modelling and Decision Making IUKM'2015, pp. 112–125.
5. V. Kreinovich and A. Bernat, Parallel algorithms for interval computations: an introduction, Interval Computations, Vol. 1994, No. 3, pp. 6–62 (1994)
6. R. D. Luce and R. Raiffa, Games and decisions: introduction and critical survey. Dover, New York (1989)
7. A. J. McNeil, R. Frey, and P. Embrechts, Quantitative risk management: concepts, techniques, tools. Princeton University Press, Princeton, New Jersey (2005)
8. R. B. Nelsen, An Introduction to copulas. Springer Verlag, Berlin, Heidelberg, New York (1999)
9. H. Raiffa, Decision analysis. Addison-Wesley, Reading, Massachusetts (1970)