12-2011

Why Bernstein Polynomials Are Better: Fuzzy-Inspired Justification

Jaime Nava
The University of Texas at El Paso, jenava@miners.utep.edu

Olga Kosheleva
The University of Texas at El Paso, olgak@utep.edu

Vladik Kreinovich
The University of Texas at El Paso, vladik@utep.edu

Comments:
Technical Report: UTEP-CS-11-62

Why Bernstein Polynomials Are Better: Fuzzy-Inspired Justification

Jaime Nava[1], Olga Kosheleva[2], and
Vladik Kreinovich[3]
[1,3]Department of Computer Science
[2]Department of Teacher Education
University of Texas at El Paso
500 W. University
El Paso, TX 79968, USA
[1]nava.jaime@gmail.com
[2,3]{olgak,vladik}@utep.edu

***Abstract*—It is well known that an arbitrary continuous function on a bounded set – e.g., on an interval $[a, b]$ – can be, with any given accuracy, approximated by a polynomial. Usually, polynomials are described as linear combinations of monomials. It turns out that in many computational problems, it is more efficient to represent a polynomial as *Bernstein polynomials* – e.g., for functions of one variable, a linear combination of terms $(x - a)^k \cdot (b - x)^{n-k}$. In this paper, we provide a simple fuzzy-based explanation of why Bernstein polynomials are often more efficient, and we show how this informal explanation can be transformed into a precise mathematical explanation.**

I. Introduction: Polynomial and Bernstein Approximations

Functional dependencies are ubiquitous. Several different quantities are used to describe the state of the world – or a state of a system in which are are interested. For example, to describe the weather, we can describe the temperature, the wind speed, the humidity, etc. Even in simple cases, to describe the state of a simple mechanical body at a given moment of time, we can describe its coordinates, its velocity, its kinetic and potential energy, etc.

Some of these quantities can be directly measured – and sometimes, direct measurement is the only way that we can determine their values. However, once we have measured the values of a few basic quantities $x_1, \ldots, x_n$, we can usually compute the values of all other quantities y by using the known dependence $y = f(x_1, \ldots, x_n)$ between these quantities. Such functional dependencies are ubiquitous, they are extremely important in our analysis of real-world data.

Need for polynomial approximations. With the large amount of data that are constantly generated by different measuring devices, most of the data processing is performed by computers. So, we need to represent each known functional dependence $y = f(x_1, \ldots, x_n)$ in a computer.

In the computer, the only operations which are directly hardware supported (and are therefore extremely fast) are addition, subtraction, and multiplication. All other operations, including division, are implemented as a sequence of addition, subtractions, and multiplications. Therefore, if we want to compute a function $f(x_1, \ldots, x_n)$, we must represent it as a sequence of additions, subtractions, and multiplications. A function which is obtained from variables $x_1, \ldots, x_n$ and constants by using addition, subtraction, and multiplication is nothing else but a polynomial. Indeed, one can easily check that every polynomial can be computed by a sequence of additions, subtractions, and multiplications. Vice versa, by induction, one can easily prove that every sequence of additions, subtractions, and multiplications leads to a polynomial; indeed:

- induction base is straightforward: each variables x_i is a polynomial, and each constant is a polynomial;
- induction step is also straightforward:
  - the sum of two polynomials is a polynomial;
  - the difference between two polynomials is a polynomial; and
  - the product of two polynomials is a polynomial.

Possibility of a polynomial approximation. The possibility to approximate functions by polynomials was first proven by Weierstrass (long before computers were invented). Specifically, Weierstrass showed that for every continuous function $f(x_1, \ldots, x_n)$, for every box (multi-interval)

$$[a_1, b_1] \times \ldots \times [a_n, b_n],$$

and for every real number $\varepsilon > 0$, there exists a polynomial $P(x_1, \ldots, x_n)$ which is, on this box, ε-close to the original function $f(x_1, \ldots, x_n)$, i.e., for which

$$|P(x_1, \ldots, x_n) - f(x_1, \ldots, x_n)| \le \varepsilon$$

for all $x_1 \in [a_1, b_1]$, ..., $x_n \in [a_n, b_n]$.

Polynomial approximations to a functional dependence have been used in science for many centuries, they are one of the main tools in physics and other disciplines. Such approximations are often based on the fact that most fundamental physical dependencies are analytical, i.e., can be expanded in convergent Taylor (polynomial) series. Thus, to get a description with a given accuracy, it is sufficient to keep only a few first terms in the Taylor expansion – i.e., in effect,

to approximate the original function by a polynomial; see, e.g. [1].

How to represent polynomials in a computer: traditional approach. A schoolbook definition of a polynomial of one variable is that it is a function of the type

$$f(x) = c_0 + c_1 \cdot x + c_2 \cdot x^2 + \ldots + c_d \cdot x^d.$$

From the viewpoint of this definition, it is natural to represent a polynomial of one variable as a corresponding sequence of coefficients $c_0, c_1, c_2 \ldots, c_d$. This is exactly how polynomials of one variable are usually represented.

Similarly, a polynomial of several variables $x_1, \ldots, x_n$ is usually defined as linear combination of monomials, i.e., expressions of the type $x_1^{d_1} \cdot \ldots \cdot x_n^{d_n}$. Thus, a natural way to represent a polynomial

$$f(x_1, \ldots, x_n) = \sum_{d_1, \ldots, d_n} c_{d_1 \ldots d_n} \cdot x_1^{d_1} \cdot \ldots \cdot x_n^{d_n}$$

is to represent it as a corresponding multi-D array of coefficients $c_{d_1 \ldots d_n}$.

Bernstein polynomials: a description. It has been shown that in many computational problems, it is more efficient to use an alternative representation. This alternative representation was first proposed by a mathematician Bernstein, and so polynomials represented in this form are known as *Bernstein polynomials*. For functions of one variable, Bernstein proposed to represent a function as a linear combination

$$\sum_{k=0}^{d} c_k \cdot (x-a)^k \cdot (b-x)^{d-k}$$

of special polynomials

$$p_k(x) = (x-a)^k \cdot (b-x)^{d-k}.$$

For functions of several variables, Bernstein's representation has the form

$$f(x_1, \ldots, x_n) = \sum_{k_1 \ldots k_n} c_{k_1 \ldots k_n} \cdot p_{k_1 1}(x_1) \cdot \ldots \cdot p_{k_n n}(x_n),$$

where

$$p_{k_i i}(x_i) \stackrel{\text{def}}{=} (x_i - a_i)^{k_i} \cdot (b_i - x_i)^{d-k_i}.$$

In this representation, we store the coefficients $c_{k_1 \ldots k_n}$ in the computer.

Bernstein polynomials are actively used, e.g., in computer graphics and computer-aided design, where they are not only more computationally efficient, but they also lead – within a comparable computation time – to smoother and more stable descriptions than traditional computer representations of polynomials.

Bernstein polynomials are useful in interval and fuzzy computations. In particular, Bernstein polynomials are useful in *interval computations* (see, e.g., [8]), computing the range

$$f([a_1, b_1], \ldots, [a_n, b_n] \stackrel{\text{def}}{=}$$

$$\{f(x_1, \ldots, x_n) : x_1 \in [a_1, b_1], \ldots, x_n \in [a_n, b_n]\}$$

of a function $f(x_1, \ldots, x_n)$ over a given box. The efficiency of Bernstein polynomials in interval computations is shown in [2], [3], [4], [5], [9], [13].

Such interval computations are extremely useful in fuzzy computations (see, e.g., [6], [12]), when we know fuzzy values of the inputs $x_1, \ldots, x_n$ (i.e., the corresponding membership functions $\mu_i(x_i)$), and we need to find the corresponding fuzzy value of $y = f(x_1, \ldots, x_n)$ (i.e., the membership function $\mu(y)$). It turns out that for every α, the α-cut

$$\mathbf{y}(\alpha) = \{y : \mu(y) \geq \alpha\}$$

is equal to the range of the function $f(x_1, \ldots, x_n)$ over the corresponding α-cuts $\mathbf{x}_i(\alpha) = \{x_i : \mu_i(x_i) \geq \alpha\}$:

$$\mathbf{y}(\alpha) = f(\mathbf{x}_1(\alpha), \ldots, \mathbf{x}_n(\alpha)).$$

This is how fuzzy computations are usually performed – by performing interval computations over the corresponding α-cuts, for different values $\alpha \in [0, 1]$.

Bernstein polynomials: open problem. Empirically, it is known that Bernstein polynomials are often more computationally efficient. However, in spite of many efforts to explain this empirical efficiency, no convincing explanation has been found so far.

What we do. In this paper, we first show that by using fuzzy techniques, we can get a reasonable explanation of why Bernstein polynomials are efficient. Then, we show how this informal explanation can be transformed into a precise mathematical justification.

II. Bernstein Polynomials: Fuzzy Explanation

Preliminary step: reducing all intervals to the interval $[0, 1]$. We want to use fuzzy logic to analyze polynomial approximations. In fuzzy logic, traditionally, possible truth values form an interval $[0, 1]$. In some intelligent systems, other intervals are used – e.g., in the historically first expert system MYCIN the interval $[-1, 1]$ was used to describe possible degrees of confidence. It is well known that it does not matter much what interval we use since we can easily reduce values x from an interval $[a, b]$ to values t from the interval $[0, 1]$ by taking $t = \dfrac{x-a}{b-a}$; vice versa, once we know the new value t, we can easily reconstruct the original value x as $x = a + t \cdot (b - a)$.

To facilitate the use of traditional fuzzy techniques, let us therefore reduce all the intervals $[a_i, b_i]$ to the interval $[0, 1]$. In other words, instead of the original function

$$f(x_1, \ldots, x_n) : [a_1, b_1] \times \ldots \times [a_n, b_n] \to \mathbb{R},$$

we consider a new function

$$F(t_1, \ldots, t_n) : [0, 1]^n \to \mathbb{R},$$

which is defined as

$$F(t_1, \ldots, t_n) = f(a_1 + t_1 \cdot (b_1 - a_1), \ldots, a_n + t_n \cdot (b_n - a_n)).$$

Vice versa, if we find a good approximation $\widetilde{F}(t_1,\ldots,t_n)$ to the new function $F(t_1,\ldots,t_n)$, we can easily generate an approximation $\widetilde{f}(x_1,\ldots,x_n)$ to the original function $f(x_1,\ldots,x_n)$ as follows:

$$\widetilde{f}(x_1,\ldots,x_n)=\widetilde{F}\left(\frac{x_1-a_1}{b_1-a_1},\ldots,\frac{x_n-a_n}{b_n-a_n}\right).$$

Fuzzy-based function approximations: reminder. Fuzzy techniques have been actively used to approximate functional dependencies: namely, such dependencies are approximated by fuzzy rules; see, e.g., [6], [7], [12]. The simplest case is when each rule has a fuzzy condition and a crisp conclusion, i.e., has the type

$$\text{"if } x \text{ is } P\text{, then } y=c\text{"},$$

where P is a fuzzy property (such as "small") characterized by a membership function $\mu(x)$, and c is a real number. For the case of several inputs, we have rules of the type

"if x_1 is P_1, x_2 is P_2, ..., and x_n is P_n, then $y=c$."

The degree to which a given input x_i satisfies the property P_i is equal to $\mu_i(x_i)$, where $\mu_i(x)$ is the membership function corresponding to the property P_i. The degree to which the tuple $(x_1,\ldots,x_n)$ satisfies the condition of the rule – i.e., the statement

"x_1 is P_1, x_2 is P_2, ..., and x_n is P_n"

– is therefore equal to $f_{\&}(\mu_1(x_1),\ldots,\mu_n(x_n))$, where $f_{\&}$ is an appropriate t-norm ("and"-operation). One of the simplest t-norms is the product $f_{\&}(a,b)=a\cdot b$. For this t-norm, the degree d to which the above rule is satisfied is equal to the product $\mu_1(x_1)\cdot\ldots\cdot\mu_n(x_n)$ of the corresponding membership degrees.

When we have several rules, then we get different conclusions $c_1,\ldots,c_r$ with degrees $d_1,\ldots,d_r$; we need to come up with a single values that combines these conclusions. The larger the degree d_i, the more weight we should give to the conclusion c_i. A natural way is thus simply to take the weighted average $c_1\cdot d_1+\ldots+c_r\cdot d_r$. This weighted average can be interpreted in fuzzy terms if we interpret the combination as the following statement:

- "either (the condition for the 1st rule is satisfied and its conclusion is satisfied)
- or (the condition for the 2nd rule is satisfied and its conclusion is satisfied)
- or ...
- or (the condition for the r-th rule is satisfied and its conclusion is satisfied),"

where we describe "and" as multiplication and "or" as addition.

Resulting interpretation of the usual polynomial representation. The functions of one variable, the traditional computer representations of a polynomial has the form

$$c_0+c_1\cdot x+c_2\cdot x^2+\ldots+c_m\cdot x^m.$$

The corresponding approximation can be interpreted as the following set of fuzzy rules:

- c_0 (with no condition);
- if x, then c_1;
- if x^2, then c_2; ...
- if x^m, then c_m.

In fuzzy logic, if we take x as the degree to which $x\in[0,1]$ is large, then:

- x^2 is usually interpreted as "very large",
- $x^4=(x^2)^2$ is interpreted as "very very large",
- $x^8=(x^4)^2=((x^2)^2)^2$ is interpreted as "very very very large", etc., and
- intermediate powers x^3, x^5, x^7, etc., are interpreted as as some intermediate hedges.

Thus, the above rules have the form:

- c_0;
- if x is large, then c_1;
- if x is very large, then c_2, etc.

Similarly, for polynomials of several variables, we have as many rules as there are monomials $c_{k_1\ldots k_n}\cdot x_1^{k_1}\cdot\ldots\cdot x_n^{k_n}$. For example, a monomial

$$c_{012}\cdot x_1^0\cdot x_1^1\cdot x_2^2=c_{012}\cdot x_2\cdot x_3^2$$

corresponds to the following rule:

"if x_2 is large and x_3 is very large, then c_{012}."

Fuzzy interpretation reveals limitations of the traditional computer representation of polynomials. From the fuzzy viewpoint, there are two limitations to this interpretation.

The first limitation is related to the fact that an accurate representation requires polynomials of higher degrees, with several distinct coefficients corresponding to different hedges such as "very", "very very", etc. In practice, we humans can only meaningfully distinguish between a small number of hedges, and this limits the possibility of meaningfully obtaining such rules from experts.

The second limitation is that for the purposes of computational efficiency, it is desirable to have a computer representation in which as few terms as possible are needed to represent each function. This can be achieved if in some important cases, some of the coefficients in the corresponding computer representation are close to 0 and can, therefore, be safely ignored. For the above fuzzy representation, all the terms are meaningful, and there seems to be no reason why some of these terms can be ignored.

How can we overcome limitations of the traditional computer representation: fuzzy analysis of the problem. From the fuzzy viewpoint, the traditional computer representation of polynomials corresponds to taking into account the opinions of a single expert. Theoretically, we can achieve high accuracy this way if we have an expert who can meaningfully distinguish between "large", "very large", "very very large", etc. However, most experts are not very good in such a distinction. A typical expert is at his or her best when this

expert distinguishes between "large" and "not large", any more complex distinctions are much harder.

Since we cannot get a good approximation by using a *single* expert, why not use *multiple* experts? In this case, there is no need to force an expert into making a difficult distinction between "very large" and "very very large". So, we can as well use each expert where each expert is the strongest: by requiring each expert to distinguish between "large" and "very large". In this setting, once we have d experts, for each variable x_i, we have the following options:

- The first option is when all d experts believe that x_i is large: the 1st expert believes that x is large, the 2nd believes that x_i is large, etc. Since we have decided to use product for representing "and", the degree to which this condition is satisfied is equal to $x_i \cdot \ldots \cdot x_i = x_i^d$.
- Another option is when $d-1$ experts believe that x_i is large, and the remaining expert believes that x is not large. The corresponding degree is equal to $x_i^{d-1} \cdot (1-x_i)$.
- In general, we can have k_i experts believing that believing that x_i is large and $d-k_i$ experts believing that x is not large. The corresponding degree is equal to

$$x_i^{k_i} \cdot (1-x_i)^{d-k_i}.$$

For this variable, general weighted combinations of such rules lead to polynomials of the type $\sum\limits_{k_i} c_{k_i} \cdot x_i^{k_i} \cdot (1-x_i)^{d-k_i}$, i.e., to Bernstein polynomials of one variable.

For several variables, we have the degree $p_{k_i i}(x_i) = x_i^{k_i} \cdot (1-x_i)^{d-k_i}$ with which each variable x_i satisfies the corresponding condition. Hence, the degree to which all n variables satisfy the corresponding condition is equal to the product $p_{k_1 1}(x_1) \cdot \ldots \cdot p_{k_n n}(x_n)$ of these degrees. Thus, the corresponding fuzzy rules lead to polynomials of the type

$$\sum_{k_1,\ldots,k_n} c_{k_1 \ldots k_n} \cdot p_{k_1 1}(x_1) \cdot \ldots \cdot p_{k_n n}(x_n),$$

i.e., to *Bernstein polynomials.*

So, we indeed get a fuzzy explanations for the emergence of Bernstein polynomials.

Why Bernstein polynomials are more computationally efficient: a fuzzy explanation. Let us show that the above explanation of the Bernstein polynomials leads to the desired explanation of why the Bernstein polynomials are more computationally efficient than the traditional computer representation of polynomials.

Indeed, the traditional polynomials correspond to rules in which conditions are "x is large", "x is very large", "x is very very large", etc. It may be difficult to distinguish between these terms, but there is no reason to conclude that some of the corresponding terms become small.

In contrast, each term $x_i^{k_i} \cdot (1-x_i)^{d-k_i}$ from a Bernstein polynomial, with the only exception of cases $k_i = 0$ and $k_i = D$, corresponds to the condition of the type

- "x_i is large (very large, etc.) *and*
- x_i is not large (very not large, etc.)".

While in fuzzy logic, such a combination is possible, there are important cases when this value is close to 0 – namely, in the practically important cases when we are either confident that x_i is large or we are confident that x_i is not large. In these cases, the corresponding terms can be safely ignored, and thus, computations become indeed more efficient.

III. From Fuzzy Explanation to a More Precise Explanation

Main idea. Let us show that terms $x_i^{k_i} \cdot (1-x_i)^{d-k_i}$ corresponding to $k_i \in (0,d)$ are indeed smaller and thus, some of them can indeed be safely ignored. To prove this fact, let us pick a threshold $\varepsilon > 0$ ($\varepsilon \ll 1$) and in each computer representation of polynomials, let us only keep the terms for which the largest possible value of this term does not exceed ε.

Traditional computer representation of polynomials: analysis. In the traditional representation, the terms are of the type $x_1^{k_1} \cdot \ldots \cdot x_n^{k_n}$. When $x_i \in [0,1]$, each such term is a product of the corresponding terms $x_i^{k_i}$. The resulting non-negative function is increasing in all its variables, and thus, its largest possible value is attained when all the variables x_i attain their largest possible value 1. The corresponding largest value is equal to $1^{k_1} \cdot \ldots \cdot 1^{k_n} = 1$.

Since the largest value of each term is 1, and 1 is larger than the threshold ε, all the terms will be retained. If we restrict ourselves to terms of order $\le d$ for each variable x_i, we get:

- $d+1$ possible terms for one variable:

$$x_i^0 = 1, \quad x_i^1 = x, \quad x_i^2, \quad \ldots, \quad x_i^d,$$

- $(n+1)^2$ terms $x_1^{k_1} \cdot x_2^{k_2}$ for two variables,
- …, and
- $(d+1)^n$ terms in the general case of n variables.

This number of retained terms grows exponentially with the number of variables n.

Bernstein polynomials: analysis. For Bernstein polynomials, each term has the product form $p_{k_1 1}(x_1) \cdot \ldots \cdot p_{k_n n}(x_n)$, where $p_{k_i i}(x_i) = x_i^{k_i} \cdot (1-x_i)^{d-k_i}$. The product of non-negative numbers $p_{k_i i}(x_i)$ is a monotonic function of its factors. Thus, its maximum is attained when each of the factors $p_{k_i i}(x_i) = x_i^{k_i} \cdot (1-x_i)^{d-k_i}$ is the largest possible. Differentiating this expression with respect to x_i, taking into account that the derivative of $f(x) = x^k$ is equal to $\dfrac{k}{x} \cdot f(x)$, and equating the resulting derivative to 0, we conclude that

$$\frac{k_i}{x_i} \cdot p_{k_i i}(x_i) - \frac{d-k_i}{1-x_i} \cdot p_{k_i i}(x_i) = 0,$$

i.e., that $\dfrac{k_i}{x_i} = \dfrac{d-k_i}{1-x_i}$. Multiplying both sides of this equality by the common denominator of the two fractions, we get

$$k_i \cdot (1-x_i) = (d-k_i) \cdot x_i,$$

i.e., $k_i - k_i \cdot x_i = d \cdot x_i - k_i \cdot x_i$. Adding $k_i \cdot x_i$ to both sides of this equation, we get $k_i = d \cdot x_i$ hence $x_i = \dfrac{k_i}{d}$. Thus, the

largest value of this term is equal to

$$x_i^{k_i} \cdot (1-x_i)^{d-k_i} = \left(\frac{k_i}{d}\right)^{k_i} \cdot \left(1-\frac{k_i}{d}\right)^{d-k_i}.$$

This value is the largest for $k_i = 0$ and $k_i = d$, when the corresponding maximum is equal to 1; as a function of k_i, it first decreases and then increases again. So, if we want to consider values for which this term is large enough, we have to consider value k_i which are close to 0 (i.e., $k_i \ll d$) or close to d (i.e., $d - k_i \ll d$).

For values k_i which are close to 0, we have $\left(1-\frac{k_i}{d}\right)^{d-k_i} \approx \left(1-\frac{k_i}{d}\right)^{d}$. It is known that for large d, this value is asymptotically equal to $\exp(-k_i)$. Thus, the logarithm of the corresponding maximum $\left(\frac{k_i}{d}\right)^{k_i} \cdot \left(1-\frac{k_i}{d}\right)^{d-k_i}$ is asymptotically equal to the logarithm of $\left(\frac{k_i}{d}\right)^{k_i} \cdot \exp(-k_i)$, i.e., to $-k_i \cdot (\ln(d) - \ln(k_i) + 1)$. Since we have $k_i \ll d$, we get $\ln(k_i) \ll \ln(d)$ and therefore, the desired logarithm is asymptotically equal to $-k_i \cdot \ln(d)$.

For the values $k_i \approx d$, we can get a similar asymptotic expression $-(d-k_i)\cdot\ln(d)$. Both expressions can be described as $-\Delta_i \cdot \ln(d)$, where Δ_i denotes $\min(k_i, d-k_i)$, i.e.,

- $\Delta_i = k_i$ when $k_i \ll d$, and
- $\Delta_i = d - k_i$ when $d - k_i \ll d$.

We want to find all the tuples $(k_1, \ldots, k_n)$ for which the product of the terms $p_{k_i i}(x_i)$ corresponding to individual variables is larger than or equal to ε. The logarithm of the product is equal to the sum of the logarithms, so the logarithm if the product is asymptotically equal to $-\sum_{i=1}^{n} \Delta_i \cdot \ln(d)$. Thus, the condition that the product is larger than or equal to ε is asymptotically equivalent to the inequality

$$-\sum_{i=1}^{n} \Delta \cdot \ln(d) \ge \ln(\varepsilon),$$

i.e., to the inequality

$$\sum_{i=1}^{n} \Delta_i \le C \stackrel{\text{def}}{=} \frac{|\ln(\varepsilon)|}{\ln(d)}.$$

The number of tuples of non-negative integers Δ_i that satisfy the inequality $\sum_{i=1}^{n} \Delta_i \le C$ can be easily found from combinatorics.

Namely, we can describe each such tuple if we start with C zeros and then place ones:

- we place the first one after Δ_1 zeros,
- we place the second one after Δ_2 zeros following the first one,
- etc.

As a result, we get a sequence of $C+n$ symbols of which C are zeros. Vice versa, if we have a sequence of $C+n$ symbols of which C are zeros (and thus, n are ones), we can take:

- as Δ_1 the number of 0s before the first one,
- as Δ_2 the number of 0s between the first and the second ones,
- etc.

Thus, the total number of such tuples is equal to the number of ways that we can place C zeros in a sequence of $C+n$ symbols, i.e., equal to

$$\binom{C+n}{C} = \frac{(n+C)\cdot(n+C-1)\cdot\ldots\cdot(n+1)}{1\cdot 2\cdot\ldots\cdot C}.$$

When n is large, this number is asymptotically equal to

$$\text{const} \cdot n^C.$$

Each value Δ_i corresponds to two different values k_i:

- the value $k_i = \Delta_i$ and
- the value $k_i = d - k_i$.

Thus, to each tuple $(\Delta_1, \ldots, \Delta_n)$, there correspond 2^n different tuples $(k_1, \ldots, k_n)$. So, the total number of retained tuples $(k_1, \ldots, k_n)$ – i.e., tuples for which the largest value of the corresponding term is $\le \varepsilon$ – is asymptotically equal to $2^n \cdot n^C$.

Conclusion: Bernstein polynomials are more efficient. As we have shown:

- In the traditional computer representation of a polynomial of degree $\le d$ in each of the variables, we need asymptotically $(d+1)^n$ terms.
- For Bernstein polynomials, we need $2^n \cdot n^C$ terms.

For large n, the factor n^C grows much slower than the exponential term 2^n, and $2^n \ll (d+1)^n$. Thus, in the Bernstein representation of a polynomial, we indeed need much fewer terms than in the traditional computer representation – and therefore, Bernstein polynomials are indeed more efficient.

Comment. There exist other explanations of why Bernstein polynomials are more computationally efficient; see, e.g., [10] and references therein; the main advantage of our explanation is that it comes from fuzzy analysis and is, therefore, intuitively clearer than previously known explanations.

ACKNOWLEDGMENT

This work was supported in part by the National Science Foundation grants HRD-0734825 (Cyber-ShARE Center of Excellence) and DUE-0926721, and by Grant 1 T36 GM078000-01 from the National Institutes of Health.

REFERENCES

[1] R. P. Feynman, R. B. Leighton, and M. Sands, *Feynman Lectures on Physics*, Addison-Wesley, Boston, Massachusetts, 2005.

[2] J. Garloff, "The Bernstein algorithm", *Interval Computation*, 1993, Vol. 2, pp. 154–168.

[3] J. Garloff, "The Bernstein expansion and its applications", *Journal of the American Romanian Academy*, 2003, Vol. 25–27, pp. 80–85.

[4] J. Garloff and B. Graf, "Solving strict polynomial inequalities by Bernstein expansion", In N. Munro, editor, *The Use of Symbolic Methods in Control System Analysis and Design*, volume 56 of IEE Contr. Eng., London, 1999, pp. 339–352.

[5] J. Garloff and A. P. Smith, "Solution of systems of polynomial equations by using Bernstein polynomials", In: G. Alefeld, J. Rohn, S. Rump, and T. Yamamoto (eds.), *Symbolic Algebraic Methods and Verification Methods: Theory and Application*, Springer-Verlag, Wien, 2001, pp. 87–97.

[6] G. J. Klir and B. Yuan, *Fuzzy Sets and Fuzzy Logic*, Prentice Hall, Upper Saddle River, New Jersey, 1995.
[7] V. Kreinovich, G. C. Mouzouris, and H. T. Nguyen, "Fuzzy rule based modeling as a universal approximation tool", In: H. T. Nguyen and M. Sugeno (eds.), *Fuzzy Systems: Modeling and Control*, Kluwer, Boston, Massachusetts, 1998, pp. 135–195.
[8] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*, SIAM Press, Philadelphia, Pennsylvania, 2009.
[9] P. S. V. Nataraj and M. Arounassalame, "A new subdivision algorithm for the Bernstein polynomial approach to global optimization", *International Journal of Automation and Computing*, 2007, Vol. 4, pp. 342–352.
[10] J. Nava and V. Kreinovich, *Theoretical Explanation of Bernstein Polynomials' Efficiency: They Are Optimal Combination of Optimal Endpoint-Related Functions*, University of Texas at El Paso, Department of Computer Science, Technical Report UTEP-CS-11-37, July 2011, available as http://www.cs.utep.edu/vladik/2011/tr11-37.pdf
[11] H. T. Nguyen and V. Kreinovich, *Applications of continuous mathematics to computer science*, Kluwer, Dordrecht, 1997.
[12] H. T. Nguyen and E. A. Walker, *First Course In Fuzzy Logic*, CRC Press, Boca Raton, Florida, 2006.
[13] S. Ray and P. S. V. Nataraj, "A New Strategy For Selecting Subdivision Point In The Bernstein Approach To Polynomial Optimization", *Reliable Computing*, 2010, Vol. 14, pp. 117–137.
[14] L. A. Zadeh, "Fuzzy sets", *Information and control*, 1965, Vol. 8, pp. 338–353.