APIC'95, El Paso, Extended Abstracts, A Supplement to the international journal of Reliable Computing

Intelligent Control Makes Sense Even Without Expert Knowledge: an Explanation

Robert N. Lea[1,2] and Vladik Kreinovich[3]

***Abstract*— Intelligent control is a methodology that transforms the knowledge of expert operators into an actual control strategy. There are many successful applications of this methodology to situations in which expert knowledge exist. Intelligent control methodology is known to be a universal approximator in the sense that for every continuous function $f(x_1, ..., x_n)$ and for every $\varepsilon > 0$, there exists a set of rules for which the resulting control strategy is $\varepsilon-$close to f.**

In some cases, intelligent control methodology is applied even when no expert knowledge exists: instead of trying to approximate the unknown control function by splines, polynomials, or by any other traditional approximation technique, researchers try to approximate it by guessing and tuning the expert rules. Surprisingly, this approximation often works fine.

In this paper, we give a mathematical explanation for this phenomenon, and show that approximation by using intelligent control methodology is indeed (in some reasonable sense) the best.

I. Control Must Be Computed Fast

In automatic control, the computer must constantly compute the current values of control. The value of the control depends on the state of the controlled object (called *plant* in control theory). So, to get a high quality control, we must measure as many characteristics $x_1, ..., x_n$ of the current state as we can. The more characteristics we measure, the more numbers we have to process, so, the more computation steps we must perform. The results of these computations must be ready in no time, before we start the next round of measurements. So, automatic control, especially high-quality automatic control, is a real-time computation problem with a serious time pressure.

The authors are with the [1]LinCom Corporation, 1020 Bay Area Blvd., Suite 200, Houston, TX 77058, [2]NASA Johnson Space Center, Houston, TX 77058, email blea@gothamcity.jsc.nasa.gov, and [3]Computer Science Department of the University of Texas at El Paso, El Paso, TX 79968, USA, vladik@cs.utep.edu. This work was supported in part by NSF Grant No. CDA-9015006 and NASA Research Grant No. 9-757. The authors are thankful to Jim Buckley for valuable discussions.

II. 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?

III. Description of the Fastest Possible Control-Oriented Parallel Computer

A. The fewer variables, the faster

As we have already mentioned, the main reason 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 space shuttle, 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.

B. 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 $u = f(x_1, ..., x_n)$ that depends on many variables $x_1, ..., x_n$, we must therefore enable our processors to compute at least one function of two variables.

What functions of two variables should we choose?

C. Choosing functions of two 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 $a+b$, $a-b$, $a\cdot b$, a/b, and $\min(a,b)$ and $\max(a,b)$. The time required for each operation, crudely speaking, corresponds to the number of bits operations that have to be performed:

- Division is done by successive multiplication, comparison and subtraction (basically, in the same way as we do it manually), so, it is a much slower operation than $-$.
- Multiplication is implemented as a sequence of additions (again, basically in the same manner as we do it manually), so it is much slower than $+$.
- $-$ and $+$ are usually implemented in the same way. To add two $n-$bit binary numbers, we need n bit additions, and also potentially, n bit additions for carries. Totally, we need about $2n$ bit operations.
- min of two $n-$bit binary numbers can be done in n binary operations: we compare the bits from the highest to the lowest, and as soon as they differ, the number that has 0 as opposed to 1 is the desired minimum: e.g., the minimum of 0.10101 and 0.10011 is 0.10011, because in the third bit, this number has 0 as opposed to 1.
- Similarly, max is an $n-$bit operation.

So, the fastest possible functions of two variables are min and max. Similarly fast is computing the minimum and maximum of several (more than two) real numbers. Therefore, we will choose these functions for our control-oriented computer.

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 min and max of two or several numbers.

D. 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.

IV. Definition 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.

- We say that a function $f(x_1,...,x_n)$ is *computable by a 1–layer computer* if either $n=1$, or the function f coincides with min or with max.
- Let $k\ge 1$ be an integer. We say that a function $f(x_1,...,x_n)$ is *computable by a $(k+1)-$layer computer* if one of the following three statements is true:
  - $f=g(h(x_1,...,x_n))$, where g is a function of one variable, and $h(x_1,...,x_n)$ is computable by a $k-$layer computer;
  - $f=\min(g_1(x_1,...,x_n),...,g_k(x_1,...,x_n))$, where all functions g_i are computed by a $k-$layer computer;
  - $f=\max(g_1(x_1,...,x_n),...,g_k(x_1,...,x_n))$, where all functions g_i 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 $\varepsilon-$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. *There exist real numbers T and $\varepsilon>0$, and a continuous function $f:[-T,T]^n\to R$ such that no function $\varepsilon-$close to f on $[-T,T]^n$ can be computed on a 2–layer computer.*

Comment. To make the text more readable, we present both proofs in the last section. However, we will make one comment here. The function that will be proved to be not computable on a 2–layer computer is not exotic at all: it is $f(x_1, x_2) = x_1 + x_2$ on the domain $[-1, 1]^2$, and the Proposition is true for $\varepsilon = 0.4$.

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

Comments.

- In other words, *functions computed by a 3–layer computer are universal approximators.*
- As we will see from the proof, this function $\tilde{f}$ is of the type $\max(A_1, ..., A_k)$, where $A_j = \min(f_{j1}(x_1), ..., f_{jn}(x_n))$. These functions correspond the so-called *intelligent control* [2]: Indeed, let us define

$$U = \max_{i,j,x_i \in [-T,T]} |f_{ji}(x_i)|,$$

and

$$\mu_{ji}(x_i) = \frac{f_{ji}(x_i) - (-U)}{U - (-U)}.$$

Let us now assume that the rules base that describes the expert recommendations for control consists of exactly two rules:

  - "if one of the conditions C_j is true, then $u = U$";
  - "else, $u = -U$",

where each condition C_j means that the following n conditions are satisfied:

  - x_1 satisfies the property C_{j1} (described by a membership function $\mu_{j1}(x_1)$);
  - x_2 satisfies the property C_{j2} (described by a membership function $\mu_{j2}(x_2)$);
  - ...
  - x_n satisfies the property C_{jn} (described by a membership function $\mu_{jn}(x_n)$).

In logical terms, the condition C for $u = U$ has the form

$$(C_{11} \& ... \& C_{1n}) \vee ... \vee (C_{k1} \& ... \& C_{kn}).$$

If we use min for &, and max for $\vee$ (these are the simplest choices in intelligent control methodology), then the degree μ_C with which we believe in a condition $C = C_1 \vee ... \vee C_k$ can be expressed as:

$$\mu_C = \max[\min(\mu_{11}(x_1), ..., \mu_{1n}), ..., \min(\mu_{k1}, ..., \mu_{kn})].$$

Correspondingly, the degree of belief in a condition for $u = -U$ is $1 - \mu_C$. According to intelligent control methodology, we must use a *defuzzification* to determine the actual control, which in this case leads to the choice of

$$u = \frac{U \cdot \mu_C + (-U) \cdot (1 - \mu_C)}{\mu_C + (1 - \mu_C)}.$$

Because of our choice of μ_{ji}, one can easily see that this expression coincides exactly with the function $\max(A_1, ..., A_k)$, where $A_j = \min(f_{j1}(x_1), ..., f_{jn}(x_n))$. So, we get exactly the expressions that stem from the intelligent control methodology.

Since our 3–layer expression describes the fastest possible computation tool, we can conclude that *for control problems, the fastest possible universal computation scheme corresponds to using intelligent control methodology.*

This result explains why intelligent control methodology is sometimes used (and used successfully) without any expert knowledge being present, as an extrapolation tool for the (unknown) control strategy.

- The fact that fuzzy controllers (i.e., control strategies generated by the intelligent control methodology) are universal approximators has been proven before [1, 3, 5, 6, 7].
- We have considered *digital* parallel computers. If we use *analog* processors instead, then min and max stop being the simplest functions. Instead, the sum is the simplest: if we just join the two wires together, then the resulting current is equal to the sum of the two input currents. In this case, if we use a sum (and more general, linear combination) instead of min and max, 3–layer computers are also universal approximators; the corresponding computers correspond to *neural networks* [4].

V. Proofs

A. Proof of the Proposition

Let us proof (by reduction to a contradiction) that if a function $\tilde{f}(x_1, x_2)$ is 0.4–close to $f(x_1, x_2) = x_1 + x_2$ on $[-1, 1]^2$, then $\tilde{f}$ cannot be computed on a 2–layer computer. Indeed, suppose that it is. Then, according to the Definition, the function $\tilde{f}(x_1, x_2)$ is of one of the following three forms:

- $g(h(x_1, x_2))$, where h is computable on a 1–layer computer;

- $\min(g_1(x_1, x_2), ..., g_k(x_1, x_2))$, where all the functions g_i are computable on a 1–layer computer;
- $\max(g_1(x_1, x_2), ..., g_k(x_1, x_2))$, where all the functions g_i are computable on a 1–layer computer.

Let us show case-by-case that all these three cases are impossible.

- In the first case, $\tilde{f}(x_1, x_2) = g(h(x_1, x_2))$, where h is computable on a 1–layer computer. Be definition, this means that h is either a function of one variable, or min, or max. Let us consider all these three sub-cases.
  - If $\tilde{f}(x_1, x_2) = g(h(x_1))$, then the function $\tilde{f}$ depends only on x_1. In particular,
  $$\tilde{f}(0, -1) = \tilde{f}(0, 1). \quad (1)$$
  But since $\tilde{f}$ is ε–close to $f(x_1 + x_2) = x_1 + x_2$, we get $\tilde{f}(0, -1) \le f(0, -1) + \varepsilon = -1 + 0.4 = -0.6$, and $\tilde{f}(0, 1) \ge f(0, 1) - \varepsilon = 1 - 0.4 > 0.6 > -0.6$. So, $\tilde{f}(0, -1) \le -0.6 < \tilde{f}(0, 1)$, hence, $\tilde{f}(0, -1) \ne \tilde{f}(0, 1)$, which contradicts to (1). So, this sub-case is impossible. Similarly, it is impossible to have h depending only on x_2.
  - Let us consider the sub-case when $\tilde{f}(x_1, x_2) = g(\min(x_1, x_2))$. In this sub-case, $\tilde{f}(-1, -1) = g(\min(-1, -1)) = g(-1) = g(\min(-1, 1)) = \tilde{f}(-1, 1)$, and
  $$\tilde{f}(-1, -1) = \tilde{f}(-1, 1). \quad (2)$$
  But $\tilde{f}(-1, -1) \le f(-1, -1) + \varepsilon = -2 + 0.4 = -1.6$, and $\tilde{f}(-1, 1) \ge f(-1, 1) - \varepsilon = 0 - 0.4 = -0.4 > -1.6$, so, the equality (2) is also impossible.
  - Let us now consider the sub-case $\tilde{f}(x_1, x_2) = g(\max(x_1, x_2))$. In this sub-case, $\tilde{f}(-1, 1) = g(\max(-1, 1)) = g(1) = g(\max(1, 1)) = \tilde{f}(1, 1)$, and
  $$\tilde{f}(-1, 1) = \tilde{f}(1, 1). \quad (3)$$
  But $\tilde{f}(-1, 1) \le f(-1, 1) + \varepsilon = 0 + 0.4 = 0.4$, and $\tilde{f}(1, 1) \ge f(1, 1) - \varepsilon = 2 - 0.4 = 1.6 > 0.4$, so, the equality (3) is also impossible.
- In the second case,
  $$\tilde{f}(x_1, x_2) = \min(g_1(x_1, x_2), ..., g_k(x_1, x_2)),$$
  where all the functions g_i are computable on a 1–layer computer. For this case, the impossibility follows from the following sequence of steps:
  - If one of the functions g_i is of the type $\min(x_1, x_2)$, then we can rewrite $\min(g_1, ..., g_{i-1}, \min(x_1, x_2), g_{i+1}, ..., g_n)$ as $\min(g_1, ..., g_{i-1}, g_i^{(1)}, g_i^{(2)}, g_{i+1}, ..., g_k)$, where $g^{(i)}(x_1, x_2) = x_i$ is a function that is clearly computable on a 1–layer computer. After we make such transformations, we get an expression for $\tilde{f}$ that only contains max and functions of one variable.
  - Let us show that this expression cannot contain max. Indeed, if it does, then $\tilde{f}(x_1, x_2) = \min(..., \max(x_1, x_2)) \le \max(x_1, x_2)$. In particular, $\tilde{f}(1, 1) \le \max(1, 1) = 1$. But we must have $\tilde{f}(1, 1) \ge f(1, 1) - \varepsilon = 2 - 0.4 = 1.6 > 1$. The contradiction shows that max cannot be one of the functions g_i.
  - So, each function g_i depends only on one variable. If all of them depend on one and the same variable, say, x_1, then the entire function $\tilde{f}$ depends only on one variable, and we have already proved (in the proof of the first case) that it is impossible. So, some functions g_i depend on x_1, and some of the functions g_i depend on x_2. Let us denote by $h_1(x_1)$ the minimum of all functions g_i that depend on x_1, and by $h_2(x_2)$, the minimum of all the functions g_i that depend on x_2. Then, we can represent $\tilde{f}$ as $\tilde{f}(x_1, x_2) = \min(h_1(x_1), h_2(x_2))$.
  - To get a contradiction, let us first take $x_1 = 1$ and $x_2 = 1$. Then, $\tilde{f}(1, 1) = \min(h_1(1), h_2(1)) \ge f(1, 1) - \varepsilon = 2 - 0.4 = 1.6$. Since the minimum of the two numbers is ≥ 1.6, we can conclude that each of them is ≥ 1.6, i.e., that $h_1(1) \ge 1.6$ and $h_2(1) \ge 1.6$. For $x_1 = 1$ and $x_2 = -1$, we have $\tilde{f}(1, -1) = \min(h_1(1), h_2(-1)) \le f(1, -1) + \varepsilon = 0.4$. Since $h_1(1) \ge 1.6$, we conclude that $\tilde{f}(1, -1) = h_2(-1)$. From
  $$\tilde{f}(1, -1) \ge f(1, -1) - \varepsilon = -0.4, \quad (4)$$
  we can now conclude that $h_2(-1) \ge -0.4$. Similarly, one can prove that $h_1(-1) \ge -0.4$. Hence, $\tilde{f}(-1, -1) = \min(h_1(-1), h_2(-1)) \ge -0.4$. But $\tilde{f}(-1, -1) \le f(-1, -1) + \varepsilon = -2 + 0.4 = -1.6 < -0.4$: a contradiction with (4).

  The contradiction shows that the second case is also impossible.
- In the third case,
  $$\tilde{f}(x_1, x_2) = \max(g_1(x_1, x_2), ..., g_k(x_1, x_2)),$$

where all the functions g_i are computable on a 1–layer computer. For this case, the impossibility (similarly to the second case) follows from the following sequence of steps:

- If one of the functions g_i is of the type $\max(x_1, x_2)$, then we can rewrite $\max(g_1, ..., g_{i-1}, \max(x_1, x_2), g_{i+1}, ..., g_n)$ as $\max(g_1, ..., g_{i-1}, g_i^{(1)}, g_i^{(2)}, g_{i+1}, ..., g_k)$, where $g^{(i)}(x_1, x_2) = x_i$ is a function that is clearly computable on a 1–layer computer. After we make such transformations, we get an expression for $\tilde{f}$ that only contains min and functions of one variable.
- Let us show that this expression cannot contain min. Indeed, if it does, then
$$\tilde{f}(x_1, x_2) = \max(..., \min(x_1, x_2)) \geq \min(x_1, x_2).$$
In particular,
$$\tilde{f}(-1, -1) \geq \min(-1, -1) = -1.$$
But we must have
$$\tilde{f}(-1, -1) \leq f(-1, -1) + \varepsilon = -2 + 0.4 = -1.6 < -1.$$
The contradiction shows that min cannot be one of the functions g_i.
- So, each function g_i depends only on one variable. If all of them depend on one and the same variable, say, x_1, then the entire function $\tilde{f}$ depends only on one variable, and we have already proved (in the proof of the first case) that it is impossible. So, some functions g_i depend on x_1, and some of the functions g_i depend on x_2. Let us denote by $h_1(x_1)$ the maximum of all functions g_i that depend on x_1, and by $h_2(x_2)$, the maximum of all the functions g_i that depend on x_2. Then, we can represent $\tilde{f}$ as
$$\tilde{f}(x_1, x_2) = \max(h_1(x_1), h_2(x_2)).$$
- To get a contradiction, let us first take $x_1 = -1$ and $x_2 = -1$. Then,
$$\tilde{f}(-1, -1) = \max(h_1(-1), h_2(-1)) \leq f(-1, -1) + \varepsilon = -2 + 0.4 = -1.6.$$
Since the maximum of the two numbers is ≤ -1.6, we can conclude that each of them is ≤ -1.6, i.e., that $h_1(-1) \leq -1.6$ and $h_2(-1) \leq -1.6$. For $x_1 = 1$ and $x_2 = -1$, we have
$$\tilde{f}(1, -1) = \max(h_1(1), h_2(-1)) \geq f(1, -1) - \varepsilon = -0.4.$$
Since $h_2(-1) \leq -1.6$, we conclude that $\tilde{f}(1, -1) = h_1(1)$. From
$$\tilde{f}(1, -1) \leq f(1, -1) + \varepsilon = 0.4,$$
we can now conclude that $h_1(1) \leq 0.4$. Similarly, one can prove that $h_2(1) \leq 0.4$. Hence,
$$\tilde{f}(1, 1) = \max(h_1(1), h_2(1)) \geq 0.4. \quad (5)$$
But
$$\tilde{f}(1, 1) \geq f(1, 1) - \varepsilon = 2 - 0.4 = 1.6 > 0.4,$$
which contradicts to (5).

  The contradiction shows that the third case is also impossible.

In all there cases, we have shown that the assumption that $\tilde{f}$ can be computed on a 2–layer computer leads to a contradiction. So, $\tilde{f}$ cannot be thus computed. Q.E.D.

B. Proof of the Theorem

Since the function f is continuous, there exists a $\delta > 0$ such that if $|x_i - y_i| \leq \delta$, then
$$|f(x_1, ..., x_n) - f(y_1, ..., y_n)| \leq \varepsilon.$$

Let us mark the grid points on the grid of size δ, i.e., all the points for which each coordinate $x_1, ..., x_n$ has the form $q_i \delta$ for integer q_i (i.e., we mark the points with coordinates $0, \pm\delta, \pm 2\delta, ..., \pm T$).

On each coordinate, we thus mark $\approx 2T/\delta$ points. So, totally, we mark $\approx (2T/\delta)^n$ grid points. Let us denote the total number of grid points by k, and the points themselves by $P_j = (x_{j1}, ..., x_{jn})$, $1 \leq j \leq k$.

By m, let us denote the minimum of f:
$$m = \min_{x_1 \in [-T,T], \ldots, x_n \in [-T,T]} f(x_1, ..., x_n).$$

For each grid point P_j, we will form piece-wise linear functions $f_{ji}(x_i)$ as follows:

- if $|x_i - x_{ji}| \leq 0.6 \cdot \delta$, then
$$f_{ji}(x_i) = f(P_j) (\geq m);$$
- if $|x_i - x_{ji}| \geq 0.7 \cdot \delta$, then
$$f_{ji}(x_i) = m;$$

- if $0.6 \cdot \delta \le |x_i - x_{ji}| \le 0.7 \cdot \delta$, then

$$f_{ji}(x_i) = m + (f(P_j) - m) \cdot \frac{0.7 \cdot \delta - |x_i - x_{ji}|}{0.7 \cdot \delta - 0.6 \cdot \delta}.$$

Let us show that for these functions f_{ji}, the function

$$\tilde{f}(x_1, ..., x_n) = \max(A_1, ..., A_k),$$

where

$$A_j = \min(f_{j1}(x_1), ..., f_{jn}(x_n)),$$

is $\varepsilon-$close to f.

To prove that, we will prove the following two inequalities:

- For all $x_1, ..., x_n$,

$$\tilde{f}(x_1, ..., i_n) \ge f(x_1, ..., x_n) - \varepsilon.$$

- For all $x_1, ..., x_n$,

$$\tilde{f}(x_1, ..., i_n) \le f(x_1, ..., x_n) + \varepsilon.$$

Let us first prove the first inequality. Assume that we have a point $(x_1, ..., x_n)$. For every $i = 1, ..., n$, by q_i, we will denote the integer that is the closest to x_i/δ. Then, $|x_i - q_i\delta| \le 0.5 \cdot \delta$. These values q_i determine a grid point $P_j = (x_{j1}, ..., x_{jn})$ with coordinates $x_{ji} = q_i\delta$. For this j, and for every i,

$$|x_i - x_{ji}| \le 0.5 \cdot \delta < 0.6 \cdot \delta,$$

therefore, by definition of f_{ji}, we have $f_{ji}(x_i) = f(P_j)$. Hence,

$$A_j = \min(f_{j1}(x_1), ..., f_{jn}(x_n)) =$$

$$\min(f(P_j), ..., f(P_j)) = f(P_j).$$

Therefore,

$$\tilde{f}(x_1, .., x_n) = \max(A_1, ..., A_k) \ge A_j = f(P_j).$$

But since $|x_{ji} - x_i| \le 0.5 \cdot \delta < \delta$, by the choice of δ, we have $|f(x_1, ..., x_n) - f(P_j)| \le \varepsilon$. Therefore, $f(P_j) \ge f(x_1, ..., x_n) - \varepsilon$, and hence,

$$\tilde{f}(x_1, ..., x_n) \ge f(P_j) \ge f(x_1, ..., x_n) - \varepsilon.$$

Let us now prove the second inequality. According to our definition of f_{ji}, the value of $f_{ji}(x_i)$ is always in between m and P_j, and this value is different from m only for the grid points P_j for which $|x_{ji} - x_i| \le 0.7 \cdot \delta$. The value

$$A_j = \min(f_{j1}(x_1), ..., f_{jn}(x_n))$$

is thus different from m only if all the values $f_{ji}(x_i)$ are different from m, i.e., when $|x_{ji} - x_i| \le 0.7 \cdot \delta$ for all i. For this grid point, $|x_{ji} - x_i| \le 0.7 \cdot \delta < \delta$; therefore,

$$|f(P_j) - f(x_1, ..., x_n)| \le \varepsilon$$

and hence, $f(P_j) \le f(x_1, ..., x_n) + \varepsilon$. By definition of f_{ji}, we have $f_{ji}(x_i) \le f(P_j)$. Since this is true for all i, we have

$$A_j = \min(f_{j1}(x_1), ..., f_{jn}(x_n)) \le$$

$$f(P_j) \le f(x_1, ..., x_n) + \varepsilon.$$

For all other grid points P_j, we have

$$A_j(x_1, ..., x_n) = m$$

for a given $(x_1, ..., x_n)$. Since m has been defined as the minimum of f, we have

$$A_j = m \le f(x_1, ..., x_n) < f(x_1, ..., x_n) + \varepsilon.$$

So, for all grid points, we have

$$A_j \le f(x_1, ..., x_n) + \varepsilon,$$

and therefore,

$$\tilde{f}(x_1, ..., x_n) = \max(A_1, ..., A_k) \le f(x_1, ..., x_n) + \varepsilon.$$

The second inequality is also proven.

So, both inequalities are true, and hence, $\tilde{f}$ is $\varepsilon-$close to f. Q.E.D.

References

[1] J. J. Buckley, "Sugeno type controllers are universal controllers", *Fuzzy Sets and Systems*, 1993, pp. 299–303.

[2] A. Kandel and G. Langholtz (Editors), **Fuzzy Control Systems**, CRC Press, Boca Raton, FL, 1994.

[3] B. Kosko, "Fuzzy systems as universal approximators", *Proceedings of the 1st IEEE International Conference on Fuzzy Systems*, San Diego, CA, 1992, pp. 1153–1162.

[4] V. Kreinovich, A. Bernat, "Parallel algorithms for interval computations: an introduction", *Interval Computations*, 1994, No. 3 (to appear).

[5] H. T. Nguyen and V. Kreinovich, "On approximation of controls by fuzzy systems", *Proceedings of the Fifth International Fuzzy Systems Association World Congress*, Seoul, Korea, July 1993, pp. 1414–1417.

[6] L.-X. Wang, "Fuzzy systems are universal approximators", *Proceedings of the IEEE International Conference on Fuzzy Systems,* San Diego, CA, 1992, pp. 1163–1169.

[7] L.-X. Wang and J. Mendel, **Generating fuzzy rules from numerical data, with applications**, University of Southern California, Signal and Image Processing Institute, Technical Report USC-SIPI # 169, 1991.