The Real Number Model in Numerical Analysis

Erich Novak

Abstract. In the real number model of computation one assumes that arithmetic operations with real numbers and comparisons can be done with unit cost. In numerical analysis one often deals with problems where the information is only partial. This is an essential assumption for problems defined on function spaces, such as numerical integration, zero finding, or optimization. For such problems we must usually deal with partial information consisting of function values or Fourier coefficients, because a digital computer can only handle finite sets of numbers instead of functions. In information-based complexity it is assumed that certain functionals can be evaluated by an oracle and each call of the oracle costs $\mathbf{c}$, where $\mathbf{c} > 0$.

In this paper we describe this model of computation more carefully. We also define two variants of stochastic (or Monte Carlo) algorithms. We believe that the results of information-based complexity, see the book of Traub, Wasilkowski, and Woźniakowski (1988), can be reconstructed in our model with only minor changes. As an example we present a “uniform algorithm” for a “uniform problem” in numerical integration.

We compare our model (in the case, where no oracle is used) with the model of Blum, Shub and Smale (1989). These authors use a different cost function. Copy instructions are rather expensive in their model, while they are for free in our model. In spite of this difference we show that the sets $\mathbf{P}$ and $\mathbf{NP}$ coincide with respect to both cost functions.

1. The real number URM (unlimited register machine) without an oracle

We describe the model for the case of complete information. The natural domain of definition for many problems is not a specific $\mathbf{R}^n$ but the union of some or all $\mathbf{R}^n$. Therefore we allow partial functions

$$f : \mathbf{R}^* \to \mathbf{R}^*,$$

where

$$\mathbf{R}^* := \bigcup_{n=1}^{\infty} \mathbf{R}^n.$$

An element $a \in D_f \subset \mathbf{R}^*$ can be (and will be) identified with the finite sequence

$$(n, a_1, a_2, \ldots, a_n).$$

Final version. To appear in J. Complexity.

We assume that we have registers R_k for $k \in \mathbf{Z}$. Each register R_k at any moment of time contains a real number denoted by r_k. The initial configuration in the registers is $r_0 = n$ and $r_i = a_i$ for $i = 1, \dots, n$ with $r_i = 0$ for $i > n$ or $i < 0$. In the same way we interpret the final configuration $(r_k)_k$ of a computation with program P and input a as an element $b \in \mathbf{R}^m$ if $m = r_0 \in \mathbf{N}^+$. We write $P(a) = b$ in this case.

If the computation does not terminate then we write $P(a) = \dagger$. If the computation terminates with $r_0 \notin \mathbf{N}^+$ then we also define $P(a) = \dagger$, i.e., this is regarded as if the computation would not terminate. In the special case of a decision problem we have a total function $f : \mathbf{R}^* \to \{0, 1\}$ and the final configuration is of the form $r_0 = 1$ and $r_1 \in \{0, 1\}$. The contents of the registers may be altered by the machine in response to certain instructions. A finite list of instructions constitutes a program. We have arithmetic instructions, a jump instruction, and several copy instructions, as follows.

Arithmetic instructions

Addition of a constant. For $i \in \mathbf{Z}$ and $s \in \mathbf{R}$ there is an instruction $A(i, s)$. The response of the machine is $r_i := r_i + s$.

Addition of two numbers. For $i, j, k \in \mathbf{Z}$ there is an instruction $A(i, j, k)$. The response of the machine is $r_i := r_j + r_k$.

Multiplication with a constant. For $i \in \mathbf{Z}$ and $s \in \mathbf{R}$ there is an instruction $M(i, s)$. The response of the machine is $r_i := r_i \cdot s$.

Multiplication of two numbers. For $i, j, k \in \mathbf{Z}$ there is an instruction $M(i, j, k)$. The response of the machine is $r_i := r_j \cdot r_k$.

Division of two numbers. For $i, j, k \in \mathbf{Z}$ there is an instruction $D(i, j, k)$. The response of the machine is $r_i := r_j / r_k$ if $r_k \neq 0$. In the case $r_k = 0$ we put $P(a) = \dagger$, as if the computation would not terminate.

These are the five different arithmetic instructions. We have a single jump instruction.

Jump instruction. The machine is able to perform jumps backwards and forwards in the list of instructions. For $i, j \in \mathbf{Z}$ and $l \in \mathbf{N}^+$ there is a jump instruction $J(i, j, l)$. The response of the machine to this instruction is as follows. The contents of R_i and R_j are compared, but all registers are unaltered. If $r_i \geq r_j$ the machine proceeds to the next instruction in the program; if $r_i < r_j$ the machine proceeds to the lth instruction in the program. If the jump is impossible because the program has less than l instructions, then the machine stops operation.

Copy instructions. We have a simple assignment instruction $V(i, s)$ with the response $r_i := s$ and four different kinds of direct and indirect copy instructions, written as $C(i, l)$,

$C(r_i, l)$, $C(i, r_l)$ and $C(r_i, r_l)$, where $i, l \in \mathbf{Z}$. The response of the machine to $C(r_i, r_l)$ is $r_{r_i} := r_{r_l}$ if $r_i \in \mathbf{Z}$ and $r_l \in \mathbf{Z}$. In all other cases we put $P(a) = \dagger$, as if the computation would not terminate. We remark that only the register R_{r_i} is changed by this instruction. The response to the other instructions is defined analogously.

Remark. Some other instructions are sometimes allowed. For example, the evaluation of elementary functions such as sin, cos, exp. The point is to allow all what is allowed in, say, a Fortran program.

To perform a computation the machine must be provided with a program P, i.e., a finite list of instructions, and an initial configuration, i.e., a sequence $(a_k)_k$ of real numbers in the registers. We always assume that $a_0 \in \mathbf{N}$ and $a_i = 0$ for $i > a_0$ or $i < 0$ and identify such an initial configuration with $a = (a_1, \ldots, a_n) \in \mathbf{R}^n$ with $n = a_0$. We can describe the operations of the machine similarly as for the usual URM over $\mathbf{N}$, see, for example, Cutland (1980). If the machine stops computation with the configuration $(b_0, b_1, \ldots)$ and $b_0 = m \in \mathbf{N}^+$, then

$$P(a) = P(a_1, \ldots, a_n) = b = (b_1, \ldots, b_m)$$

is considered as the result of the computation. In all other cases we write $P(a) = \dagger$.

Definition. Let f be a partial function from $\mathbf{R}^*$ to $\mathbf{R}^*$ with a domain D_f. We write $f(a) = \dagger$ if $a \notin D_f$. The program P computes f if $f(a) = \dagger$ iff $P(a) = \dagger$ and

$$f(a) = P(a)$$

for all $a \in D_f \subset \mathbf{R}^*$. The function f is called computable if there is a program that computes f.

2. The cost of a computation

We define the cost of computation $P(a)$ for a machine over $\mathbf{R}$. There are different interesting cost functions, the most natural one seems to be the number of the steps of the computation. Here one step means the execution of an arbitrary instruction. See, for example, Preparata and Shamos (1985).

We prefer a slightly modified definition. In numerical analysis one usually counts the number of arithmetic operations and the number of jumps. Therefore we decide that all copy instructions have zero cost, while all other instructions (arithmetic operations and jumps) have unit cost. This seems to be the usual definition in information-based complexity, see Traub et al. (1988), and numerical analysis – though often the model of computation is not explained in detail.

Definition. Let P be a program (for the machine over $\mathbf{R}$) and $a \in \mathbf{R}^*$. We write cost for the function given by

$$\mathrm{cost}(P, a) = \begin{cases} \text{number of arithmetic operations and jumps needed to compute } P(a) \in \mathbf{R}^*, \\ \infty, \quad \text{if } P(a) = \dagger. \end{cases}$$

Remark. There is another definition of the cost in the literature, see Blum, Shub, Smale (1989). There are several differences, for example, arbitrary polynomials can be evaluated in just one time-step in the BSS-model. The main difference, however, is that the cost of the copy instruction $C(r_i, r_l)$ depends on the size of r_i and r_l. The reason for this alternative cost function seems to be a technical one. In the alternative model, the cost of the copy instruction also depends on its position in the program.

The BSS-cost function is bounded by cost*, defined as follows. First we put

$$\mathrm{cost}^*(C(r_i, r_l)) = \max(|r_i|, |r_l|).$$

Similarly, the cost* of $C(r_i, l)$ is $|r_i|$ and the cost* of $C(i, r_l)$ is $|r_l|$ while the cost* of the other instructions is one. In Section 8 we will compare the two cost functions cost and cost*.

3. Problems with partial information: some examples

In numerical analysis we study problems such as numerical integration of a function f, optimal reconstruction of f, zero finding, or solution of differential or integral equations. Usually we know certain properties of f. This a priori knowledge can be written as $f \in F$, where F is a class of functions. In some cases the elements $f \in F$ can be parameterized by finitely many real numbers, in particular, if F is a space of polynomials. Then it is natural to assume that we have complete information about f, i.e., the function f can be involved directly in the computation via the identification of $f \in F$ with some $x \in \mathbf{R}^*$. Often, however, F is too large and therefore the elements $f \in F$ cannot be involved directly in the computation. In such a case the computation does not use f itself but relies only on some partial information $N(f) \in \mathbf{R}^*$ about f. We describe three examples related to numerical integration.

Example 1. We want to compute

$$S(f) = \int_0^1 f(x)\, dx$$

for a function f from the set

$$F = \{f : [0,1] \to \mathbf{R} \mid |f(x) - f(y)| \le |x - y|\}.$$

We assume that we can obtain finitely many function values $N(f) = (f(t_1), \dots, f(t_n))$. In general, the value of $S(f)$ is not determined by $N(f)$ and therefore using only the information $N(f)$ the value $S(f)$ can only be computed up to some error.

Assume that we want to design an algorithm P for computing $S(f)$ for $f \in F$ up to some fixed $\varepsilon > 0$. In this case we do not have any input for P, i.e., we may assume that at the beginning of the computation the content r_k of the register R_k is zero for every k. During the computation we compute a function value $f(t_1)$, i.e., we need an instruction of the form $r_k := f(t)$, where $t \in [0,1]$. We may assume, for example, that t is in the register R_{-4}. Hence it is enough – at least for this example – to assume that an instruction

$$r_k := f(r_{-4})$$

is available. Later during the computation we may compute a value t_2 and then use the instruction $r_k := f(r_{-4})$ with $r_{-4} = t_2$. Of course it would also be possible to use fixed knots $t_1, \dots, t_n$ for every $f \in F$. Then the t_i may be contained as constants in the program. We define $n \in \mathbf{N}$ by $n = \lceil \frac{1}{4\varepsilon} \rceil$ and consider the quadrature formula

$$Q_n(f) = \frac{1}{n} \sum_{i=1}^{n} f\left(\frac{2i-1}{2n}\right).$$

It is known that $|S(f) - Q_n(f)| \leq \varepsilon$ for $f \in F$. It is easy to write a program that computes $Q_n(f)$ using n arithmetic operations and n instructions of the form $r_k := f(r_{-4})$ (and several assignment instructions which are for free). Observe that $\varepsilon > 0$ is fixed and so is n. Hence n is not computed by the program and we may say that n is precomputed by the designer of the algorithm in advance. Using known results about quadrature formulas and information-based complexity one can prove that this special algorithm almost minimizes the cost.

Example 2. In Example 1 we assumed that $\varepsilon > 0$ is given. It would be better to have an algorithm which, on input ε, computes an ε-approximation of $S(f)$. In this case we could assume that

$$r_0 = 1, \qquad r_1 = \varepsilon, \qquad \text{and } r_k = 0 \text{ for all other } k$$

at the beginning of the computation. Again we must have an instruction such as

$$r_k := f(r_{-4}).$$

Now, however, the number and the position of the knots depend on the input ε. Since any program can only contain finitely many numbers, the knots t_i must be computed during the computation. When the computation terminates we obtain a result $P(\varepsilon, f)$. Of course we demand that this is a real number with

$$|S(f) - P(\varepsilon, f)| \leq \varepsilon.$$

Observe that this example is different from Example 1, because now we have to "compute the quadrature formula" depending on ε. This is easy here because we can put $n = \lceil \frac{1}{4\varepsilon} \rceil$ (which can be computed in about $|\log \varepsilon|$ steps for small ε) and then we compute $Q_n(f)$ with Q_n as in Example 1. This means, however, that the knots are not for free. They cannot be precomputed and used as free constants of the program, using the simple assignment instruction $V(i, s)$.

We want to stress that problems of this kind are rarely studied in information-based complexity, see Paskov (1993) for an interesting exception. One should observe that for most classes F we do not have a simple formula for the knots of the optimal quadrature formula with n knots.

Example 3. We propose an even more "uniform problem", where not only the error tolerance ε is an input but also a parameter $k \in \mathbf{N}^+$ describing the smoothness of f: On input $\varepsilon > 0$ and $k \in \mathbf{N}^+$ compute the integral $S(f) = \int_{-1}^{1} f(x)\, dx$ of

$$f \in F^k := \{g \in C^k([-1,1]) \mid \|g^{(k)}\|_\infty \leq 1\}$$

up to an error less or equal to ε. Assuming that some elementary functions are available, this problem can be solved by compound Clenshaw-Curtis formulas, see Brass (1977) for their properties. First we compute the Clenshaw-Curtis formula $Q_k^{CC} = \sum_{i=1}^{k} a_i f(x_i)$ for the interval $[-1, 1]$, defined by

$$x_i = -\cos\left(\frac{(i-1)\pi}{k-1}\right)$$

with a_i such that $Q_k^{CC}(p) = \int_{-1}^{1} p(x)\, dx$ for all polynomials of degree less than k. If $k = 1$ then we put $x_1 = 0$. Explicit formulas for the a_i are available and it costs $ck \log k$ steps to compute Q_k^{CC}, see Gentleman (1972).

Though better error bounds are known, we use the simple estimate

$$|S(f) - Q_k^{CC}(f)| \leq \frac{\|f^{(k)}\|_\infty}{2^{k-3}\, k!}$$

which is true for all interpolatory quadrature formulas with positive weights. For $n = l \cdot k$ with $l \in \mathbf{N}$ we consider the respective compound quadrature Q_n with the error bound

$$|S(f) - Q_n(f)| \leq \frac{1}{l^k\, 2^{k-3} k!}$$

for $f \in F^k$. To achieve an error ε it is enough to choose

$$(*) \qquad l(\varepsilon) = \left\lceil \frac{2^{3/k-1}}{(k!)^{1/k}\, \varepsilon^{1/k}} \right\rceil.$$

Even if it is not possible to compute $l(\varepsilon)$ from $(*)$ quickly, we can compute a very good approximation of $l(\varepsilon)$ in constant time using Stirling's formula. This way we obtain an algorithm P such that $|P(\varepsilon,k,f) - \int_{-1}^{1} f(x)\,dx| \le \varepsilon$ for all $\varepsilon > 0$, $k \in \mathbf{N}^+$, and $f \in F^k$. Since $l(\varepsilon)$ is of the order $k^{-1}\varepsilon^{-1/k}$ we finally obtain

$$\mathrm{cost}\,(P,\varepsilon,k,f) \le c_1\,\varepsilon^{-1/k} + c_2 k \log k,$$

see Section 4 for the definition of cost. The constants c_i do not depend on ε, k and $f \in F^k$. It would be interesting to know whether a similar upper bound can be achieved even without the trigonometric functions.

Remarks. a) Of course one could (and should) study even more uniform problems, where also the dimension d for the integration of $f \in C([0,1]^d)$ is part of the input. Now we need an instruction of the form $r_k := f(x)$ with $x \in [0,1]^d$. We may assume, for example, that the coordinates $x_1,\ldots,x_d$ of x are contained in $R_{-4}, R_{-6},\ldots,R_{-2d-2}$ while R_{-2} contains the number d of coordinates. An algorithm P for this problem would compute a number $P(a,f)$ with $|S(f) - P(a,f)| \le \varepsilon$ for any allowed input a and for any allowed f.

b) The difference between Examples 1 and Example 2 or 3 can be stated in the words that we can precompute the quadrature formula in the first example but not in the others. In general, any program can only contain finitely many precomputed values, because it has a finite length.

Another prominent example on precomputation is the evaluation of polynomials. It is known that the Horner scheme is optimal if we want to compute

$$S(c_0,c_1,\ldots,c_n,x) = \sum_{i=0}^{n} c_i x^i.$$

If, however, the polynomial is fixed and we want to evaluate

$$S(x) = \sum_{i=0}^{n} c_i x^i$$

for given c_i then the Horner scheme is not optimal because we can "precompute" a better scheme. Observe again that we consider two different sets of input.

c) In many applications the available "information functionals" are function values $x \mapsto f(x)$ or derivatives. One also can consider other functionals, such as Fourier coefficients. In general we assume that certain linear functionals $I(x,\,.\,)$ with $x \in \mathbf{R}^*$ can be evaluated. So we assume that an instruction $V(k,x)$ with the meaning

$$r_k := I(x,f)$$

is available. If $x = (x_1,\ldots,x_d)$ then d is contained in R_{-2} while the coordinates of x are contained in $R_{-4}, R_{-6},\ldots,R_{-2d-2}$. An instruction $r_k := I(x,f)$ only makes sense, of course, if the meaning of the functional $I(x,\,.\,)$ is defined in advance. We do not assume, however, that such a functional is defined for every $x \in \mathbf{R}^*$.

4. Computations over the reals with an oracle

The mathematical idealization of a computer that is used is a real number URM, as before. The basic new instruction is

$$r_k := I(x, f),$$

where $x \in \mathbf{R}^*$ and $f \in F$. We assume that $x = (x_1, \ldots, x_d)$ can be recognized by looking into some registers: $d, x_1, \ldots, x_d$ are contained in the registers R_{-2}, $R_{-4}, \ldots, R_{-2d-2}$. We usually assume that F is a set of functions, but it is enough to suppose that F is any abstract set. We also assume that $I : (x, f) \mapsto I(x, f) \in \mathbf{R}$ is a partial real function on $\mathbf{R}^* \times F$. The instruction with the response $r_k := I(x, f)$ is denoted by $V(k, x)$.

We want to compute – or to approximate – $S(f) \in \mathbf{R}^*$, where

$$S : F \to \mathbf{R}^*.$$

This operator is called a "solution operator".

The input $a \in \mathbf{R}^*$ may contain some "a priori information" about $f \in F$ or about the problem. A computation $P(a, f)$ can be described similarly as above. We have three kinds of registers. An input $a \in \mathbf{R}^n \subset \mathbf{R}^*$ will be identified with the finite sequence

$$(n, a_1, a_2, \ldots, a_n)$$

in the registers $R_0, \ldots, R_n$. So the registers R_k for $k \geq 0$ are necessary for the input. They also serve for the output. We interpret the final configuration $(b_k)_k$ as the element $b = (b_1, \ldots, b_m)$ of $\mathbf{R}^m$ with $m = b_0$. The vector $P(a, f) = b$ is considered as the result of the computation $P(a, f)$. If $b_0 \notin \mathbf{N}^+$ then we define $P(a, f) = \dagger$, i.e., this is regarded as if the computation would not terminate. Sometimes we have no input at all, i.e., the initial configuration is $r_i = 0$ for all $i \in \mathbf{Z}$. We write $P(f)$ for the computation – and its result – in this case. We have the new instruction $V(k, x)$ which generates the respond

$$r_k := I(x, f),$$

where $x \in \mathbf{R}^*$ and $f \in F$. We assume that $x = (x_1, \ldots, x_d)$ is in the registers $R_{-4}, \ldots, R_{-2d-2}$ and $r_{-2} = d$. Therefore the registers $R_{-2}, R_{-4}, \ldots$ describe the oracle $I(x, .)$ for instructions of the form $V(k, x)$. The registers $R_{-1}, R_{-3}, \ldots$ serve as a free working space. If the instruction $V(k, x)$ cannot be performed because the respective $I(x, k)$ is not defined for f then we put $P(a, f) = \dagger$, as if the computation would not terminate.

Further definitions. Many problems of numerical analysis can be studied within this model of computation. Usually the set F is a set of functions and we want to compute an ε-approximation of $S : F \to \mathbf{R}^*$ or $S^* : F \to G$, see Section 5. Often the error tolerance ε is fixed and we have computations of the type $P(f)$, without any input. Although it is important to construct more "uniform" algorithms, see Examples 2 and 3 from Section 3, lower bounds on the cost function usually are proved for ε-approximations with fixed ε. Often the information consists on function values, i.e., $V(k,x)$ has the meaning

$$r_k := f(x),$$

where x can be found via the registers $R_{-2}, \ldots, R_{-2d-2}$ as above.

We define the cost of computation $P(a,f)$. This definition corresponds to the definition in Section 2 if there are no instructions of the type $V(k,x)$. In this case, $\mathrm{cost}(P,a,f)$ is defined by the number of arithmetic operations and jumps needed to compute $P(a,f)$. The computation of function values may be more expensive and hence we assume that each evaluation of type $V(k,x)$ costs $\mathbf{c}$, where $\mathbf{c} > 0$ and usually $\mathbf{c} > 1$. Assume that k information functionals are evaluated during the computation $P(a,f)$. Then we write

$$\mathrm{cost}(P,a,f) = \mathrm{cost}_c(P,a,f) + \mathrm{cost}_i(P,a,f) = \mathrm{cost}_c(P,a,f) + k\mathbf{c}.$$

The number $\mathrm{cost}_c(P,a,f)$ is called computational cost, while $\mathrm{cost}_i(P,a,f)$ is called informational cost. Often we are interested in $\mathrm{cost}(P,a,f)$ for certain inputs $a \in A \subset \mathbf{R}^*$ and f from $F_a \subset F$, see Example 3 from Section 3. Then we study the worst case cost function

$$\mathrm{cost}(P,a) := \sup_{f \in F_a} \mathrm{cost}(P,a,f).$$

In the case $a = (\varepsilon, d)$ we can ask, for example, whether there is an algorithm P such that

$$\mathrm{cost}(P,a) \le c \cdot d^K / \varepsilon^K$$

with $c, K \in \mathbf{R}^+$. A similar definition of "nonuniform" tractability is studied in Woźniakowski (1994).

5. Problems with values in a vector space

In numerical analysis and information-based complexity we also want to study the computation or approximation of operators

$$S^* : F \to G,$$

where F is a class of functions and G is a vector space, possibly infinitely dimensional. In our model of computation one cannot study such problems directly because we do not allow elements from G during the computation or as an output of an algorithm. Nevertheless it is possible to study operators $S^* : F \to G$ within our model of computation.

It is a familiar procedure to compute a vector $b = P(a, f) \in \mathbf{R}^*$ and to identify the result b as the coefficients of the "really interesting object"

$$\tilde{b} = \sum_{i=1}^{m} b_i g_i \in G.$$

Of course we must agree upon the g_i, the identification of b with $\tilde{b}$ is made outside of the computer by us. To obtain a meaningful result we need $P(a)$ together with its interpretation, given by the g_i. Therefore the pair $(P, (g_i)_i)$ describes the informal algorithm with output $\tilde{b} \in G$, but only $P(a) = b \in \mathbf{R}^*$ is really computed by P. Via such an identification we may say that we are "computing with elements from G".

Remark. One also could study an extension of our model of computation, where we can directly use elements of G. Basically, the following changes are necessary:

i) There is another type of registers that can contain elements g from the vector space G.
ii) In these registers we can perform the basic vector space operations (addition, scalar multiplication) with elements from G.
iii) These registers also are used for the output which is a finite sequence of elements from G.
iv) We have an assignment instruction for these new registers of the form $r_k := g$ and copy instructions.
v) If we want to use linear functionals $L : F \to \mathbf{R}$ then we need still other registers and instructions to deal with these objects.

Observe that every result of such an extended computation is a linear combination of some g_i, where the g_i are explicitly mentioned in an assignment instruction $r_k := g_i$ of the program. Therefore we only can compute finite linear combinations

$$\tilde{b} = \sum_{i=1}^{m} b_i g_i,$$

where the g_i are constants. This means that this model is not more general than ours. Furthermore, there are additional problems in the case when we want to compute

$$\tilde{b} = \sum_{i=1}^{m(a)} b_i g_i$$

if the number $m(a)$ depends on the input a and can be arbitrarily large. In that case one has to assume that one can store infinitely many elements from G.

Because of these problems we prefer the simpler and more realistic model even if the extended model has the following advantage: every program P is complete without any further explanation, because the g_i are contained in the program – they must be known only to the designer of the algorithm. In our model the g_i are not contained in P and must be communicated together with the program.

6. Randomized algorithms

Randomization is being widely used to solve both continuous and discrete problems. For some problems it is known that the stochastic error of optimal randomized methods is much smaller than the error of optimal deterministic methods – where we compare methods with a given cost. This is true, e.g., for the problem of multivariate integration of functions $f \in F$, where F is a class of Hölder functions, see Novak (1988) and Traub et al. (1988).

In the classical literature on Monte Carlo methods it is assumed that we can simulate arbitrary random variables. This means that we have a random number generator which can produce a number $\omega \in [0,1]$ randomly according to the Lebesgue measure. In that model we have another instruction

$r_k := \mathrm{RAN}$

which produces independent random numbers in $[0,1]$. There is some discussion, however, whether the existence of such a general random number generator is realistic or not, see Novak (1988). The classical notion of a Monte Carlo method seems to be too general, it does not confirm to the practical possibilities of computation: only the random sampling out of a finite set seems to be possible. So we may assume that instead of the instruction $r_k := \mathrm{RAN}$ we only have an instruction

$r_k := \mathrm{COIN}$

which produces a random bit, i.e., $r_k \in \{0,1\}$. It is interesting to compare the power of general Monte Carlo methods, using the instruction RAN, with the power of restricted Monte Carlo methods, using only the instruction COIN, see Novak (1988) and Traub and Woźniakowski (1992). We only mention one simple result concerning numerical integration. It is known that restricted Monte Carlo methods fail for arbitrary continuous functions – while the "unrealistic" method

$$f \mapsto \frac{1}{n} \sum_{i=1}^{n} f(\mathrm{RAN})$$

is good for any L_2-function.

7. The complexity classes P and NP

The cost of a computation $P(a)$ for $a \in \mathbf{R}^*$ usually depends on a and is bigger with the size of a. Here the size of a is defined by the number of coordinates, i.e., $\mathrm{size}(a) = n$ for $a \in \mathbf{R}^n$. We define the complexity classes **P** and **NP**. Using two different cost functions, see Section 3, we expect to get two different classes **P** and $\mathbf{P}^*$, or **NP** and $\mathbf{NP}^*$, respectively. We define **P** and **NP**, the definition of $\mathbf{P}^*$ and $\mathbf{NP}^*$ is analogous.

Definition. A total function $f : \mathbf{R}^* \to \mathbf{R}^*$ is called computable in polynomial time if there is a program P and a polynomial p with

$$P(a) = f(a) \quad \text{and} \quad \mathrm{cost}(P, a) \le p(\mathrm{size}(a)) \qquad \text{for all } a \in \mathbf{R}^*.$$

In this case P is called a polynomial time algorithm and we may say that f is a problem with a polynomial time algorithm. We denote by $\mathbf{P}$ the class of decision problems $f : \mathbf{R}^* \to \{0, 1\}$ solvable in polynomial time with respect to cost.

Remarks. a) It is easy to construct $f : \mathbf{R}^* \to \mathbf{R}^*$ such that f is computable in polynomial time with respect to cost, but not with respect to cost*, see Section 3, for example $f(a_1, \ldots, a_n) = (0, 0, \ldots b_{2^n})$ with $b_{2^n} = a_1$. We will prove, however, that the classes of decision problems with respect to both cost functions coincide.

b) There is another important class of decision problems, the class **NP**. First we define this class informally. Assume that a decision problem $f : \mathbf{R}^* \to \{0, 1\}$ is given. In some cases the yes-instances, i.e., the cases with $f(a) = 1$, can be easily checked given a certain certificate $w \in \mathbf{R}^*$. A decision problem f is in **NP** if all yes-instances can be verified in polynomial time given a certificate. The exact definition is as follows.

Definition. A decision problem $f : \mathbf{R}^* \to \{0, 1\}$ is in the class **NP** if there is a program P with inputs $(a, w) \in \mathbf{R}^* \times \mathbf{R}^*$ and a polynomial p such that

a) $P(a, w) \in \{0, 1\}$ for all $(a, w) \in \mathbf{R}^* \times \mathbf{R}^*$;

b) if $f(a) = 0$ then $P(a, w) = 0$ for all $w \in \mathbf{R}^*$;

c) if $f(a) = 1$ then there is a $w \in \mathbf{R}^*$ with $P(a, w) = 1$ and

$$\mathrm{cost}(P, (a, w)) \le p(\mathrm{size}(a)).$$

Remarks. a) For a machine over $\mathbf{R}$ we have only allowed elements of $\mathbf{R}^*$ as inputs, so we have to identify $(a, w) \in \mathbf{R}^* \times \mathbf{R}^*$ with an $z \in \mathbf{R}^*$. Let

$$a = (a_1, \ldots, a_n) \qquad \text{and} \qquad w = (w_1, \ldots, w_m).$$

We define $z \in \mathbf{R}^*$ by

$$z = \alpha(a, w) = (n, a_1, a_2, \ldots, a_n, w_1, w_2, \ldots w_m)$$

and observe that the mapping α is injective and defines a coding which is explicit and effective in the informal sense.

b) The proof technique for the following result is not new. It is contained, for example, in Karpinski and Meyer auf der Heide (1990). In Theorem 1 of this paper the authors prove that each S-RAM can be simulated by a family of S-computation trees with only logarithmic time loss. See also Meyer auf der Heide (1989) for a related survey. I want to thank Mike Shub and F. Meyer auf der Heide for their information about these references.

Result: $\mathbf{P} = \mathbf{P}^*$ and $\mathbf{NP} = \mathbf{NP}^*$

Proof: It is enough to consider the classes $\mathbf{P}$ and $\mathbf{P}^*$, because the proof for the $\mathbf{NP}$-classes is analogous. Assume that $f : \mathbf{R}^* \to \{0,1\}$ is a decision problem with $f \in \mathbf{P}$. Then there is a program P and a polynomial p such that $P(a) = f(a)$ and

$$\mathrm{cost}(P, a) \leq p(\mathrm{size}(a))$$

for all $a \in \mathbf{R}^*$. We want to simulate a computation $P(a)$ by a different program $\widetilde{P}$ such that:

a) $P(a) = \widetilde{P}(a)$ for all $a \in \mathbf{R}^*$;

b) during the computation $\widetilde{P}(a)$ registers R_k with very big $|k|$ are avoided such that $\mathrm{cost}^*(\widetilde{P})$ is polynomially bounded.

Step 1. Since the program P is finite, there is a smallest number $\varrho(P)$ such that none of the registers R_v for $|v| > \varrho(P)$ is explicitly mentioned in P.

This means that every content r_i of a register R_i for $|i| > \varrho(P)$ can only be changed by a copy instruction with indirect addressing. In our modified program $\widetilde{P}$ we have to distinguish between the registers R_i with $-\varrho(P) \leq i \leq \max(\varrho(P), n)$ and other registers. Observe that the registers R_k with $k < -\varrho(P)$ are not used by the input a and also not used by any of the "standard" instructions. Such a register can only be used by a copy instruction. Therefore we use a few of these registers in our modified program $\widetilde{P}$ as a working space. We denote the numbers in this working space by $w_1, \ldots, w_k$. The other registers R_q, R_{q-1} and so on, where $q < -\varrho(P)$ is a suitable constant, can be used for the protocol of the copy instructions. We put $w_1 := q$, later the number w_1 means the "end of the protocol".

Step 2. We replace every instruction

$$r_{r_i} := r_{r_l}$$

by the two instructions

$$w_2 := r_{r_l}$$

and

$$r_{r_i} := w_2.$$

Step 3. Here we explain how to replace the instructions

$$r_{r_i} := w_2.$$

We consider two cases. In the first case we have $-\varrho(P) \leq r_i \leq \max(\varrho(P), n)$. In this case we do not change the instruction. In all other cases the number r_i could be very large and therefore we replace it by the following instructions:

$r_{w_1} := r_i$
$w_1 := w_1 - 1$
$r_{w_1} := w_2$
$w_1 := w_1 - 1.$

Therefore the first instruction of the form $r_{r_i} := w_2$ with $r_i > \max(\varrho(P), n)$ or $r_i < -\varrho(P)$ will be replaced by

$r_q := r_i$
$r_{q-1} := w_2,$

the second instruction of this kind will be replaced by

$r_{q-2} := r_i$
$r_{q-3} := w_2,$

and so on. The register with w_1 serves as a counter as described.

Step 4. Here we deal with the instructions of the form $w_2 := r_{r_l}$. We consider the same two cases. If $-\varrho(P) \leq r_l \leq \max(\varrho(P), n)$ then we do not change the instruction $w_2 := r_{r_l}$. In the other case we must compare r_l with r_q, r_{q-2}, and so on till r_{w_1-2}, in this order. If we find a k with

$$r_l = r_{q-2k}$$

then we put

$$w_2 := r_{q-2k-1}.$$

Of course it is possible that $r_l = r_{q-2k}$ for several of the k. We do the little programming using only the free working space. If we do not find r_l in the registers R_{q-2k} then we put $w_2 := 0$.

In this way we obtain a new program $\widetilde{P}$ with $\widetilde{P}(a) = P(a)$ for all $a \in \mathbf{R}^*$. In the computation $\widetilde{P}(a)$ one step of the computation $P(a)$ is replaced by a number of steps which is at most proportional to the number $\mathrm{cost}(P, a)$. So we get

$$\mathrm{cost}(\widetilde{P}, a) \leq c_1 \cdot \mathrm{cost}(P, a)^2 + c_2.$$

The biggest (absolute) number of a register used during the computation $\widetilde{P}(a)$ is at most proportional to $\mathrm{cost}(P, a)$ or n. Together we obtain

$$\mathrm{cost}^*(\widetilde{P}, a) \leq C_1 \cdot p(\mathrm{size}(a))^3 + C_2$$

if the degree of p is at least 1 and $\mathrm{cost}^*(\widetilde{P}, a) \leq C_1 n + C_2$ if p is a constant.

Acknowledgment. I want to thank Armin Hemmerling, Klaus Meer, Leszek Plaskota, Klaus Ritter, Joe Traub, and Henryk Woźniakowski for valuable remarks and comments. The author is supported by a Heisenberg scholarship of the DFG.

References

1. L. Blum, M. Shub, and S. Smale (1989): On a theory of computation and complexity over the real numbers: NP completeness, recursive functions and universal machines. Bull. of the AMS **21**, 1–46.
2. H. Brass (1977): Quadraturverfahren. Göttingen.
3. N. J. Cutland (1980): Computability. Cambridge University Press.
4. W. M. Gentleman (1972): Implementing Clenshaw-Curtis quadrature. Communications of the ACM **15**(5), 337-346.
5. M. Karpinski, F. Meyer auf der Heide (1991): On the complexity of genuinely polynomial computation. Lecture Notes in Computer Science **452**, 362–368.
6. F. Meyer auf der Heide (1989): On genuinely time bounded computations. Lecture Notes in Computer Science **349**, 1–16.
7. E. Novak (1988): Deterministic and stochastic error bounds in numerical analysis. Lecture Notes in Mathematics **1349**, Springer-Verlag.
8. S. H. Paskov (1993): Average case complexity of multivariate integration for smooth functions. J. Complexity **9**, 291–312.
9. F. P. Preparata, M. I. Shamos (1985): Computational geometry. Springer-Verlag.
10. J. F. Traub, G. W. Wasilkowski, H. Woźniakowski (1988): Information-based complexity. Academic Press.
11. J. F. Traub, H. Woźniakowski (1992): The Monte Carlo algorithm with a pseudorandom generator. Math. of Comp. **58**, 323–339.
12. H. Woźniakowski (1944): Tractability and strong tractability of linear multivariate problems. J. Complexity **10**, 96–128.

Erich Novak
Mathematisches Institut
Universität Erlangen-Nürnberg
Bismarckstr. 1 1/2
91054 Erlangen
Germany
novak@mi.uni-erlangen.de