Globally Convergent Algorithms for DC Operating Point Analysis of Nonlinear Circuits

Duncan A. Crutchley and Mark Zwolinski, *Senior Member, IEEE*

***Abstract*—An important objective in the analysis of an electronic circuit is to find its quiescent or dc operating point. This is the starting point for performing other types of circuit analysis. The most common method for finding the dc operating point of a nonlinear electronic circuit is the Newton–Raphson method (NR), a gradient search technique. There are known convergence issues with this method. NR is sensitive to starting conditions. Hence, it is not globally convergent and can diverge or oscillate between solutions. Furthermore, NR can only find one solution of a set of equations at a time. This paper discusses and evaluates a new approach to dc operating-point analysis based on evolutionary computing. Evolutionary algorithms (EAs) are globally convergent and can find multiple solutions to a problem by using a parallel search. At the operating point(s) of a circuit, the equations describing the current at each node are consistent and the overall error has a minimum value. Therefore, we can use an EA to search the solution space to find these minima. We discuss the development of an analysis tool based on this approach. The principles of computer-aided circuit analysis are briefly discussed, together with the NR method and some of its variants. Various EAs are described. Several such algorithms have been implemented in a full circuit-analysis tool. The performance and accuracy of the EAs are compared with each other and with NR. EAs are shown to be robust and to have an accuracy comparable to that of NR. The performance is, at best, two orders of magnitude worse than NR, although it should be noted that time-consuming setting of initial conditions is avoided.**

***Index Terms*—Circuit simulation, dc circuit analysis, differential evolution, evolution strategies, tournament selection.**

I. Introduction

THE FIRST task in simulating the behavior of a circuit is to find the quiescent or dc operating point. This is important because the operating point is required when performing other types of circuit analysis. For example, the dc operating point is used as the starting point for transient analysis (circuit response in the time domain) [1]. Circuit design algorithms also need the dc operating point of the circuit. In this case, the operating point is required to evaluate the dc performance of the current design under a given set of constraints on the circuit's components [1].

Traditionally, the operating point is found by using the Newton–Raphson method (NR). This method has three potential problems. The first problem is that, at the start of each iteration, we must recompute the Jacobian matrix. The Jacobian matrix contains all the partial derivatives of the nonlinear device equations with respect to the circuit variables, node voltages, and/or branch currents, which is computationally costly. The second problem is that the solution can diverge or fail to converge by oscillating between several potential solutions. This latter situation can occur in circuits with a large amount of feedback.

Finally, convergence is only guaranteed if a suitable initial solution vector is chosen. For circuits with more than one possible solution, the initial guess can influence the final solution and hence finding multiple global solutions is generally difficult. For example, an RS latch, which is a common subcircuit found in computer memory, has three potential solutions. NR-based algorithms will usually only find the metastable solution unless the user intervenes. This solution represents the latch in a balanced state, but it is often more important to know what the circuit does in its two other conjugate state outputs at logic (1,0) in one case and (0,1) in the other. Obviously, these solutions would give three different starting points for a transient analysis. Hence, the ability to find multiple dc operating points, when they exist, can prove very useful for determining the behavior of the circuit over time.

In this paper, we discuss various aspects of evolutionary computing (EC), in particular evolution strategies (ESs), and differential evolution (DE), and how these techniques can be applied to dc circuit analysis. As will be seen, EC has certain advantages over NR. The main benefits are improved convergence and the ability to find multiple solutions. These can be attributed to the parallel nature of EC algorithms, i.e., a search through a population of solutions rather than a sequential search for an individual solution, as in NR. There are further adaptations that can be made to ES, such as more sophisticated mutations and selection schemes. At present, the alternatives to NR are slow, in part because of the way in which the algorithms have been implemented. We will also see that evolutionary algorithms (EAs) can find solutions to circuits that fail (without user intervention) when using NR.

At the end of this paper we will discuss of the results obtained from a SPICE-compatible evolutionary circuit simulator [evolutionary analog circuit simulator (EACS)] that implements versions of the basic EAs, including some more sophisticated features such as tournament selection and higher configurability with regard to the evolutionary operators and how they are used.

Before continuing with a detailed look at the techniques outlined above, we will define the notation that is to be used throughout this work. In the NR algorithm, we represent the trial solutions, e.g., the vector of node voltages and/or branch currents, as a real-valued trial vector $\mathbf{x}^k = (x_1^k, \ldots, x_n^k)^{\mathrm{T}}$, at iteration k. In general, the aim is to find a set of n variables $\mathbf{x}^* = (x_1^*, \ldots, x_n^*)^{\mathrm{T}}$ such that, for some objective function

Manuscript received August 27, 2001; revised January 29, 2002. This work was supported by the Engineering and Physical Sciences Research Council.

The authors are with the Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, U.K.

Digital Object Identifier 10.1109/TEVC.2002.804319

1089-778X/03$17.00 © 2003 IEEE

f, we have $f(\mathbf{x}^*) = 0$. In the case of EAs, the trial vector has an extra superscript i that denotes the ith member of the population. k denotes the generation count and, thus, the trial vector is written as $\mathbf{x}^{i,k} = (x_1^{i,k}, \ldots, x_n^{i,k})^{\mathrm{T}}$. Without loss of generality, we can restrict ourselves to the task of minimization because maximizing a function $f(\mathbf{x}^k)$ is the same as minimizing $-f(\mathbf{x}^k)$, where in general $f(\mathbf{x}^k) \in \Re$ and $\mathbf{x}^k \in \Re^n$. In the case of circuit analysis, the objective function is a vector $\mathbf{f}(\mathbf{x}^k) \in \Re^n$ representing the characteristic equations of the nonlinear circuit components. In EAs, the objective function f represents the *fitness* of a particular trial vector.

It is important at this point to comment on the nature of the problem that is addressed in this paper. NR is used to find the roots of an equation or a set of equations, and as such this is not a minimization problem. We can, however, convert the root-finding problem of operating-point analysis into a minimization problem by specifying a fitness function and attempting to minimize that. This enables us to use EAs, which are search techniques that easily lend themselves to solving minimization problems.

II. Conventional Circuit Analysis Techniques

In this section, we discuss the NR method for dc analysis. Conventionally, one analyzes a circuit to find its node voltages using Kirchhoff's Current Law (KCL) [1]. A node voltage is calculated with respect to a common reference point. Sometimes a branch current is also required; in which case, Kirchhoff's Voltage Law (KVL) is used [1]. A branch current is the current flowing between two nodes in the circuit. Before continuing, it is important to define KCL and KVL. KCL states, "The sum of currents flowing into and out of a node is zero" and KVL states, "The sum of branch voltages around a closed loop in any circuit is zero."

We formulate equations to represent each branch current and apply KCL to sum the currents at each node. Thus, we obtain n simultaneous linear equations which must be solved, as a matrix–vector equation, to find the node voltages. The matrix is often called the nodal admittance matrix, which contains the transconductances (partial derivatives of each device's characteristic equations with respect to the circuit variables, i.e., the Jacobian) of nonlinear devices as well as the conductances of linear devices e.g., resistors. The solution vector contains node voltages and possibly branch currents, and the right-hand side (RHS) vector contains the circuit excitations in the form of current sources. The admittance matrix is normally constructed using *element stamps* [3], which are briefly discussed below. Later we will see how evolutionary methods have advantages over this traditional technique. For example, EAs do not require the Jacobian, and therefore we do not need to solve a matrix–vector system. Hence, we no longer require element stamps.

A. Equation Formulation and Solution

Element stamps are small component-specific tables containing matrix and excitation data [3]. The table indicates the component values to insert in the nodal admittance matrix and in the RHS vector. For example, consider a linear conductance

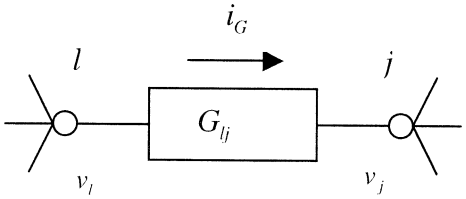

Fig. 1. Conductive component.

TABLE I
MATRIX STAMP FOR CONDUCTANCE

	v_l	v_j	RHS
l	G_{lj}	$-G_{lj}$	
j	$-G_{lj}$	G_{lj}	

TABLE II
MATRIX STAMP FOR FET

	v_D^{k+1}	v_S^{k+1}	v_G^{k+1}	RHS
D	G_{DS}^k	$-G_{DS}^k - G_{GS}^k$	G_{GS}^k	$-I_{DS}^k$
S	$-G_{DS}^k$	$G_{DS}^k + G_{GS}^k$	$-G_{GS}^k$	I_{DS}^k
G				

G_{lj} between two nodes l and j, with node voltages v_l and v_j, Fig. 1.

This component has the stamp shown in Table I.

It is similarly possible to define stamps for nonlinear elements, and such stamps form an efficient method for updating the admittance matrix and RHS vector. Equation (1) gives the branch current for a field-effect transistor (FET) at the $(k+1)^{st}$ NR iteration

$$\begin{aligned} i_{\mathrm{DS}}^{k+1} &= I_{\mathrm{DS}}^k + G_{\mathrm{DS}}^k \cdot v_{\mathrm{DS}}^{k+1} + G_{\mathrm{GS}}^k \cdot v_{\mathrm{GS}}^{k+1} \\ &= i_{\mathrm{DS}}^k + G_{\mathrm{DS}}^k \cdot \left(v_{\mathrm{DS}}^{k+1} - v_{\mathrm{DS}}^k\right) + G_{\mathrm{GS}}^k \cdot \left(v_{\mathrm{GS}}^{k+1} - v_{\mathrm{GS}}^k\right). \end{aligned} \quad (1)$$

where i_{DS}^k denotes the FET drain-source current i_{DS} at the k^{th} iteration. The transconductances G_{DS}^k, G_{GS}^k, and G_{GD}^k are the derivatives of i_{DS}^k with respect to the voltage $v_l^k - v_j^k = v_{lj}^k$, where l and j ($l \neq j$) can be any of D, S, or G (the drain, the source, and the gate). Note that the above derivation is for the forward bias case of the MOSFET. If the MOSFET goes into reverse bias, then D and S are swapped. Table II shows the general element stamp for a FET transistor.

The matrix–vector equations can be solved by Gaussian elimination or a related algorithm, such as LU factorization. There are other improvements that can be made when we have certain types of matrices. In particular, if a matrix is sparse, one only need store the nonzero entries, thus eliminating unnecessary calculations [1].

B. The NR Method

The most common method for nonlinear circuit analysis is the NR method coupled with the Gaussian elimination algorithm. We solve the set of nonlinear equations $\mathbf{f}(\mathbf{x}^k) = \mathbf{0}$ by formulating the linearized matrix vector equation (2) using element stamps

$$\mathbf{G}^k \mathbf{x}^{k+1} = \mathbf{I}^k. \quad (2)$$

Equation (2) is solved to find $\mathbf{x}^{k+1}$, the node voltage vector at the $(k+1)^{\text{st}}$ iteration. The matrix $\mathbf{G}^k$ is the nodal admittance matrix, and the RHS vector $\mathbf{I}^k$ is the vector of excitations. Both are initially set to zero and are updated using element stamps. In fact, to form (2), we need to use the fact that $\mathbf{I}^k = \mathbf{G}^k\mathbf{x}^k - \mathbf{f}(\mathbf{x}^k)$. If we substitute this expression into (2) and multiply both sides by $\mathbf{G}^{-k}$, we get $\mathbf{x}^{k+1} = \mathbf{x}^k - \mathbf{G}^{-k} \cdot \mathbf{f}(\mathbf{x}^k)$, which is the multidimensional form of the standard one-dimensional Newton formula $x_{k+1} = x_k - f(x_k)/f'(x_k)$, where $f'(x_k)$ is the derivative of the device equation with respect to x, evaluated at x_k, the kth approximation to the root of f. By repeatedly solving (2) (using Gaussian elimination or a related technique) and using the solution to formulate the matrix and excitation vector for the next iteration, the solution vector should converge to an accurate representation of the state of the circuit. For further reading, the reader is directed to [1]–[3].

This process is computationally intensive. The matrix equation needs to be set up and solved once per Newton iteration. Building the Jacobian matrix requires the evaluation of the partial derivatives of the device equations. Typically, evaluating the device equations and derivatives, along with the associated matrix operations, requires about 70% or more of the total CPU time [4], [5].

C. Convergence

NR has some inherent problems that manifest themselves quite frequently. When they arise, the algorithm can fail to work correctly. One such problem is NR's sensitivity to the initial values in the solution vector used to start the analysis. This is especially noticeable when we are dealing with nonlinear circuit equations that have multiple solutions. In this case, different initial settings can result in convergence to different solutions or to divergence. In the absence of any other knowledge, the solution vector is initialized to $\mathbf{0}$. This may be simplistic because, if there are multiple solutions, they will be missed and the algorithm may potentially fail to converge. A randomly initialized set of start points may seem better, but the main reason this is not done is that it may yield more failures than successes or repeated occurrences of the same solution, all of which increase the running time. There is, however, a technique called *homotopy* which is a more sophisticated approach to this idea [6]–[8], and which is potentially globally convergent.

The main problem with the selection of the initial values is that one can never be sure of the radius of convergence for a particular problem, and so picking an initial solution that is outside this radius can lead to divergence, or if there are multiple solutions, it could lead to finding a solution other than that being sought. There are several techniques that can be used to help convergence, such as the y-coordinate method [2], the error-function-curvature-driven Newton update correction method [9], damping algorithms [10], the source stepping algorithm [11], and the $G_{\min}$ stepping procedure [12].

In 1977, Ho *et al.* [10] proposed a damping algorithm to aid the convergence of NR. A damping factor is used to aid convergence as follows. The damping factor α in (3) is initialized to a small number such that $0 < \alpha \ll 1$; α is then increased at each iteration until it reaches 1, at which point it remains constant

$$\mathbf{x}^{(k+1)'} = \mathbf{x}^k + \alpha \cdot \left(\mathbf{x}^{k+1} - \mathbf{x}^k\right). \tag{3}$$

The increments of α should be quite large for the first few iterations but should then decrease gradually. Hence, this technique stops sudden large changes that occur between successive approximate solutions, particularly over the first few iterations.

The $G_{\min}$ stepping procedure aids convergence by adding a constant ($G_{\min}$) onto each diagonal component of the Jacobian matrix $\mathbf{G}^k$. $G_{\min}$ is set to a large initial value and is decreased at each iteration to a value of 10^{-9} or less. The constant $G_{\min}$ stops zeros from occurring on the diagonal, which in turn prevents the matrix from becoming singular. This procedure is equivalent to a large resistance (small conductance) being connected between every node in the circuit and ground.

These techniques do not all work for every problem. Hence, commercial simulators employ a number of these techniques and will switch as necessary between them if convergence problems arise. The aim is to develop an EA to find a more general method for solving the majority of circuits.

III. Evolutionary Computing

In general, when using EC techniques, for each member of the *population* $\mathrm{P}^k = \{\mathbf{x}^{1,k}, \ldots, \mathbf{x}^{N,k}\}$, we aim to optimize and in this case minimize, n_{O} objectives (nodal equations) for each individual. N is the size of the population. Here, $\mathbf{x}^{i,k}$ represents the ith trial vector of node voltages for $i = 1, 2, \ldots, N$ at the kth generation for $k = 1, 2, \ldots, K_{\max}$. We then form the *objective vector* denoted by $\mathbf{y}^{i,k} = (y_1^{i,k}, y_2^{i,k}, \ldots, y_{n_O}^{i,k})^{\mathrm{T}}$. We denote the objectives by y_m, $m = 1, 2, \ldots, n_{\mathrm{O}}$. For the purpose of dc analysis, $n_{\mathrm{O}} = n$ (the total number of node voltages and branch currents). In the simplest case, $\mathbf{x}^{i,k}$ contains only node voltages, so we can use these in the evaluation of the device equations. The resulting device currents can then be used to form the KCL equations for each node. Hence, we define y_m as the net current flowing at node m, which is the node's KCL equation.

During the optimization process, we aim to minimize the y_m values; hence, as a trial vector reaches optimality, the y_m values in the corresponding $\mathbf{y}^{i,k}$ vector will be tending toward zero. In other words, the net current flowing into each node should be zero for a perfect solution, and likewise, if KVL is being used, the net voltage around a closed loop should be zero. We minimize the y_m values by minimizing the "fitness" of the solutions. For instance, we could calculate the Euclidean norm of $\mathbf{y}^{i,k}$ and this should be zero for an optimal solution. Hence, by keeping solutions with the better fitnesses, we will steadily be reducing the values in the objectives vectors over successive generations.

By using EAs, we hope to—among other things—find multiple dc operating points of the circuit when they exist. It is important to note that the problem of predicting the number of possible solutions to an arbitrary nonlinear circuit is impossible. The potential number of solutions may be known in advance due to the expertise and experience of the user. Obviously, this is only of use when analyzing variations of known circuits and

device types. Therefore, regardless of which algorithm we employ to find the dc operating points, we can never be sure if there are any solutions left unfound. It is hoped that by the inherent globally convergent properties of EAs, we will find many, if not all, of the solutions when they exist and will not have to rely on user intuition to find the solutions as we do with NR.

A. Fitness Functions

EAs are designed to work on a population of trial vectors and exhibit an implicit parallelism. To enable the processes of evolution to be simulated, it is necessary for each member of the population to be assigned a value representing the worth of the solution. This is called the *fitness* of the individual. The fitness can then be used to decide which trial vectors in the population are to survive from one generation to the next. The general technique is to use $\mathbf{x}^{i,k}$ and other knowledge about the problem to compute $\mathbf{y}^{i,k}$. In the case of dc operating-point analysis, we use the KCL and KVL equations. We use the data in $\mathbf{y}^{i,k}$, which is essentially the error vector for $\mathbf{x}^{i,k}$, to obtain a fitness score for the trial vector in question. The overall optimization procedure then aims to minimize the fitness scores.

In developing our circuit simulator, we implemented five fitness functions but after early investigations, the Euclidean norm was chosen because it provided the best all-round compatibility with the EAs discussed here in terms of convergence times and accuracy. The fitness function is

$$f_{\mathrm{NORM}}(\mathbf{x}^{i,k}) = \mathrm{NORM}(\mathbf{y}^{i,k}) = \sqrt{\sum_{m=1}^{n} \left(y_m^{i,k}\right)^2}. \quad (4)$$

B. ESs

ESs are probabilistic heuristic direct-search optimization techniques, invented independently in 1965 by Rechenberg [13] and Schwefel [14]. They operate at the phenotypic level, which has advantages for real-valued problems because there is no need to define suitable genotype representations and the potentially complex genotype-to-phenotype mapping functions. There is generally no crossover or inversion in ES [or at least not in the same sense as with genetic algorithms (GAs)], so there is not always a need to find cutting points. Sometimes, however, it can be beneficial to have a crossover-like operator. When this is the case, we use *recombination*. The cutting points needed for recombination are simpler than those for GA. In ES, cutting points are equivalent to simply deciding which components of the parents' trial vectors are used to build a recombined intermediate vector $\mathbf{v}$. This kind of recombination is called *discrete recombination* because we are recombining two parents by discretely swapping their vector components, selected at random.

We use a population of size N divided into two pools such that $N = \mu + \lambda$, where the first μ members of the population form the parent pool and the remaining λ members form the offspring pool. There are several variations of ESs. The first is (μ, λ)-ES, where at the end of each generation, the μ best children are taken as the parents for the next generation for $1 \leq \mu < \lambda < \infty$. All other individuals are discarded. Although it can seem like a waste to throw away potentially useful solutions, this form of ES is known to avoid stagnation. In the developmental stages of this work, the use of this ES was explored but was discarded because it incurred longer running times, and so another form of ES was chosen instead. This version is denoted $(\mu + \lambda)$-ES; at the end of a generation, the best μ individuals from the union of the parent and offspring pools are taken as the parents for the next generation. Generally, each parent is required to generate at least one offspring. The exact number depends on the value of λ.

Usually, ESs mutates a parent by adding a Gaussian distributed random vector of mean zero and predefined deviation to it [15] as follows:

$$\tilde{\mathbf{x}}^{i,k} = \mathbf{x}^{i,k} + \mathbf{u}^i. \quad (5)$$

Here, the *mutation vector* $\mathbf{u}^i$ is computed from

$$\begin{aligned} \mathbf{u}^i &= (u_1^i, u_2^i, \ldots, u_n^i)^T \\ u_j^i &= N_j(0, \sigma). \end{aligned} \quad (6)$$

In (6), $\sigma = \tau.\sigma_j^k$ represents a predefined deviation or step size of the mutation vector at generation k, τ is a user-set scale factor, and σ_j^k is the standard deviation of the jth component over the entire population at generation k. The new solution has its fitness evaluated, and if its fitness is better than the mean fitness of the population, then it is included in the offspring pool. This continues until the offspring pool is full.

It can be seen that the basic ES uses the same deviation to generate each variable in all the mutation vectors in a single generation. This is not very realistic because the magnitude of each component of the solution vector can be very different. It can sometimes be better to have a different deviation or step size for each of the components. This can allow for more diversity among the solutions and a better exploration of the solution space [15]. If one implemented this directly, it would involve many user-defined parameters; hence, it is useful if the step sizes can self-adapt, thus letting the algorithm find the best settings [16].

One self-adaptive technique is given as follows:

$$\begin{aligned} u_j^i &= N(0, \sigma_j^{k+1}) \\ \sigma_j^{k+1} &= \sigma_j^k \cdot \exp\left(\tau' \cdot N(0, 1) + \tau \cdot N_j(0, 1)\right). \end{aligned} \quad (7)$$

This provides a different deviation for each variable in $\mathbf{u}^i$. Overall, we form a *deviation vector* $\boldsymbol{\sigma}^{i,k+1}$ for each trial vector $\mathbf{x}^{i,k}$. The variable $N(0, 1)$ is a normalized Gaussian random deviate globally set and regenerated at the start of each generation and $N_j(0, 1)$ is the jth independent normalized Gaussian random deviate. The parameters τ and τ' are defined in (8) [16]. ζ is a user-set scale factor

$$\begin{aligned} \tau &= \zeta \Big/ \sqrt{2n} \\ \tau' &= \zeta \Big/ \sqrt{2\sqrt{n}}\ . \end{aligned} \quad (8)$$

Other similar self-adaptive mutations are possible [15].

In general, ESs use only a mutation operator, but the self-adaptive scheme (ESA) outlined above also uses a recombination operator. One possible recombination operator has already

been mentioned: discrete recombination, but for our ESA algorithm, a different recombination operator has been used as a result of earlier experiments. It works as follows. Two parents vectors $\mathbf{x}^{r_1,k}$ and $\mathbf{x}^{r_2,k}$, $r_1 \neq r_2 \in 1, 2, \ldots, \mu$ are randomly selected before mutation occurs. The recombined intermediate vector $\mathbf{v}$ and the intermediate deviation vector $\overline{\boldsymbol{\sigma}}$ are calculated as

$$\begin{aligned}\mathbf{v} &= \mathbf{x}^{r_1,k} + \rho \cdot (\mathbf{x}^{r_2,k} - \mathbf{x}^{r_1,k}) \\ \overline{\boldsymbol{\sigma}} &= \boldsymbol{\sigma}^{r_1,k} + \rho.(\boldsymbol{\sigma}^{r_2,k} - \boldsymbol{\sigma}^{r_1,k}).\end{aligned} \tag{9}$$

The variable ρ is a uniformly distributed random deviate between 0 and 1. The vector $\mathbf{v}$ becomes the new offspring $\tilde{\mathbf{x}}^{i,k}$, and if its fitness is better than the mean fitness of the population, then it is included in the offspring pool. This continues until the offspring pool is full. When using both mutation and recombination, we first use recombination on the parent pool and then, after generating offspring, we apply mutation to the parent pool and generate further offspring.

C. An EA and Tournament Selection Scheme

The EAs described thus far have had one thing in common, they each use *truncation selection* [15]. This means that the best μ individuals are selected from the union of the parent and offspring pools, which become the set of new parents and are ranked in order of fitness. However, with truncation selection, we do not directly have any control over the selection pressure on an individual. It has been demonstrated that as selection pressure increases convergence time decreases [15], but increasing selection pressure can make it harder to escape from local optima.

Tournament selection is a selection mechanism found in the area of EC called evolutionary programming [15], [17]. At the start of each generation, each member of the population $\mathbf{x}^{i,k}$ is, in turn, compared pairwise with each of γ randomly selected and distinct members of the population, where $1 \leq \gamma \leq \mu$. For each of the γ members, a tally point is added onto a temporary tally score T^i, if $\mathbf{x}^{i,k}$ is fitter than that member. Therefore, any $\mathbf{x}^{i,k}$ can achieve at most $T^i = \gamma$ tally points. Hence, we calculate the selection probability P^i_{sel} for $\mathbf{x}^{i,k}$ as $P^i_{\text{sel}} = T^i/\gamma$, and so $0 \leq P^i_{\text{sel}} \leq 1$. When a parent is required, such as in the recombination or mutation operators, we randomly pick a parent from the population and randomly accept or reject that choice using a coin toss biased according to P^i_{sel}. We can increase the selection pressure by pitting a parent against more population members, e.g., increasing γ. Typically, $\gamma \leq 0.6 \cdot \mu$, where μ is the size of the parent pool.

Here, we apply tournament selection to the ES population described in the previous section. The *tournament selection EA* (TSEA) uses the mutation operator found in the standard ES and it also uses a recombination operator as used in ESA. The algorithm's operation is the same as ESA, except that we have no parameter self-adaptation and, hence, the deviation vector is no longer required, but mutation and recombination operations are carried out in much the same way. We also do not compare the offspring with the population's mean fitness before including them in the offspring pool as with ES and ESA. Instead, we just place the first λ offspring that gets generated, by an operator, directly into the offspring pool. The only other difference is the need to calculate selection probabilities for each trial solution, as discussed above.

By increasing the selection pressure, we increase the level of discrimination made by the algorithm and hence we get a substantial speed increase. This can have the side effect of making the algorithm find only one out of several possible solutions as can be seen later in the experimental results. If the speed increase is sufficient, it is possible to perform multiple runs of the algorithm to find other solutions.

D. Differential Evolution

Storn and Price have described an EA that is self-adaptive, simple, and yet very powerful called differential evolution (DE) [18]. DE is perhaps the simplest EA to implement and to understand out of those described in this paper. It has also been shown [19] to be one of the most robust methods. It has been tested against many other methods, including simulated annealing, adaptive simulated annealing, genetic algorithms, and annealed genetic algorithms, and was found to be at least as good as the other techniques, and in many cases far better.

Several DE schemes have been proposed by Storn [20]; some are more successful than others, and some are problem dependent. Only two schemes will be discussed here: DE1 and DE2 [18]. In DE1, for each trial vector $\mathbf{x}^{i,k}$ in the population, we generate an intermediate vector $\mathbf{v}^i$ as follows:

$$\mathbf{v}^i = \mathbf{x}^{r_1,k} + \tau \cdot (\mathbf{x}^{r_2,k} - \mathbf{x}^{r_3,k}). \tag{10}$$

In (10), τ is a positive real-valued user-set scale factor and r_1, r_2, and r_3 are randomly[1] selected integers in the range $[1, N]$ and are all mutually distinct. The intermediate vector $\mathbf{v}^i$ is then used with $\mathbf{x}^{i,k}$ in a crossover procedure to generate a new offspring $\tilde{\mathbf{x}}^{i,k}$. If $\tilde{\mathbf{x}}^{i,k}$ is fitter than $\mathbf{x}^{i,k}$, then $\mathbf{x}^{i,k+1} = \tilde{\mathbf{x}}^{i,k}$ and we discard $\mathbf{x}^{i,k}$, else we keep $\mathbf{x}^{i,k}$. We generate potential offspring using the following formula:

$$\tilde{\mathbf{x}}^{i,k} = \begin{cases} v^i_j, & \text{for } j = \langle K \rangle_n + 1, \langle K+1 \rangle_n + 1, \ldots, \\ & \langle K + L - 1 \rangle_n + 1 \\ x^{i,k}_j, & \text{otherwise.} \end{cases} \tag{11}$$

In (11), K is a randomly selected integer in the range [0, $n-$ 1] and L is an integer selected from the same range but with the probability $\Pr(L = r) = P^r_c$, where P_c is the user set crossover probability such that $P_c \in [0, 1]$. The notation $\langle K \rangle_n$ denotes the function $K \bmod n$.

DE2 is identical to DE1 except for the generation of the intermediate vector $\mathbf{v}^i$. This time, an additional difference vector is used, as follows:

$$\mathbf{v}^i = \mathbf{x}^{i,k} + \tau' \cdot (\mathbf{x}^{\text{best},k} - \mathbf{x}^{i,k}) + \tau \cdot (\mathbf{x}^{r_1,k} - \mathbf{x}^{r_2,k}). \tag{12}$$

Note that this time we only need two random integers r_1 and r_2, and τ' is positive user-set scale factor. The point of DE2 is that by including the extra difference vector, involving the current generation's best solution, we enhance the greediness of the algorithm.

[1]All the random numbers used in DE are assumed to be uniformly distributed unless stated otherwise.

When using DE there are several rules that, where possible, should be obeyed to improve the performance of the algorithm. For instance, it has been suggested that the initial population should be spread over the full range of the problem variables [20]. P_c should usually be set to a value less than 0.5, but if the algorithm fails to converge, then P_c can be increased to as much as 1.0. As an initial guess, the best population size is usually $N = 10 \cdot n$ and the user should try τ, $\tau' \in [0.5, 1.0]$. Furthermore, as N is increased above $10 \cdot n$, then τ and τ' should be decreased.

IV. DC Analysis Using EC

A. Implementation

The new simulator, EACS, has been built on an existing SPICE-like simulator. The existing simulator uses linked lists and similar data structures to represent the circuit components: the circuit nodes and the sparse network matrix. The EC package uses the existing device models to evaluate branch currents (to calculate the fitness of the solution), and returns newly calculated node voltages into the simulator structure. To some extent, therefore, the use of an existing simulator has compromised the performance of the new solution methods, but on the other hand, there are significant advantages to using existing implementations of complex device models.

In addition to NR, the following solution algorithms may be selected: ES, ESA, DE1, DE2, and TSEA. It is possible to manually set all relevant scale factors and to choose a fitness function from those discussed in section or to use default settings for each algorithm. As in a conventional SPICE simulator, the settings can be applied by setting options in the circuit netlist file. EACS has been tested using a variety of benchmark circuits. A short description of each of these circuits is given in the next subsection and the results of these tests follow.

B. Benchmark Circuits

To test the basic test simulator, several CMOS benchmark circuits were used to evaluate the performance of all of the EAs. SPICE level-3 MOS models were used throughout. Each circuit has one solution, with the inputs described, unless otherwise stated. Circuits such as the latch, the Schmitt trigger, the CMOS inverter, the multiplexers, and the differential pair are often used as benchmark circuits for simulators and the remaining circuits given here are used to test scalability when simulating composite circuits, involving subcircuits of devices such as transmission gates and inverters. Thus, the benchmark circuits are as follows.

- An inverter containing two MOS transistors, a p-type and an n-type.
- A tristate inverter consisting of four MOS transistors.
- An RS latch consisting of two cross-coupled NAND gates (eight MOSFETs in total). The latch inputs, set (S) and reset (R), were both set at logic 1 (or in analog terms at the supply voltage V_{DD}). In this mode, the circuit has three possible solutions: 1) output Q at V_{DD}; 2) Q at 0 V; and 3) the metastable state with Q at approximately $V_{DD}/2$. This circuit is not difficult to simulate with NR but illustrates NR's failure to find multiple solutions without the user assistance.
- A transmission gate XOR with inputs $A = 1$ and $B = 1$. This circuit contains six MOSFETs. The inputs $A = 1$ and $B = 0$ were also tried but NR failed to converge, while the EAs found the correct solution. NR fails on the second configuration due to gain in the circuit and strong positive feedback.
- A transmission gate multiplexer (MUX1) consisting of two transmission gates (four transistors). This circuit should simulate without problem; it is used here to test a composite circuit.
- A tristate inverter multiplexer (MUX2) formed by using tristate and regular inverters and consisting of twelve MOSFETs. This circuit is, again, used to illustrate the use of a larger composite circuit.
- An inverting Schmitt trigger. The Schmitt trigger is made up of five p-type and five n-type MOSFETs. This circuit usually has one solution: the inverse of its input. However, the circuit has hysteresis and when the input voltage is between two critical thresholds the output depends on the previous state of the circuit. In dc analysis, there are two possible solutions as there is no memory of any previous state. This circuit is used to illustrate the failure of NR to find multiple solutions and to show that EAs can be used to simulate a circuit for which NR fails to converge to either solution without user assistance.
- A CMOS differential pair with two nMOS transistors, resistive loads, and a constant current source.
- A one-bit adder that is constructed using a transmission gate XOR (see above) together with two inverters and four transmission gates. The three inputs were set to logic 1. The adder contains 18 transistors. Again, this is another composite circuit and is used to test the scalability of our circuit simulator when using EAs.

C. Experimental Results

Five EAs are implemented in the simulator, along with NR. The NR algorithm employs damping in the form of step-size limiting to assist convergence. Initial values can be set to assist convergence.

The following EC algorithms were used for all the benchmark circuits, with the control settings shown.

- **DE1**: $\tau = 0.4$, $P_c = 0.5$, convergence threshold $= 5 \times 10^{-8}$, population size N (i.e., number of parents) $= 10 \cdot n$ (n is the number of circuit nodes)
- **DE2**: $\tau = 0.8$, $\tau' = 0.9$, $P_c = 0.3$, convergence threshold $= 5 \times 10^{-8}$, $N = 10 \cdot n$
- **ES**: $\tau = 0.7$, Convergence Threshold $= 5 \times 10^{-6}$, $N =$ 50 ("smaller" circuits), 100 ("larger" circuits)
- **ESA**: $\zeta = 1.0$, convergence threshold $= 5 \times 10^{-6}$, $N =$ 150
- **TSEA**: $\tau = 1.5$, convergence threshold $= 5 \times 10^{-7}$, $N =$ 50 ("smaller" circuits), 100 ("larger" circuits), $\gamma = 0.6 \cdot N$.

These settings were found to work well for all circuits. For DE1 and DE2, τ can be varied, and for TSEA it is necessary to vary τ between 1.0 and 2.0 to find multiple solutions. All the EAs are sensitive to the convergence threshold; making the threshold smaller increases the accuracy, but also increases the run time. In fact, the thresholds suggested above were found

TABLE III
RESULTS FOR A CMOS NOT GATE (INPUT = 1)

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1	8	~	5
DE1	1	257	1.19×10^{-6}	170
DE2	1	248	1.16×10^{-6}	110
ES	1	35	1.41×10^{-5}	550
ESA	1	34	1.24×10^{-5}	980
TSEA	1	20	1.42×10^{-5}	90

TABLE IV
RESULTS FOR A CMOS TRISTATE INVERTER

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1	9	~	3
DE1	1	175	2.61×10^{-2}	320
DE2	1	300	1.89×10^{-2}	550
ES	1	93	3.37×10^{-2}	390
ESA	1	149	3.70×10^{-1}	2030
TSEA	1	48	2.19×10^{-2}	270

TABLE V
RESULTS FOR A CMOS RS LATCH ($R = S = 1$)

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1+1+1	10+184+184	~	60+10+10
DE1	3	301	4.39×10^{-2}	1270
DE2	2	376	4.20×10^{-2}	1920
ES	3	43	4.60×10^{-2}	2030
ESA	1+1	247+935	2.27×10^{-2}	20650+4610
TSEA	1+1+1	23+24+32	9.05×10^{-3}	600+660+880

TABLE VI
RESULTS FOR A CMOS XOR GATE ($A = B = 1$)

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1	9	~	5
DE1	1	133	8.72×10^{-3}	440
DE2	1	126	4.57×10^{-3}	490
ES	1	21	1.39×10^{-2}	990
ESA	1	89	1.16×10^{-2}	4720
TSEA	1	14	5.16×10^{-3}	280

by performing several "tune-up" runs of the algorithms across the range of benchmark circuits and these values were found to work well in general. The algorithm halts once the best member of the current population has a fitness less than the threshold.

For the Schmitt trigger and the RS latch, it was necessary to manually set initial conditions to force NR to find all the solutions. By default, NR will find the metastable state for the latch. NR will not converge for the Schmitt trigger—it was necessary to artificially set the input voltage just outside the hysteresis band to find a solution.

The performance for each algorithm with each of the circuits is shown in Tables III–XI. The RS latch (Table V) and the Schmitt trigger (Table IX) have multiple solutions. The algorithm is stated in the first column of each table. In the second column, the number of solutions found by each algorithm automatically is stated as an integer. If multiple solutions could be found by changing settings, this is stated as an expression

TABLE VII
RESULTS FOR A CMOS MULTIPLEXER 1

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1	9	~	4
DE1	1	85	2.55×10^{-2}	110
DE2	1	118	5.06×10^{-2}	110
ES	1	44	1.22×10^{-2}	170
ESA	1	90	1.07×10^{-2}	1100
TSEA	1	15	1.12×10^{-2}	103

TABLE VIII
RESULTS FOR A CMOS MULTIPLEXER 2

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1	10	~	6
DE1	1	132	1.76×10^{-2}	880
DE2	1	100	4.01×10^{-2}	720
ES	1	99	3.63×10^{-2}	1210
ESA	1	235	7.36×10^{-1}	11320
TSEA	1	20	9.98×10^{-3}	600

TABLE IX
RESULTS FOR SCHMITT TRIGGER

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1+1	144+13	~	11+2
DE1	2	270	1.68×10^{-1}	1810
DE2	2	517	1.48×10^{-1}	3350
ES	2	114	2.18×10^{-1}	1430
ESA	1	29	2.01×10^{-1}	490
TSEA	1+1	31+32	8.37×10^{-2}	490+550

TABLE X
RESULTS FOR A CMOS DIFFERENTIAL AMP

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1	28	~	4
DE1	1	292	6.03×10^{-2}	710
DE2	1	152	6.04×10^{-2}	390
ES	1	333	6.04×10^{-2}	980
ESA	1	1502	3.78×10^{-1}	5000
TSEA	1	28	2.07×10^{-2}	490

TABLE XI
RESULTS FOR A CMOS ONE-BIT ADDER

Algorithm	No. of Solutions	No. of Generations or Iterations	Mean Error Per Node	CPU Time (milliseconds)
NR	1	31	~	16
DE1	1	321	5.37×10^{-4}	3080
DE2	1	311	2.09×10^{-4}	3130
ES	1	238	6.04×10^{-4}	3020
ESA	1	275	1.65×10^{-1}	5760
TSEA	1	106	1.54×10^{-3}	2690

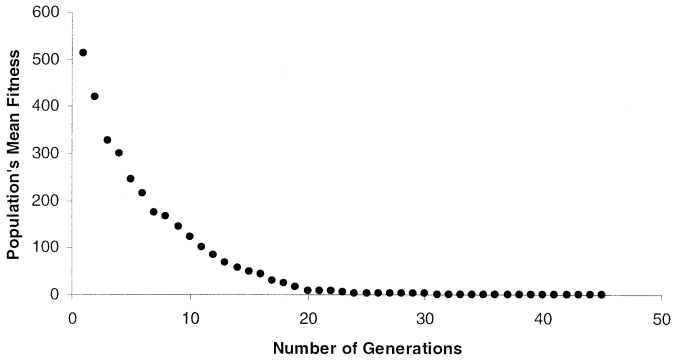

Fig. 2. Convergence graph for DE1 and an RS latch.

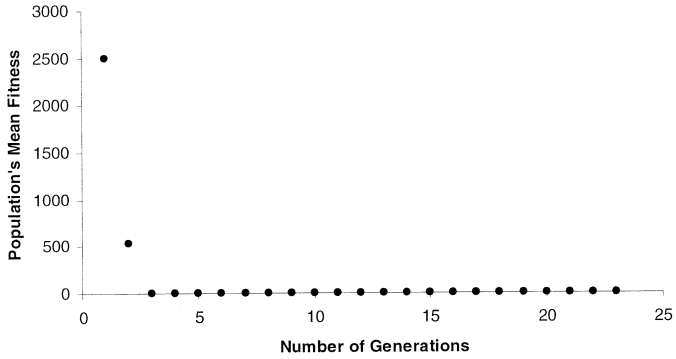

Fig. 3. Convergence graph for TSEA and an RS latch.

(e.g., $1 + 1$ means two solutions were found by restarting the algorithm). The third column shows the number of generations (or for NR, the number of iterations). The fourth column shows the accuracy compared with the solution found by NR (assumed to be the most accurate). In the case of multiple solutions, the mean error across all solutions is stated. Finally, the CPU time in milliseconds is given. The benchmark tests were run on a PC workstation with an 800-MHz 586 CPU and 256 MB of RAM, running Windows NT. Again, if multiple runs were needed to find multiple solutions, this is stated as a sum. As well as monitoring the best solution of the current generation, various other values were monitored at each generation to provide an idea of the performance of the EAs. In particular, the mean fitness, mean solution, and the standard deviation of the population were obtained and stored in a separate results file for later study. This data gives an insight into the convergence of the EAs. Fig. 2 illustrates the progression of DE1 for an RS latch, with respect to the mean fitness of the population. (The analysis takes 301 generations, but the graph shows only the first 45 generations.) The initial population's mean fitness was $\approx 5.12 \times 10^2$ and the final population's mean fitness was $\approx 1.08 \times 10^{-6}$. As another example of the convergence behavior of the EAs, Fig. 3 shows the convergence of TSEA for an RS latch, but unlike DE1, only one solution was found for a single run of TSEA. In this case, the initial population's mean fitness was $\approx 2.50 \times 10^3$ and the final population's mean fitness was $\approx 5.82 \times 10^{-7}$. A similar pattern can be found among the data collected for the other algorithms and circuits.

DE1 and ES are the best algorithms for finding multiple solutions automatically. ESA is least good at finding multiple solutions, even when restarted.

NR is always the fastest, in terms of CPU time (but note the comment above concerning the manual intervention needed to find multiple solutions). For circuits with a single solution, TSEA is always the fastest of the EAs in terms of the number of generations and in terms of CPU time, with the exception of the RS latch, where DE1 is fastest. ESA is consistently the slowest (apart from for the Schmitt trigger circuit, where it only found one solution). The best-performing EAs are, however, between 16 and 170 times slower than NR.

The accuracy of the EAs is very similar. TSEA or DE2 are the most accurate in all cases except MUX1, when ESA is best. It must be noted that all these accuracy figures are relative to NR, and are not absolute errors. The error is calculated as the mean of the difference between the NR solution(s) and the EA solution(s).

In general, therefore, DE1, DE2, ES, and TSEA are accurate and robust in terms of convergence and the number of solutions found. Accuracy and speed can be gained at the expense of finding multiple solutions. Although NR is always fast, it may depend on the user setting the initial state of the solution vector. The EAs are more likely to succeed from arbitrary starting points. Therefore, the CPU time does not necessarily represent the total effort required to find a solution. This is particularly true when multiple solutions exist and are sought. It can therefore be argued that the best algorithms in terms of accuracy, speed, and the ability to find multiple solutions and to analyze problem circuits, such as the Schmitt trigger, are DE1 and DE2.

V. Conclusions

The use of EAs for nonlinear operating-point analysis of MOS circuits has been demonstrated. It has been shown that EAs, and particularly DE and TSEA, have some significant advantages over conventional NR. DE and the other EAs are globally convergent, whereas NR is only locally convergent. NR requires manual intervention to find all the solutions to a circuit; it has been shown that DE can find multiple solutions in a single pass. An important property of EAs is that they can find multiple solutions in a single pass, but this can sometimes take significantly longer than using NR to find a single solution. It is important to get a good balance between speed, accuracy, and the number of solutions found. We can often make improvements to an EA that, for instance, increases the speed of the algorithm, but this can have side effects. For example, if we increase the amount of discrimination an algorithm makes with regards to selecting parents, then this gives a speed increase along with improved accuracy, i.e., TSEA. We have seen, however, that there is a significant side effect in losing the ability to find multiple solutions. Hence, if TSEA is to be a suitable alternative to NR, improvements to the algorithm must be made to give it the ability to find all the solutions in a single pass.

All of the EAs are sensitive, by varying degrees, to reproduction parameters, such as the mutation rate, population size,

recombination strategies, etc. The success of DE is partly due to its self-adaptive nature, and although DE uses mutation as a primary operator, it also contains a recombination operator so as to not neglect the benefits of sexual reproduction. Another excellent feature of the DE algorithms is that the population size is automatically scaled in proportion to the size of the given problem, which can help avoid over- and under-sized populations. These features and the way they are implemented in DE have been the major contribution to DE's good performance.

All of the EAs here are slow compared with NR, even though the Jacobian matrix is not constructed. This can be attributed to two factors. First, a significant amount of sorting of populations has to be done. This accounts for the majority of the CPU time taken. For example, in a $(\mu+\lambda)$-ES, with typical values of $\mu = 100$ and $\lambda = 200$, we will have it so the parent pool is ordered fittest first and the offspring pool is in no particular order. Hence, we will need to reorder a population that may not be close to being correctly ordered and with 300 members, as with the example above; this is not a trivial task. The sorting algorithm used, in this version of EACS, is simple and based on the insertion sort. A better choice of algorithm, such as one based on the quick sort algorithm, would produce a significant speed up. Secondly, because the device models have been inherited from an earlier simulator, they evaluate both the current (as required for EC) and the partial derivatives, which are not required. If the models were modified to remove these unnecessary calculations, we would expect the time taken for device evaluation to be approximately halved. Therefore, overall we can reasonably expect that the EAs can be speeded up by at least an order of magnitude. This would make them very competitive with NR. Having demonstrated the computational effectiveness of using evolutionary computation for circuit analysis, the next phase of this research will seek to increase the speed of the algorithms.

As well as increasing the speed, we will also endeavor to improve the accuracy. It is felt that the best way to approach this is by way of a hybrid method with the NR algorithm. In other words, a population can be searched in a fairly coarse way using an EA, perhaps DE1, and then the solution can be refined using NR. This will provide accuracy almost identical to NR and should also reduce the number of generations needed to reach convergence. Much larger benchmark circuits, up to 100 nodes, will be constructed and used to test the performance of the EAs when simulating such large circuits.

ACKNOWLEDGMENT

The authors would like to thank Dr. Z. R. Yang for suggesting the original idea that led to this research.

REFERENCES

[1] V. Litovski and M. Zwolinski, *VLSI: Circuit Simulation and Optimization*. London, U.K.: Chapman and Hall, 1997.
[2] D. A. Calahan, *Computer Aided Network Design*, revised ed. New York: McGraw-Hill, 1972.
[3] C. W. Ho, A. E. Ruehli, and P. A. Brennan, "The modified nodal approach to network analysis," *IEEE Trans. Circuits and Simulation*, vol. CAS-22, pp. 504–509, June 1975.
[4] P. F. Cox, R. G. Burch, D. E. Hocevar, P. Yang, and B. D. Epler, "Direct circuit simulation algorithms for parallel processing," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 714–725, June 1991.
[5] T. A. Johnson and D. J. Zukowski, "Waveform-relaxation-based circuit simulation on the Victor V256 parallel processor," *IBM J. Res. Develop.*, vol. 35, no. 5/6, Sept./Nov. 1991.
[6] R. C. Melville, L. Trajković, and L. T. Watson, "Artificial parameter homotopy methods for the DC operating point problem," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 861–877, June 1993.
[7] L. Trajković, "Homotopy methods for computing DC operating points," School of Eng. Sci., Simon Fraser Univ., Burnaby, BC, Canada, no. 2526, 1996.
[8] D. M. Wolf and S. R. Sanders, "Multiparameter homotopy methods for finding DC operating points of nonlinear circuits," *IEEE Trans. Circuits Syst. I*, vol. 43, pp. 824–838, Oct. 1996.
[9] E. Ngoya, J. Rousset, and J. J. Obregon, "Newton–Raphson iteration speed-up algorithm for the solution of nonlinear circuit equations in general purpose CAD programs," *IEEE Trans. Computer-Aided Design*, vol. 16, pp. 638–643, June 1997.
[10] C. W. Ho, D. A. Zien, A. E. Ruehli, and P. A. Brennan, "An algorithm for DC solutions in an experimental general purpose interactive circuit design program," *IEEE Trans. Circuits and Simulation*, vol. CAS-24, Aug. 1977.
[11] C. G. Broyden, "A new method of solving nonlinear simultaneous equations," *Comput. J.*, vol. 12, pp. 94–99, 1969.
[12] T. N. Najibi, "Continuation methods as applied to circuit simulation," *IEEE Circuits Devices Mag.*, vol. 5, pp. 48–49, 1989.
[13] I. Rechenberg, *Cybernetic Solution Path of an Experimental Problem*. Farnborough, U.K.: Ministry of Aviation, Royal Aircraft Establishment, Aug. 1965, Library Translation no. 1122.
[14] H.-P. Schwefel, "Kybernetische Evolution als Stategie der Experimentellen Forschung in der Strömungstechnik," Diploma thesis, Tech. Univ. Berlin, Berlin, Germany, 1965.
[15] D. B. Fogel, *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*, 2nd ed. New York: IEEE Press, 2000.
[16] T. Bäck and H.-P. Schwefel, "An overview of evolutionary algorithms for parameter optimization," *Evolut. Comput.*, vol. 1:1, pp. 1–23, 1993.
[17] L. J. Fogel, "Autonomous Automata," *Indust. Res.*, vol. 4, pp. 14–1, 1962.
[18] R. Storn and K. Price, "Differential evolution: A simple and efficient adaptive scheme for global optimization over continuous spaces," ICSI, Berkeley, CA, Tech. Rep. TR-95-012, 1995.
[19] ——, "Minimizing the real functions of the ICEC'96 contest by differential evolution," in *Proc. Int. Conf. Evolutionary Computing*, Nagoya, Japan, 1996, pp. 842–844.
[20] R. Storn, "On the usage of differential evolution for function optimization," ICSI, Berkeley, CA, Tech. Rep., 1996.

Duncan A. Crutchley received the Master's degree in pure mathematics (with specialization in elliptic curve cryptography) in 1999 from the University of Southampton, Southampton, U.K., in 1999, where he is currently working toward the Ph.D. degree in electronic engineering.

During 1999, he was with Philips Semiconductors, Southampton, U.K. During his time at Philips, he developed elliptic curve cryptosystems for smartcards and IEEE1394 FireWire devices. His research interests are in the area of globally convergent numerical methods for circuit analysis and using such methods to develop a SPICE-like analysis tool. He has published several conference papers on this topic.

Mark Zwolinski (M'92–SM'00) received the B.Sc. and Ph.D. degrees in electronics from the University of Southampton, Southampton, U.K., in 1982 and 1986, respectively.

He is a Senior Lecturer in the Department of Electronics and Computer Science, University of Southampton. His research interests include simulation and modeling algorithms for analog and mixed-signal integrated circuits, fault simulation, high-level synthesis, and test synthesis. He has co-authored over 90 research papers in technical journals and conferences.

Dr. Zwolinski is Chair of the Fault Simulation Study Group of the IEEE Design Automation Technical Committee.