```
for k=1:size, x=X(:,k); z(k)=fitnessf(A,b,c,x,m); end
%Vector z gives fitness of each of the members in a generation
```

S.K. Sen and Sagar Sen, A shrinking-rectangle randomized algorithm with interpolation for a complex zero of a function, **Proc. 47th Congress of ISTAM (An International Meet)**, **Indian Institute of Technology, Guwahati,** Dec 23-26, 2002, 72-80.

A Shrinking-rectangle Randomized Algorithm with Interpolation for a Complex Zero of a Function

S.K. Sen[1] and Sagar Sen[2]

Abstract A polynomial-time deterministic randomised algorithm is described to compute a zero of a complex/real polynomial or a complex/real transcendental function in a complex plane. The algorithm starts with a specified rectangle enclosing a complex zero, shrinks it successively by at least 50% in each iteration somewhat like a two-dimensional bisection, and then a single application of linear two-variable interpolation in the highly shrunk rectangle provides the required zero. A parallel implementation of this algorithm is discussed while its sequential and parallel computational complexities as well as its space complexity are discussed. The algorithm is found to be reasonably good for zero clusters and also for multiple zeros. This method can be extended to minimize globally a polynomial or a transcendental function of several variables without resorting to the computation of its partial derivatives and can be used along with the deflation of the polynomial or with different specified initial rectangle

Keyword Complex zero, deterministic randomised algorithm, linear interpolation, shrinking-rectangle, two-dimensional bisection

1.**Introduction**

There exist several deterministic nonrandomized polynomial-time algorithms in the literature (Krishnamurthy and Sen 2001, Mathews 1994, Schilling and Harries 2002) to compute real and complex roots of an algebraic or a transcendental equation. Specifically, automatic two-dimensional (2-D) bisection methods (Sen and Lord 1990, Wilf 1978) have been described. These bisection methods needing only function computations provide excellent accuracy for well-conditioned functions, i.e., functions without zero-clusters (closely spaced zeros). A function with distinct well-separated zeros or one with multiple zeros is well-conditioned with respect to its zero finding using a bisection method. We present here a

[1]Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, India
[2] Department of Computer Science and Engineering, Dr. Ambedkar Institute of Technology, Bangalore 560056, India

variation of the 2-D bisection method, called here the SRA algorithm, that differs widely from these methods in its conceptual approach. We specify a reasonably small rectangle in the complex plane so that it contains one of the complex zeros of the given one-variable function. We throw uniformly distributed darts, say 100 of them, onto the rectangle and compute at each point of hit the value of the function. This dart-throwing is accomplished by generating uniformly distributed pairs of pseudo-random numbers each representing a point in the rectangle. We select the point of hit, that corresponds to the minimum value of the norm of the function; replace the rectangle by one whose area is at least 50% less than the previous one. This smaller rectangle now has in it the zero of the function. We continue this process a number of times, say 10 times, and obtain the highly shrunk rectangle which contains this zero. Carry out a two-variable linear interpolation to obtain the zero with sufficient accuracy. To obtain other complex zeros of the function, one can deflate the function (algebraic) and follow the proposed algorithm on the deflated function or one can choose a different rectangle that would contain the second zero of the function (algebraic or trigonometric) and apply the algorithm. This algorithm can be modified for a function of several variables by generating a set of several pseudorandom numbers instead of the pair and by using multi-variable interpolation to obtain a global minimum of the function. We will not discuss this aspect here.

The algorithm with the justification of each of its steps is described in Sec. 2 while its computational and space complexities are discussed in Sec. 3. A Matlab program that not only takes care of complex zeros with nonzero real and nonzero imaginary parts but also purely real as well as purely imaginary zeros is presented in Sec. 4. Test examples and conclusions are included in Secs. 5 and 6, respectively.

2. **The SRA Algorithm**

Let f(x) be an algebraic or a transcendental function of the single variable x and be continuous and bounded. The function f(x) may be real or complex. The problem is to find a complex zero of f(x), where the zero may have (i) both real and imaginary parts nonzero or may be (ii) only real or (iii) only imaginary. It can be seen that the two-variable interpolation needed for Case (i) is not applicable to any of Cases (ii) and (iii). In the later two cases, we have to explicitly use distinct single variable interpolations.

Let the real part of the zero of the continuous bounded function f(x) lie in the interval [a, b] while the imaginary part in the interval [c, d]. These intervals define a rectangle D (Fig. 1a), i.e. the domain of search. The function values corresponding to each of the points of the domain will define a plane (i.e., a hyperplane of dimension 2) R whose sides are in general curved (not straight lines) (Fig. 1b). Assume that this domain contains only one zero for the sake of observing how the algorithm works, such an assumption is not strictly needed though. This fact will be seen when we consider numerical test examples that illustrate the algorithm.

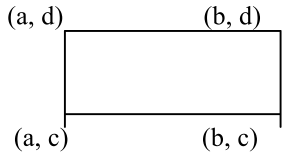

Fig. 1a Rectangle D containing a zero of a function (Domain space)

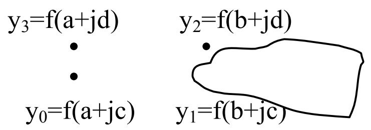

Fig. 1b Plane R with curved sides (Function space)

We now describe the steps of the algorithm with justification/explanation for each step.

S. 1 *Throwing darts onto the rectangle* D *and choosing the one nearest zero* Generate uniformly distributed n_l (n_l = 20, say) ordered pairs of random numbers such that the first number of each pair lies in [a, b] while the second lies in [c, d]. Each pair defines a complex random number. The first one of the pair defines the real part while the second one the imaginary part of this complex random number. Let rand(1) produce a random number in (0, 1). Then x = [rand(1)(b – a) + a] + j [rand(1)(d – c) + c] is a complex random number inside the rectangle D. This number corresponds to a thrown dart inside D and the corresponding f(x) will have a numerical value which is usually complex and which will lie inside the plane R. Corresponding to n_l such complex random numbers within the rectangle D, we will have n_l function values within the plane R. We choose that random number which produces the minimum absolute value of the function. Call this number x_1. Clearly this absolute value of f(x_1) will be nearest the zero of f(x).

S. 2 *Shrinking the Rectangle D by at least 50%* Set b_1 := real(x_1) + .354(b – a), a_1:= real(x_1) – 0.354(b – a), where real(x_1) is the real part of x_1. If $b_1 < b$ then set b := b_1; if $a_1 > a$ then set a := a_1.

Similarly, set d_1 := imag(x_1) + .354(d – c), c_1 := imag(x_1) – (d – c), where imag(x_1) is the imaginary part of x_1. If $d_1 < d$ then set d := d_1; if $c_1 > c$ then set c := c_1.

The step S. 2 reduces the rectangle D by at least half its size. The new rectangle will enclose the zero of f(x) assuming that it is not too violently fluctuating or the zeros are not too closely spaced.

S. 3 *Getting the smallest rectangle after k iterations* Repeat the steps S. 1 and S. 2 for k (k = 10, say) times. This step will produce a highly shrunk rectangle that contains the zero of f(x).

S. 4 *Two-variable interpolation for a complex zero* Use the two-variable Lagrange linear interpolation using the most recent values of a, b, c, d and the corresponding function values. This interpolation includes extrapolation automatically. Let (x_i, y_i) i = 0(1)3 be the table for interpolation, where x_i as well as y_i are both complex and the interpolation (that includes extrapolation too) problem is posed as follows.

x	**y**
$x_0 = a + jc$	$y_0 = f(x_0)$
$x_1 = b + jc$	$y_1 = f(x_1)$
$x_2 = b + jd$	$y_2 = f(x_2)$

$x_3 = a + jd$ $y_3 = f(x_3)$

$x = ?$ $y = f(x) = 0$

Hence, if $a \neq 0$, $b \neq 0$, $a \neq b$, $d_1 = y_0 - y_1 \neq 0$, $d_2 = y_0 - y_2 \neq 0$, $d_3 = y_0 - y_3 \neq 0$, $d_4 = y_1 - y_2 \neq 0$, $d_5 = y_1 - y_3 \neq 0$, $d_6 = y_2 - y_3 \neq 0$, $d_7 = y_1y_2$, $d_8 = y_1y_3$, $d_9 = y_2y_3$, then

$$x = - x_0y_1d_9/(d_1d_2d_3) + x_1y_0d_9/(d_1d_4d5) - x_2y_0d_8/(d_2d_4d_6) + x_3y_0d_7/(d_3d_5d_6) \qquad (1)$$

This interpolation is carried out only once in the final highly shrunk rectangle. The x thus obtained is the required zero of the function f(x).

Interpolation for computing only a real zero The foregoing interpolation formula (1) is not valid for obtaining a real zero of f(x) since $y_0 = y_3$ and $y_1 = y_2$ and consequently d_3 and d_4 both are zero and each one occurs in the denominator in the formula (1). Therefore, we use the modified interpolation formula

$$x = -x_0y_1/d_1 + x_1y_0/d_1 \quad (\textit{for real zeros only}) \qquad (2)$$

Interpolation for computing only an imaginary zero The formula (1) is invalid here too. The modified interpolation formula is

$$x = - x_0y_3/d_3 + x_3y_0/d_3 \ (\textit{for imaginary zeros only}) \qquad (3)$$

The x that we obtain in the formula (1) or (2) or (3) is the required solution. The corresponding function value f(x) will be sufficiently small so that the zero x could be accepted as the required zero for all practical purposes.

S. 5 *Error in (quality of) the zero x* "How good is the quality of the zero?" is a very pertinent question that is almost always asked. The answer is obtained through computing a relative error (i.e., error-bound) in the zero x. Observe that an absolute error is not much meaningful in numerical computation. In the absence of the knowledge of the exact zero (solution) which is never known (for if it is numerically known then we do not bring error unnecessarily into the scene), we consider usually the solution (zero) of higher order accuracy for the exact solution. Thus the error in the solution of lower order accuracy will be computed, denoting the solution of higher order accuracy = x_h and the solution of lower order accuracy = x_t, as

$$E_r = (x_h - x_t)/x_h \qquad (4)$$

Clearly $|f(x_h)| < |f(x_t)|$ by at least an order (Sen 2002). If we consider the interpolated zero (solution) x as the zero (x_t) of lower order accuracy then we do not have the zero (x_h) of higher order accuracy. To determine x_h, we shrink the already highly shrunk rectangle *once more* and carry out the interpolation as in the step S. 4. This interpolated zero will be the zero (x_h) of higher order accuracy. Thus we can compute the relative error E_r. The step S. 5 has not been included in the MATLAB program for physical conciseness and for better comprehension. The reader may achieve this step of error computation by running the program for the second time replacing k by k + 1 and obtaining the zero x_h of higher order accuracy. Otherwise, he may automate the program by appropriately modifying it.

3. Computational and Space Complexities

The *computational complexity* of the SRA algorithm can be derived as follows.

To generate n_l pairs of random numbers using the multiplicative congruential generator or, equivalently, the power residue method (Banks et al. 1998), we need $2n_l$ multiplications and $2n_l$ divisions (to carry out mod operations). To obtain n_l complex random numbers in the specified rectangle D (Fig.1a), we need further $2n_l$ multiplications and $2n_l$ additions. If we do not distinguish between a division and a multiplication then so far we need $6n_l$ *real multiplications* and $2n_l$ *real additions* for generating n_l complex random numbers. If the function f(x) is a *polynomial of degree* n, then the computation of f(x) using the nested multiplication scheme (Krishnamurthy and Sen 2001) would need n complex multiplications and n complex additions, i.e., 2n real multiplications and 2n real additions for each complex random number. Hence, for n_l complex random numbers, we need $2n \times n_l$ *real multiplications* + $2n \times n_l$ *real additions.* Since we have k rectangles before we reach the smallest one we need, for the computation of the *smallest rectangle*, $6k \times n_l + 2k \times n \times n_l$ *multiplications* and $2k \times n_l + 2k \times n \times n_l$ *additions*. Since k, n_l are independent of the size n of the function f(x), our computational complexity will $\mathbf{O(2k \times n_l \times n)}$ assuming n very large (compared to n_l and k, and the size of the program) but finite. A typical value of k is 10 and that of n_l is 20. These values, however, will be larger if the initial rectangle chosen is larger.

The *space complexity*, i.e., the storage space needed to store the input data, viz., the (n + 1) complex coefficients of the nth degree polynomial f(x), we need 2n locations. We also need the storage space to store the program. Since the storage space for the program is independent of the size, i.e., the degree n of f(x), the space complexity is simply **O(2n)** assuming n very large but finite.

If the function f(x) is a *transcendental function* then the computational complexity will be O(2k $\times$ n_l $\times$ number of operations needed to compute f(x)) while the space complexity will be the space needed for the function. Observe that the transcendental function though may be written as a polynomial of degree $\propto$, does not have the computational complexity $O(\propto)$ nor has the space complexity $O(\propto)$.

These complexities are comparable with those of other existing methods. The space complexity as well as the computational complexity in terms of the input size n for all these methods will not be usually $O(n^s)$, where $s > 1$.

The *parallel computational complexity* using n processors will clearly depend only on the values of n_l and k. If we use $p < n$ processors then the complexity will increase proportionately. The space complexity, however, will remain unchanged.

4. MATLAB Program for the SRA Algorithm

This program is self-explanatory and computes a complex zero of a polynomial or a transcendental function.

```
function[]=func2(rmin, rmax, imin, imax, nmax, eps, fun)
%func2 computes a complex zero of a function fun
%using a randomized algorithm with an interpolation

%Description of input parameters rmin, rmax, imin, imax, etc.

%[rmin, rmax]=interval of real part of the zero.
%[imin, imax]=interval of imaginary part of the zero.
%nmax=maximum no, of bisections (nmax=10 usually;
%for better accuracy, nmax may be taken as 20 0r 30.
%eps=.5*10^-4 usually; for better accuracy, eps=.5*10^-8.
%However, eps is used here as a relative error term and
%should be chosen compared to the input quantities involved.
%fun is the function, one of whose zeros is to be obtained.
%For example, fun='x^2+x+1' for the function f(x)=x^2+x+1.

for k=1:10
   %This number 10 implies that the original rectangle is
   % shrunk successively 10 times. This number seems reasonably
   %good; however, it may be increased depending on the accuracy
   % needed within the limit of the precision of the computer.

xvect=[];fvect=[]; absfvect=[];
for i=1:nmax
   x=(rand(1)*(rmax-rmin)+rmin)+j*(rand(1)*(imax-imin)+imin);
   f=eval(fun); absf=abs(f);
   xvect=[xvect;x];
   fvect=[fvect;f];
   absfvect=[absfvect; absf];
end;

x_f_absf=[xvect fvect absfvect];
x_f_absf_s=sortrows(x_f_absf, 3);
string 'sorted x, f(x), absolute f(x)'
x_f_absf_s
if abs(x_f_absf_s(1,3))<eps
   string 'root, function-value, absolute function value'
   x_f_absf_s(1,:)
   break
end;

x1=x_f_absf_s(1,1);
realdiff=rmax-rmin; imagdiff=imax-imin;
rmax1=real(x1)+0.354*realdiff; rmin1=real(x1)-0.354*realdiff;
if rmax1<rmax
   rmax=rmax1;
end;
if rmin1>rmin
   rmin=rmin1;
end;
```

```
   imax1=imag(x1)+0.354*imagdiff; imin1=imag(x1)-0.354*imagdiff;
   if imax1<imax
      imax=imax1;
   end;
   if imin1>imin
      imin=imin1;
   end;
   string 'rmax,rmin,imax,imin'
   rmax,rmin,imax,imin
end;

a=rmin; b=rmax; c=imin; d=imax;

%The foregoing statements reduce the rectangle to maximum half its
size.
%This reduction has resemblance with 2-D bisection for a complex zero.


   x=a+j*c; x0=x; y0=eval(fun); x=b+j*c; x1=x; y1=eval(fun);
   x=b+j*d; x2=x; y2=eval(fun); x=a+j*d; x3=x; y3=eval(fun);
   d1=y0-y1; d2=y0-y2; d3=y0-y3;d4=y1-y2; d5=y1-y3;d6=y2-y3;

   d7=y1*y2; d8=y1*y3; d9=y2*y3;

   if abs(d1)<eps, d1=1; end; if abs(d2)<eps, d2=1; end;if abs(d3)<eps,
d3=1; end;
   if abs(d4)<eps, d4=1; end; if abs(d5)<eps, d5=1; end;if abs(d6)<eps,
d6=1; end;


       xx0=-x0*y1*d9/(d1*d2*d3);
       xx1=-x1*y0*d9/(-d1*d4*d5);
       xx2=-x2*y0*d8/(d2*d4*d6);
       xx3=-x3*y0*d7/(-d3*d5*d6);

       if abs(c)<eps & abs(d)<eps, xx0=-x0*y1/d1; xx1=x1*y0/d1;xx2=0;
xx3=0; end;
       %This statement is for interpolation for only real zeros.

       string 'x0, yo, x3,y3,d3'
       % Imaginary x0 & x3 and corresponding y0 & y3 for linear
interpolation
       x0,y0,x3,y3,d3

       if abs(a)<eps & abs(b)<eps, xx0=-x0*y3/d3;xx3=x3*y0/d3;xx1=0;
xx2=0;  end;
      %This statement is for inperpolation for only imaginary zeros.



       x=xx0+xx1+xx2+xx3;
   f=eval(fun); absf=abs(f);

   string 'interpolated (including extrapolated) zero, f-value, abs f-
value'
```

```
   x, f, absf

 if absf<eps
   string 'root,f-value, abs_f-value (correct up to 1/eps digits)'
   x, f, absf
   break
end;
```

5. Test Examples

To check the SRA algorithm, we have constructed several typical test functions (i.e., functions whose zeros are exactly known using the MATLAB function **poly**.

Example 1 (A real quadratic function with complex zeros) $f(x) = x^2 + x + 1$, whose exact roots are $-0.5 + i\sqrt{3}$ and $-0.5 - i\sqrt{3}$, where $i = \sqrt{-1}$.

The *inputs* are

rmin=-1;rmax=0;imin=0;imax=1;nmax=10;eps=.5*10^-4;fun='x^2+x+1';
func2(rmin,rmax,imin,imax,nmax,eps,fun)

The *outputs* are

x = -0.5000 + 0.8660i, f = -1.9222e-006 +2.8105e-007i, absf =1.9426e-006

Example 2 (A real quatric polynomial with only real zeros) $f(x) = x^4 - 5.2x^3 + 10.04x^2 - 8.528x + 2.688$ whose exact zeros are 1, 1.2, 1.4, and 1.6 and which is constructed using the MATLAB command **poly([1 1.2 1.4 1.6])**.

The *inputs* are

rmin=0;rmax=1.19;imin=0;imax=0;nmax=10;eps=.5*10^-8;fun='x^4-5.2*x^3+10.04*x^2-8.528*x+2.688';
func2(rmin,rmax,imin,imax,nmax,eps,fun)

The *outputs* are

x = 1.2015, f = 2.3317e-005, absf = 2.3317e-005.

The second run of the program with the same inputs produced the *outputs*

x = 1.1998, f = -3.0973e-006, absf = 3.0973e-006

The third run with the same inputs produced the *outputs*

```
x = 1.1998,  f = -3.0969e-006,  absf = 3.0969e-006
```

The fourth run with the same inputs resulted in the *outputs*

```
x = 1.0016,  f = -7.5359e-005,  absf = 7.5359e-005
```

Example 3 (A quatric real polynomial having only imaginary zeros) f(x) = $x^4 + 5x^2 + 4$ whose exact zeros are − i, i, − 2i, and 2i.

The *inputs* are

```
rmin=0;rmax=0;imin=-1.5;imax=-.5;nmax=10;eps=.5*10^-4;fun='x^4+5*x^2+4';
func2(rmin,rmax,imin,imax,nmax,eps,fun)
```

The *outputs* are

```
x = 0 - 1.0000i,  f = -1.4188e-004,  absf = 1.4188e-004.
```

*Example 4 (A quatric complex polynomial with **zero-clusters**: a **highly ill-conditioned** problem)* f(x) = $x^4 - (8.04 + .22j)x^3 + (24.2227 + 1.3266j)x^2 - (32.410446 + 2.665828j)x + (16.25009862 + 1.78524984j)$ whose exact zeros are 2.01 + j.04, 2.01 +j.05, 2.01 + j.06, and 2.01 +j.07, where $j = \sqrt{-1}$.

The *inputs* are

```
rmin=2;rmax=2.019;imin=0;imax=0.045;nmax=10;eps=.5*10^-8;

» fun='x^4-(8.04+.22*j)*x^3+(24.2227+1.3266*j)*x^2-
(32.410446+2.665828*j)*x+(16.25009862+1.78524984*j)';
func2(rmin,rmax,imin,imax,nmax,eps,fun)
```

The *outputs* are

```
x = 2.0112 + 0.0470i,  f = -6.5411e-009 -2.3059e-009i,  absf = 6.9356e-009
```

When the program was rerun for the second time with the same foregoing inputs, the *outputs* became

```
x = 2.0110 + 0.0519i,  f = 3.5115e-009 -1.4762e-009i,  absf =  3.8092e-009
```

The foregoing results seem reasonably good for the precision of 15 digits.

Example 5(A quatric real polynomial with somewhat closely spaced real zeros) f(x) = $x^4 - 9x^3 + 30.35x^2 - 45.45x + 25.5024$ whose exact zeros are 2.1, 2.2, 2.3, and 2.4

The *inputs* are

```
rmin=2;rmax=2.19;imin=0;imax=0;nmax=10;eps=.5*10^-8;fun='x^4-9*x^3+30.35*x^2-45.45*x+25.5024';
func2(rmin,rmax,imin,imax,nmax,eps,fun)
```

The *outputs* are

```
x = 2.1997,  f = -5.4080e-007,   absf =  5.4080e-007.
```

When the program was rerun with the foregoing inputs, the *outputs* became

```
x = 2.1997,  f = -5.3846e-007,  absf =  5.3846e-007.
```

When the program was rerun for the third time with the same inputs the *outputs* became

```
x = 2.1001,  f =  -5.8460e-007,  absf = 5.8460e-007
```

Examples 6 (A tenth degree real polynomial with large coefficients and distinct zeros) $f(x) = x^{10} - 55x^9 + 1320x^8 - 18150x^7 + 157773x^6 - 902055x^5 + 3416930x^4 - 8409500x^3 + 12753576x^2 - 10628640x + 3628800$ whose zeros are 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10.

The *inputs* are

```
» func2(.1,1.5,0,0,10, .5*10^-8,'x^10-55*x^9+1320*x^8-18150*x^7+157773*x^6-
902055*x^5+3416930*x^4-8409500*x^3+12753576*x^2-10628640*x+3628800')
```

The *outputs* are

```
x = 1.0000,  f = -9.4986,  absf = 9.4986
```

When the program was rerun with the same polynomial with changed *inputs*

```
» func2(.1,11,0,0,10, .5*10^-8,'x^10-55*x^9+1320*x^8-18150*x^7+157773*x^6-
902055*x^5+3416930*x^4-8409500*x^3+12753576*x^2-10628640*x+3628800')
```

we obtained the *outputs*

```
x = 6.0000, f = -0.0570,  absf = 0.0570
```

When we reran the program with the foregoing inputs, then the *outputs* became

```
x = 5.9977,  f = -6.5615,  absf = 6.5615
```

For the third run with the foregoing inputs, we got *outputs* as

```
x = 4.0030,  f = 12.9012, absf = 12.9012
```

For the fourth run with the foregoing inputs, the *outputs* became

```
x = 5.9988, f = -3.5642,   absf = 3.5642
```

For the fifth run we got *outputs* as

```
x = 4.0015,  f = 6.6179,  absf = 6.6179
```

6. Conclusions

Shrinking rectangle converges faster than 2-D bisection. When k goes to 10, the initial rectangle D (Fig. 1a) that encloses/contains a zero of the function f(x) will be shrunk to the rectangle whose area will be less than or equal to $D/2^k = D/2^{10} = 0.00097656D$. This shrinking is significantly rapid compared to the automatic bisection for complex zeros (Sen and Lord 1990, Wilf 1978).

The non-existence of a zero in the wrongly chosen initial rectangle can be detected. The SRA algorithm will come out indicating that the chosen rectangle D does not contain a zero if the choice is incorrect, i.e., if it really does not contain a zero.

Interpolation (including extrapolation) is carried out in the final highly shrunk rectangle only once It is possible to interpolate linearly in each of the k (= 10) rectangles. However, it is not done because the linear interpolation could be sufficiently inaccurate when the rectangle is large. Moreover, such repeated interpolations will not only increase the computation but also might result in excluding the actual zero in the rectangle-shrinking process.

The zero existing in the initial rectangle D will exist in the final shrunk rectangle. In our numerical experiment with numerous functions and with reasonably chosen initial rectangle D, the zero that was located in D always remained in the final shrunk rectangle. The SRA algorithm thus seems an efficient fail-proof complex zero finding method and it is *deterministic*.

The SRA algorithm is not worse than most algorithms for finding a zero in a zero-cluster. A function having zero-clusters (closely spaced zeros) is always an ill-conditioned problem with respect to finding a zero accurately in the cluster. Any method so far existing as well as any method that could be proposed in future would be only satisfactory to a varying extent for a specified precision. Our numerical experiment depicts that the SRA algorithm is reasonably good when dealing with zero-clusters.

Multiple zeros do not pose any problem to the SRA algorithm. Unlike the Newton method and its variations which need to compute derivatives of a function and in which an oscillation around a multiple zero (in a finite precision machine) sets in, the SRA algorithm has absolutely no such problem. It gives, like bisection methods, the multiple zero accurately as it does not depend on the computation of the derivatives of a function. For a polynomial having multiple zeros, *repeated deflations* will provide the order of multiplicity.

Use deflation or different rectangles to seeve out all the zeros. One way of seeving **all the zeros** of a polynomial with or without multiple zeros is to deflate the polynomial successively after computing a zero. The other way is to choose different appropriate intervals/rectangles each enclosing a zero and compute all the zeros. For a transcendental function that cannot be written as the product of a polynomial (with multiple zeros) and another transcendental function, deflations may not be useful.

The SRA algorithm has a sequential complexity O(n) and its parallel implementation is straight-forward. As we have seen in Sec. 3 that the SRA algorithm has a sequential computational complexity $O(2k \times n1 \times n)$ where the input size is $O(2n)$ for an nth degree complex polynomial. Observe that k (=10, say) and n_1 (=10 or 20, say) are independent of n. The parallel computational complexity, when we have n processors, is $O(k \times n_1)$ which is independent of the input size. For a fixed number of processors $< n$, this complexity will increase proportionately.

The SRA algorithm can be extended to obtain the global minimum of a multi-variable function. Instead of generating a pair of pseudorandom numbers for a complex zero of a function f(x), we have to generate an ordered set of pseudorandom numbers for this purpose and suitably modify this algorithm.

References

Banks, J.;Carson, J.S., II; Nelson, B.L., Discrete-event Simulation (2nd ed.), Prentice-Hall of India, New Delhi, 1998.

Krishnamurthy, E.V.; Sen, S.K., Numerical Algorithms: Computations in Science and Engineering, Affiliated East-West Press, New Delhi, 2001.

Mathews, J.H., Numerical Methods for Mathematics, Science, and Engineering, 2nd ed., Prentice-Hall of India, New Delhi, 1994.

Schilling, R.J.; Harries, S.L., Applied Numerical Methods for Engineers, using MATLAB and C, Thomson Asia Pvt. Ltd., Singapore, 2002.

Sen, S.K., Error and Computational Complexity in Engineering, in J.C. Misra (ed.), **Computational Mathematics, Modelling and Algorithms** as Chap. 5, Narosa Publishing House, New Delhi, 2002, 110-145.

Sen, S.K.; Lord, E.A., An automatic bisection to compute complex zeros of function, in S. Bandyopadhyay (ed.), **Information Technology: Key to Progress,** Tata-McGraw-Hill, New Delhi, 1990, 9-13.

Wilf, H., A global bisection method for computing the zeros of a polynomial in the complex plane, J. ACM, 415-420, 1978.

Arial Font of the foregoing paper with probably some minor changes

A Shrinking-rectangle Randomized Algorithm with Interpolation for a Complex Zero of a Function

S.K. Sen
Supercomputer Computer Education and Research Centre
Indian Institute of Science, Bangalore 560 012, India
e-mail: sksen@serc.iisc.ernet.in Fax:091-80-3602648

Sagar Sen
Department of Computer Science and Engineering
Dr. Ambedkar Institute of Technology, Bangalore 560 056, India
e-mail: sagarsen@rediffmail.com

ABSTRACT

A polynomial-time deterministic randomised algorithm is described to compute a zero of a complex/real polynomial or a complex/real transcendental function in a complex plane. The algorithm starts with a specified rectangle enclosing a complex zero, shrinks it successively by at least 50% in each iteration somewhat like a two-dimensional bisection, and then a single application of linear two-variable interpolation in the highly shrunk rectangle provides the required zero. A parallel implementation of this algorithm is discussed while its sequential and parallel computational complexities as well as its space complexity are presented. The algorithm is found to be reasonably good for zero clusters and also for multiple zeros. This method can be extended to minimize globally a polynomial or a transcendental function of several variables without resorting to the computation of its partial derivatives and can be used along with the deflation of the polynomial or with different specified initial rectangle

1. INTRODUCTION

There exist several deterministic nonrandomized polynomial-time algorithms in the literature (Krishnamurthy and Sen 2001, Mathews 1994, Schilling and Harries 2002) to compute real and complex roots of an algebraic or a transcendental equation. Specifically, automatic two-dimensional (2-D) bisection methods (Sen and Lord 1990, Wilf 1978) have been described. These bisection methods needing only function computations provide excellent accuracy for well-conditioned functions, i.e., functions without zero-clusters (closely spaced zeros). A function with distinct well-separated zeros or one with multiple zeros is well-conditioned with respect to its zero-finding using a bisection method. We present here a variation of the 2-D bisection method, called here the SRA algorithm, that differs widely from these methods in its conceptual approach. We specify a reasonably small rectangle in the complex plane so that it contains one of the complex zeros of the given one-variable function. We throw uniformly distributed darts, say 10 or 20 of them, onto the rectangle and compute at each point of hit the value of the function. This dart-throwing is accomplished by generating uniformly distributed pairs of pseudo-random numbers each representing a point in the rectangle. We select the point of hit, that corresponds to the minimum value of the norm of the function; replace the rectangle by one whose area is at least 50% less than the previous one. This smaller rectangle now has in it the zero of the function. We continue this process a number of times, say 10 times, and obtain the highly shrunk rectangle which contains this zero. Carry out a two-variable linear interpolation to obtain the zero with sufficient accuracy. To obtain other complex zeros of the function, one can deflate the function (algebraic) and follow the proposed algorithm on the deflated function or one can choose a different rectangle that would contain the second zero of the function (algebraic or trigonometric) and apply the algorithm. This algorithm can be modified for a function of several variables by generating a set of several pseudorandom numbers instead of the pair and by using multi-variable interpolation to obtain a global minimum of the function. We will not discuss this aspect here.

The algorithm with the justification of each of its steps is described in Sec. 2 while its computational and space complexities are discussed in Sec. 3. A Matlab program that not only takes care of complex zeros with nonzero real and nonzero imaginary parts but also purely real as well as purely imaginary zeros is presented in Sec. 4. Test examples and conclusions are included in Secs. 5 and 6, respectively.

2. The SRA Algorithm

Let f(x) be an algebraic or a transcendental function of the single variable x and be continuous and bounded. The function f(x) may be real or complex. The problem is to find a complex zero of f(x), where the zero may have (i) both real and imaginary parts nonzero or may be (ii) only real or (iii) only imaginary. It can be seen that the two-variable interpolation needed for Case (i) is not applicable to any of Cases (ii) and (iii). In the later two cases, we have to explicitly use distinct single variable interpolations.

Let the real part of the zero of the continuous bounded function f(x) lie in the interval [a, b] while the imaginary part in the interval [c, d]. These intervals define a rectangle D (Fig. 1a), i.e. the domain of search. The function values corresponding to each of the points of the domain will define a plane (i.e., a hyperplane of dimension 2) R whose sides are in general curved (not straight lines) (Fig. 1b). Assume that this domain contains only one zero for the sake of observing how the algorithm works, such an assumption is not strictly needed though. This fact will be seen when we consider numerical test examples that illustrate the algorithm.

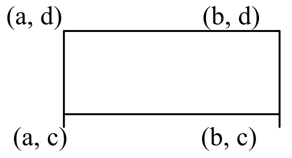

Fig. 1a Rectangle D containing a zero of a function (Domain space)

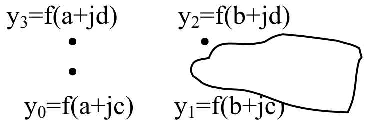

Fig. 1b Plane R with curved sides (Function space)

We now describe the steps of the algorithm with justification/explanation for each step.

S. 1 *Throwing darts onto the rectangle* D *and choosing the one nearest zero* Generate uniformly distributed n_1 (n_1 = 20, say) ordered pairs of random numbers such that the first number of each pair lies in [a, b] while the second lies in [c, d]. Each pair defines a complex random number. The first one of the pair defines the real part while the second one the imaginary part of this complex random number. Let rand(1) produce a random number in (0, 1). Then x = [rand(1)(b – a) + a] + j [rand(1)(d – c) + c] is a complex random number inside the rectangle D. This number corresponds to a thrown dart inside D and the corresponding f(x) will have a numerical value which is usually complex and which will lie inside the plane R. Corresponding to n_1 such complex random numbers within the rectangle D, we will have n_1 function values within the plane R. We choose that random number which produces the minimum absolute value of the function. Call this number x_1. Clearly this absolute value of f(x_1) will be nearest the zero of f(x).

S. 2 *Shrinking the Rectangle D by at least 50%* Set b_1 := real(x_1) + .354(b – a), a_1:= real(x_1) – 0.354(b – a), where real(x_1) is the real part of x_1. If b_1 < b then set b := b_1; if a_1 > a then set a := a_1. Similarly, set d_1 := imag(x_1) + .354(d – c), c_1 := imag(x_1) – (d – c), where imag(x_1) is the imaginary part of x_1. If d_1 < d then set d := d_1; if c_1 > c then set c := c_1.

The step S. 2 reduces the rectangle D by at least half its size. The new rectangle will enclose the zero of f(x) assuming that it is not too violently fluctuating or the zeros are not too closely spaced.

S. 3 *Getting the smallest rectangle after k iterations* Repeat the steps S. 1 and S. 2 for k (k = 10, say) times. This step will produce a highly shrunk rectangle that contains the zero of f(x).

S. 4 *Two-variable interpolation for a complex zero* Use the two-variable Lagrange linear interpolation using the most recent values of a, b, c, d and the corresponding function values. This interpolation includes extrapolation automatically. Let (x_i, y_i) i = 0(1)3 be the table for interpolation, where x_i as well as y_i are both complex and the interpolation (that includes extrapolation too) problem is posed as follows.

x	$x_0 = a + jc$	$x_1 = b + jc$	$x_2 = b + jd$	$x_3 = a + jd$	$x = ?$
y	$y_0 = f(x_0)$	$y_1 = f(x_1)$	$y_2 = f(x_2)$	$y_3 = f(x_3)$	$y = f(x) = 0$

Hence, if $a \neq 0$, $b \neq 0$, $a \neq b$, $d_1 = y_0 - y_1 \neq 0$, $d_2 = y_0 - y_2 \neq 0$, $d_3 = y_0 - y_3 \neq 0$, $d_4 = y_1 - y_2 \neq 0$, $d_5 = y_1 - y_3 \neq 0$, $d_6 = y_2 - y_3 \neq 0$, $d_7 = y_1y_2$, $d_8 = y_1y_3$, $d_9 = y_2y_3$, then

$$x = - x_0y_1d_9/(d_1d_2d_3) + x_1y_0d_9/(d_1d_4d5) - x_2y_0d_8/(d_2d_4d_6) + x_3y_0d_7/(d_3d_5d_6) \qquad (1)$$

This interpolation is carried out only once in the final highly shrunk rectangle. The x thus obtained is the required zero of the function f(x).

Interpolation for computing only a real zero The foregoing interpolation formula (1) is not valid for obtaining a real zero of f(x) since $y_0 = y_3$ and $y_1 = y_2$ and consequently d_3 and d_4 both are zero and each one occurs in the denominator in the formula (1). Therefore, we use the modified interpolation formula

$x = -x_0y_1/d_1 + x_1y_0/d_1$ (*for real zeros only*) (2)

Interpolation for computing only an imaginary zero The formula (1) is invalid here too. The modified interpolation formula is

$x = - x_0y_3/d_3 + x_3y_0/d_3$ (*for imaginary zeros only*) (3)

The x that we obtain in the formula (1) or (2) or (3) is the required solution. The corresponding function value f(x) will be sufficiently small so that the zero x could be accepted as the required zero for all practical purposes.

S. 5 *Error in (quality of) the zero x* "How good is the quality of the zero?" is a very pertinent question that is almost always asked. The answer is obtained through computing a relative error (i.e., error-bound) in the zero x. Observe that an absolute error is not much meaningful in numerical computation. In the absence of the knowledge of the exact zero (solution) which is never known (for if it is numerically known then we do not bring error unnecessarily into the scene), we consider usually the solution (zero) of higher order accuracy for the exact solution. Thus the error in the solution of lower order accuracy will be computed, denoting the solution of higher order accuracy = x_h and the solution of lower order accuracy = x_t, as

$E_r = (x_h - x_t)/x_h$ (4)

Clearly $|f(x_h)| < |f(x_t)|$ by at least an order (Sen 2002). If we consider the interpolated zero (solution) x as the zero (x_t) of lower order accuracy then we do not have the zero (x_h) of higher order accuracy. To determine x_h, we shrink the already highly shrunk rectangle *once more* and carry out the interpolation as in the step S. 4. This interpolated zero will be the zero (x_h) of higher order accuracy. Thus we can compute the relative error E_r. The step S. 5 has not been included in the MATLAB program for physical conciseness and for better comprehension. The reader may achieve this step of error computation by running the program for the second time replacing k by k + 1 and obtaining the zero x_h of higher order accuracy. Otherwise, he may automate the program by appropriately modifying it.

3. COMPUTATIONAL AND SPACE COMPLEXITIES

The *computational complexity* of the SRA algorithm can be derived as follows. To generate n_1 pairs of random numbers using the multiplicative congruential generator or, equivalently, the power residue method (Banks et al. 1998), we need $2n_1$ multiplications and $2n_1$ divisions (to carry out mod operations). To obtain n_1 complex random numbers in the specified rectangle D (Fig.1a), we need further $2n_1$ multiplications and $2n_1$ additions. If we do not distinguish between a division and a multiplication then so far we need $6n_1$ *real multiplications* and $2n_1$ *real additions* for generating n_1 complex random numbers. If the function f(x) is a *polynomial of degree* n, then the computation of f(x) using the nested multiplication scheme (Krishnamurthy and Sen 2001) would need n complex multiplications and n complex additions, i.e., 2n real multiplications and 2n real additions for each complex random number. Hence, for n_1 complex random numbers, we need $2n \times n_1$ *real multiplications* + $2n \times n_1$ *real additions.* Since we have k rectangles before we reach the smallest one we need, for the computation of the *smallest rectangle*, $6k \times n_1 + 2k \times n \times n_1$ *multiplications* and $2k \times n_1 + 2k \times n \times n_1$ *additions*. Since k, n_1 are independent of the size n of the function f(x), our computational complexity will **$O(2k \times n_1 \times n)$** assuming n very large (compared to n_1 and k, and the size of the program) but finite. A typical value of k is 10 and that of n_1 is 20. These values, however, will be larger if the initial rectangle chosen is larger.

The *space complexity*, i.e., the storage space needed to store the input data, viz., the (n + 1) complex coefficients of the nth degree polynomial f(x), we need 2n locations. We also need the storage space to store the program. Since the storage space for the program is independent of the size, i.e., the degree n of f(x), the space complexity is simply **$O(2n)$** assuming n very large but finite.

If the function f(x) is a *transcendental function* then the computational complexity will be $O(2k \times n_1 \times$ number of operations needed to compute f(x)) while the space complexity will be the space needed for the function. Observe that the transcendental function though may be written as a polynomial of degree $\propto$, does not have the computational complexity $O(\propto)$ nor has the space complexity $O(\propto)$.

These complexities are comparable with those of other existing methods. The space complexity as well as the computational complexity in terms of the input size n for all these methods will not be usually $O(n^s)$, where $s > 1$.

The *parallel computational complexity* using n processors will clearly depend only on the values of n_1 and k. If we use $p < n$ processors then the complexity will increase proportionately. The space complexity, however, will remain unchanged.

4. MATLAB PROGRAM FOR THE SRA ALGORITHM

This program is self-explanatory and computes a complex zero of a polynomial or a transcendental function.

```
function[]=func2(rmin, rmax, imin, imax, nmax, eps, fun)
%func2 computes a complex zero of a function fun
%using a randomized algorithm with an interpolation

%Description of input parameters rmin, rmax, imin, imax, etc.

%[rmin, rmax]=interval of real part of the zero.
%[imin, imax]=interval of imaginary part of the zero.
%nmax=maximum no, of bisections (nmax=10 usually;
%for better accuracy, nmax may be taken as 20 0r 30.
%eps=.5*10^-4 usually; for better accuracy, eps=.5*10^-8.
%However, eps is used here as a relative error term and
%should be chosen compared to the input quantities involved.
%fun is the function, one of whose zeros is to be obtained.
%For example, fun='x^2+x+1' for the function f(x)=x^2+x+1.

for k=1:10
  %This number 10 implies that the original rectangle is
  % shrunk successively 10 times. This number seems reasonably
  %good; however, it may be increased depending on the accuracy
  % needed within the limit of the precision of the computer.

xvect=[];fvect=[]; absfvect=[];
for i=1:nmax
  x=(rand(1)*(rmax-rmin)+rmin)+j*(rand(1)*(imax-imin)+imin);
  f=eval(fun); absf=abs(f);
  xvect=[xvect;x];
  fvect=[fvect;f];
  absfvect=[absfvect; absf];
end;

x_f_absf=[xvect fvect absfvect];
x_f_absf_s=sortrows(x_f_absf, 3);
```

```
string 'sorted x, f(x), absolute f(x)'
x_f_absf_s
if abs(x_f_absf_s(1,3))<eps
   string 'root, function-value, absolute function value'
   x_f_absf_s(1,:)
   break
end;

x1=x_f_absf_s(1,1);
realdiff=rmax-rmin; imagdiff=imax-imin;
rmax1=real(x1)+0.354*realdiff; rmin1=real(x1)-0.354*realdiff;
if rmax1<rmax
   rmax=rmax1;
end;
if rmin1>rmin
   rmin=rmin1;
end;

   imax1=imag(x1)+0.354*imagdiff; imin1=imag(x1)-0.354*imagdiff;
   if imax1<imax
      imax=imax1;
   end;
   if imin1>imin
      imin=imin1;
   end;
   string 'rmax,rmin,imax,imin'
   rmax,rmin,imax,imin
end;

a=rmin; b=rmax; c=imin; d=imax;

%The foregoing statements reduce the rectangle to maximum half its size.
%This reduction has resemblance with 2-D bisection for a complex zero.

   x=a+j*c; x0=x; y0=eval(fun); x=b+j*c; x1=x; y1=eval(fun);
   x=b+j*d; x2=x; y2=eval(fun); x=a+j*d; x3=x; y3=eval(fun);
   d1=y0-y1; d2=y0-y2; d3=y0-y3;d4=y1-y2; d5=y1-y3;d6=y2-y3;

   d7=y1*y2; d8=y1*y3; d9=y2*y3;

   if abs(d1)<eps, d1=1; end; if abs(d2)<eps, d2=1; end;if abs(d3)<eps, d3=1; end;
   if abs(d4)<eps, d4=1; end; if abs(d5)<eps, d5=1; end;if abs(d6)<eps, d6=1; end;

       xx0=-x0*y1*d9/(d1*d2*d3);
       xx1=-x1*y0*d9/(-d1*d4*d5);
       xx2=-x2*y0*d8/(d2*d4*d6);
       xx3=-x3*y0*d7/(-d3*d5*d6);

       if abs(c)<eps & abs(d)<eps, xx0=-x0*y1/d1; xx1=x1*y0/d1;xx2=0; xx3=0; end;
       %This statement is for interpolation for only real zeros.

       string 'x0, yo, x3,y3,d3'
       % Imaginary x0 & x3 and corresponding y0 & y3 for linear interpolation
       x0,y0,x3,y3,d3

       if abs(a)<eps & abs(b)<eps, xx0=-x0*y3/d3;xx3=x3*y0/d3;xx1=0; xx2=0;  end;
```

```
    %This statement is for inperpolation for only imaginary zeros.

     x=xx0+xx1+xx2+xx3;
  f=eval(fun); absf=abs(f);

  string 'interpolated (including extrapolated) zero, f-value, abs f-value'
  x, f, absf

 if absf<eps
  string 'root,f-value, abs_f-value (correct up to 1/eps digits)'
  x, f, absf
  break
end;
```

5. TEST EXAMPLES

To check the SRA algorithm, we have constructed several typical test functions (i.e., functions whose zeros are known through the MATLAB function **poly**). To conserve space we present here just four examples.

Example 1 (A real quatric polynomial with only real zeros) $f(x) = x^4 - 5.2x^3 + 10.04x^2 - 8.528x + 2.688$ whose exact zeros are 1, 1.2, 1.4, and 1.6 and which is constructed using the MATLAB command **poly([1 1.2 1.4 1.6])**. The *inputs* are

```
rmin=0;rmax=1.19;imin=0;imax=0;nmax=10;eps=.5*10^-8;fun='x^4-5.2*x^3+10.04*x^2-
8.528*x+2.688';
func2(rmin,rmax,imin,imax,nmax,eps,fun)
```

The *outputs* are x = 1.1998, f = -3.0969e-006, absf = 3.0969e-006. The second run with the same inputs resulted in the *outputs* x = 1.0016, f = -7.5359e-005, absf = 7.5359e-005

Example 2 (A quatric real polynomial having only imaginary zeros) $f(x) = x^4 + 5x^2 + 4$ whose exact zeros are – i, i, – 2i, and 2i. The *inputs* are

```
rmin=0;rmax=0;imin=-1.5;imax=-.5;nmax=10;eps=.5*10^-4;fun='x^4+5*x^2+4';
func2(rmin,rmax,imin,imax,nmax,eps,fun)
```

The *outputs* are x = 0 - 1.0000i, f = -1.4188e-004, absf = 1.4188e-004.

*Example 3 (A quatric complex polynomial with **zero-clusters**: a **highly ill-conditioned** problem)* $f(x) = x^4 - (8.04 + .22j)x^3 + (24.2227 + 1.3266j)x^2 - (32.410446 + 2.665828j)x + (16.25009862 + 1.78524984j)$ whose exact zeros are 2.01 + j.04, 2.01 +j.05, 2.01 + j.06, and 2.01 +j.07, where $j = \sqrt{-1}$. The *inputs* are

```
rmin=2;rmax=2.019;imin=0;imax=0.045;nmax=10;eps=.5*10^-8;
» fun='x^4-(8.04+.22*j)*x^3+(24.2227+1.3266*j)*x^2-
(32.410446+2.665828*j)*x+(16.25009862+1.78524984*j)';
func2(rmin,rmax,imin,imax,nmax,eps,fun)
```

The *outputs* are x = 2.0112 + 0.0470i, f = -6.5411e-009 -2.3059e-009i, absf = 6.9356e-009. When the program was rerun with the same inputs, the *outputs* became x = 2.0110 + 0.0519i, f = 3.5115e-009 -1.4762e-009i, absf = 3.8092e-009. The foregoing results seem reasonably good for the precision of 15 digits that MATLAB provides.

Examples 4 (A tenth degree real polynomial with large coefficients and distinct real zeros) $f(x) = x^{10} - 55x^9 + 1320x^8 - 18150x^7 + 157773x^6 - 902055x^5 + 3416930x^4 - 8409500x^3 + 12753576x^2 - 10628640x + 3628800$ whose zeros are 1, 2, 3, 4, 5, 6*, 7, 8, 9, and 10.* The *inputs* are

```
» func2(.1,1.5,0,0,10, .5*10^-8,'x^10-55*x^9+1320*x^8-18150*x^7+157773*x^6-
902055*x^5+3416930*x^4-8409500*x^3+12753576*x^2-10628640*x+3628800')
```

The *outputs* are x = 1.0000, f = -9.4986, absf = 9.4986. When the program was rerun with the same polynomial with changed *inputs*

```
» func2(.1,11,0,0,10, .5*10^-8,'x^10-55*x^9+1320*x^8-18150*x^7+157773*x^6-
902055*x^5+3416930*x^4-8409500*x^3+12753576*x^2-10628640*x+3628800')
```

we obtained the *outputs* x = 6.0000, f = -0.0570, absf = 0.0570. When we reran the program with the foregoing inputs for the second, third, fourth, and fifth times then the *outputs* became (a) x = 5.9977, f = -6.5615, absf = 6.5615, (b) x = 4.0030, f = 12.9012, absf = 12.9012, (c) x = 5.9988, f = -3.5642, absf = 3.5642, (d) x = 4.0015, f = 6.6179, absf = 6.6179, respectively.

6. CONCLUSIONS

Shrinking rectangle converges faster than 2-D bisection. When k goes to 10, the initial rectangle D (Fig. 1a) that encloses/contains a zero of the function f(x) will be shrunk to the rectangle whose area will be less than or equal to $D/2^k = D/2^{10} = 0.00097656D$. This shrinking is significantly rapid compared to the automatic bisection for complex zeros (Sen and Lord 1990, Wilf 1978).

The non-existence of a zero in the wrongly chosen initial rectangle can be detected. The SRA algorithm will come out indicating that the chosen rectangle D does not contain a zero if the choice is incorrect, i.e., if it really does not contain a zero.

Interpolation (including extrapolation) is carried out in the final highly shrunk rectangle only once It is possible to interpolate linearly in each of the k (= 10) rectangles. However, it is not done because the linear interpolation could be sufficiently inaccurate when the rectangle is large. Moreover, such repeated interpolations will not only increase the computation but also might result in excluding the actual zero in the rectangle-shrinking process.

The zero existing in the initial rectangle D will exist in the final shrunk rectangle. In our numerical experiment with numerous functions and with reasonably chosen initial rectangle D, the zero that was located in D always remained in the final shrunk rectangle. The SRA algorithm thus seems an efficient fail-proof complex zero finding method and it is *deterministic*.

The SRA algorithm is not worse than most algorithms for finding a zero in a zero-cluster. A function having zero-clusters (closely spaced zeros) is always an ill-conditioned problem with respect to finding a zero accurately in the cluster. Any method so far existing as well as any method that could be proposed in future would be only satisfactory to a varying extent for a specified precision. Our numerical experiment depicts that the SRA algorithm is reasonably good when dealing with zero-clusters.

Multiple zeros do not pose any problem to the SRA algorithm. Unlike the Newton method and its variations which need to compute derivatives of a function and in which an oscillation around a multiple zero (in a finite precision machine) sets in, the SRA algorithm has absolutely no such problem. It gives, like bisection methods, the multiple zero accurately as it does not depend on the computation of the derivatives of a function. For a polynomial having multiple zeros, *repeated deflations* will provide the order of multiplicity.

Use deflation or different rectangles to seeve out all the zeros. One way of seeving **all the zeros** of a polynomial with or without multiple zeros is to deflate the polynomial successively after computing a zero. The other way is to choose different appropriate intervals/rectangles each enclosing a zero and compute all the zeros. For a transcendental function that cannot be written as the product of a polynomial (with multiple zeros) and another transcendental function, deflations may not be useful.

The SRA algorithm has a sequential complexity O(n) and its parallel implementation is straight-forward. As we have seen in Sec. 3 that the SRA algorithm has a sequential computational complexity $O(2k \times n1 \times n)$ where the input size is $O(2n)$ for an nth degree complex polynomial. Observe that k (=10, say) and n_1 (=10 or 20, say) are independent of n. The parallel computational complexity, when we have n processors, is $O(k \times n_1)$ which is independent of the input size. For a fixed number of processors < n, this complexity will increase proportionately.

The SRA algorithm can be extended to obtain the global minimum of a multi-variable function. Instead of generating a pair of pseudorandom numbers for a complex zero of a function f(x), we have to generate an ordered set of pseudorandom numbers for this purpose and suitably modify this algorithm.

References

1. Banks, J.;Carson, J.S., II; Nelson, B.L., Discrete-event Simulation (2nd ed.), Prentice-Hall of India, New Delhi, 1998.
2. Krishnamurthy, E.V.; Sen, S.K., Numerical Algorithms: Computations in Science and Engineering, Affiliated East-West Press, New Delhi, 2001.
3. Mathews, J.H., Numerical Methods for Mathematics, Science, and Engineering, 2nd ed., Prentice-Hall of India, New Delhi, 1994.
4. Schilling, R.J.; Harries, S.L., Applied Numerical Methods for Engineers, using MATLAB and C, Thomson Asia Pvt. Ltd., Singapore, 2002.
5. Sen, S.K., Error and Computational Complexity in Engineering, in J.C. Misra (ed.),**Computational Mathematics, Modelling and Algorithms** as Chap. 5, Narosa Publishing House, New Delhi, 2002, 110-145.
6. Sen, S.K.; Lord, E.A., An automatic bisection to compute complex zeros of function, in S. Bandyopadhyay (ed.), **Information Technology: Key to Progress,** Tata-McGraw-Hill, New Delhi, 1990, 9-13.
7. Wilf, H., A global bisection method for computing the zeros of a polynomial in the complex plane, J. ACM, 415-420, 1978.