A Preconditioner Selection Heuristic for Efficient Iteration with Decomposition of Arithmetic Expressions for Nonlinear Algebraic Systems

R. B. Kearfott and Xiaofa Shi*

Abstract

We have recently considered decomposing a system of nonlinear equations by defining new variables corresponding to the intermediate results in the evaluation process. In that previous work, we applied both a derivative-free component solution process and an interval Gauss–Seidel method to the large, sparse system of equations so obtained.

An analysis of the component solution process indicates when a linearized Gauss–Seidel step is necessary, and how to make it more effective. In this paper, we will present preliminary results on an improved, efficient hybrid algorithm combining the component solution process with only an occasional Gauss–Seidel step on a single component.

1 Introduction, Background, and Motivation

The general goal of this paper is to find, with certainty, approximations to all solutions of the nonlinear system

$$F(X) = \begin{bmatrix} f_1(x_1, x_2, ..., x_n) \\ \vdots \\ f_n(x_1, x_2, ..., x_n) \end{bmatrix} = 0, \qquad (1)$$

where bounds $\underline{x}_i$ and $\overline{x}_i$ are known such that

$$\underline{x}_i \le x_i \le \overline{x}_i \quad \text{for } 1 \le i \le n.$$

We write $X = (x_1, x_2, ..., x_n)^T$, and we denote by $\mathbf{B}$ the box given by the above inequalities on the variables x_i.

*This work was supported in part by National Science Foundation grants CCR-9203730 and DMS 9216120.

A general approach to such problems is to transform the nonlinear system $F(X) = 0$ to the interval linear system

$$\mathbf{F}'(\mathbf{X}_k)(\tilde{\mathbf{X}}_k - X_k) \ni -F(X_k), \tag{2}$$

where $\mathbf{F}'(\mathbf{X}_k)$ is a suitable interval expansion of the Jacobi matrix over the box $\mathbf{X}_k$ ($\mathbf{X}_0 = \mathbf{B}$) and $X_k \in \mathbf{X}_k$ represents a predictor or initial guess point. A preconditioned interval Gauss–Seidel method may then be used to compute a new interval $\tilde{\mathbf{x}}_k$ for the k-th variable. The method in [10] uses this, in combination with a nonlinear solution process.

The method considered here is an improvement of the prototypical approach explained in [10]. The general approach makes use of both interval arithmetic and techniques of automatic differentiation. For brevity, we assume familiarity with techniques of interval arithmetic; see [1], [13] or [14] for introductions. We also assume that the reader has access to [10], where examples of the prototypical approach are worked.

Our method uses the intermediate quantities obtained through evaluation of arithmetic expressions. Our technology for obtaining and storing such expressions is essentially the backward mode of automatic differentiation, as explained in the review [5] or the proceedings [4]. However, our use of the intermediate quantities differs from that in straightforward (or "straightbackward") automatic differentiation: we store interval values of the intermediate quantities. We then recompute those intermediate quantities that depend on quantities that have changed due to relationships defined in the original equations, due to an interval Gauss–Seidel step, or due to bisection; we intersect the recomputed values with the original values, continuing the process until stationary. Thus, in contrast to usual automatic differentiation, we must view the defining relationships for the intermediate quantities as functional relationships, rather than as linearly ordered arithmetic operations.

This approach is very similar to that underlying the software described in [2], but our terminology differs. The main substantive difference is that, in addition to re-solving for the intermediate quantities and using bisection, we occasionally apply preconditioned interval Gauss–Seidel steps to the expanded system of equations obtained by treating each intermediate quantity as a variable. Such linear algebra (either on the expanded system or on the original system 2) is necessary when the system of equations is highly coupled[1]. Without such steps, even linear systems such as that in Example 1 in §5 below require a large amount of computation. This is obviated in the software of [2] with an extremely friendly environment in which users can interactively change intervals while exploring problems.

Our method is an improvement of that in [10], since far fewer expensive preconditioned interval Gauss–Seidel steps are used. In particular, in the algo-

[1]essentially, when the Jacobi matrix is not a row and column permutation of a diagonally dominant matrix

rithm of [10], an *entire sweep* of the interval Gauss–Seidel method was applied, whereas here an interval Gauss–Seidel step is applied only to a *selected coordinate* whose index is heuristically chosen. Additionally, our algorithm here uses *extended intervals* in cases of division by zero-containing intervals, etc., whereas the algorithm in [10], for programming simplicity, did not. Also, our selection of coordinate to bisect (when bisection is necessary) differs here from that in [10].

A brief introduction to the underlying ideas and terminology appears in §2. We explain our heuristic for determining the variable for the interval Gauss–Seidel step in §3. Our algorithm appears in §4, while numerical results appear in §5. Conclusions and future directions then appear.

2 Outline of Underlying Ideas

We write $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)^T$ for $\mathbf{X}_k$, $lb(\mathbf{x})$ or $\underline{\mathbf{x}}$ for the lower bound of the interval $\mathbf{x}$, $ub(\mathbf{x})$ or $\overline{\mathbf{x}}$ for the upper bound of the interval $\mathbf{x}$, and $\mathbf{f}'_{ij}$ for the interval in the i-th row and j-th column of $\mathbf{F}' = \mathbf{F}'(\mathbf{X})$. Similarly, we write $X = (x_1, x_2, ..., x_n)^T$ and $F = F(X) = (f_1, f_2, ..., f_n)^T$, so that (2) becomes

$$\mathbf{F}' \cdot (\tilde{\mathbf{X}} - X_k) \ni -F. \tag{3}$$

Suppose $Y_k = (y_{k1}, y_{k2}, ..., y_{kn})$ is the preconditioner for x_k. The preconditioned Gauss–Seidel method may then be stated as

Algorithm 2.1 (Preconditioned Gauss–Seidel method)

1. Compute $Y_k \cdot \mathbf{F}'(\tilde{\mathbf{X}} - X_k)$ and $-Y_k F$. Then compute

$$\tilde{\mathbf{x}}_k = x_k - \left[\sum_{i=1}^{n} y_{ki} f_i + \sum_{\substack{j=1 \\ j \neq k}}^{n} \left(\sum_{i=1}^{n} y_{ki} \mathbf{f}'_{ij} \right) (\mathbf{x}_j - x_j) \right] \Big/ \sum_{i=1}^{n} y_{ki} \mathbf{f}'_{ik} \tag{4}$$

2. If $\tilde{\mathbf{x}}_k \cap \mathbf{x}_k = \emptyset$, then return, indicating that there is no root of F in $\mathbf{X}$.

3. Replace $\mathbf{x}_k$ by $\mathbf{x}_k \cap \tilde{\mathbf{x}}_k$.

As explained in [8] and [11], the preconditioner Y_k is used to minimize the width of $\tilde{\mathbf{x}}_k$ represented by (4). Y_k can be computed by solving an L-P problem, as indicated in [8] and [11].

The Decomposition Process and the Code List

In this paper, a *code list* is used to represent the system of equations. The code list is essentially the list of elementary operations required to evaluate the

Operation 1:	x_p	$=$	$ax_q + bx_r$
Operation 2:	0	$=$	$ax_q + bx_r$
Operation 3:	0	$=$	$ax_q + b$
Operation 4:	x_p	$=$	$x_q x_r$
Operation 5:	x_p	$=$	x_q^2

Table 1: Some operation codes for a code list.

function. Such code lists are commonly used in automatic differentiation; see [5] or [4]. We explain the main idea of the decomposition process and the code list here.

Our code list is in the following format:

$$\underline{\text{Operation number} \quad P \quad Q \quad R \quad A \quad B},$$

where P, Q and R are variable indices and A and B are possible constants associated with the operation. Operation numbers are listed in Table 1. For example, we could decompose the equation

$$x^3 - 3x^2 + 2x = 0$$

as follows.

$$\begin{aligned} y_1 &= x \\ y_2 &= y_1^2 \\ y_3 &= y_1 y_2 \\ y_4 &= y_3 - 3y_2 \\ y_4 + 2y_1 &= 0 \end{aligned}$$

Examining Table 1, where we list operation numbers, we see the corresponding code list to be:

1	4				
5	2	1	0	0.	0.
4	3	1	2	0.	0.
1	4	3	2	1.	− 3.
2	0	4	1	1.	2.

The 1 and 4 in the first row are the numbers of equations in the original system and the expanded system, respectively.

Forward Substitution and "Solving Analytically"

In the above example, after decomposing the equation, we compute y_1, y_2, y_3 and y_4 from the expanded equations by forward substitution. For example, suppose the interval value of x is $\mathbf{x} = [-1, 0]$; then we compute the remaining intervals as follows:

$\mathbf{y}_1 = \mathbf{x} = [-1, 0]$
$\mathbf{y}_2 = \mathbf{y}_1^2 = [0, 1]$
$\mathbf{y}_3 = \mathbf{y}_1\mathbf{y}_2 = [-1, 0] \cdot [0, 1] = [-1, 0]$
$\mathbf{y}_4 = \mathbf{y}_3 - 3\mathbf{y}_2 = [-1, 0] - 3 \cdot [0, 1] = [-4, 0]$

From the last equation in the expanded system, we have
$y_4 = -2y_1$
Solving for $\mathbf{y}_4$ in this way, we get
$\mathbf{y}_4 = -2\mathbf{y}_1 = -2 \cdot [-1, 0] = [0, 2]$.
This is the meaning of **solving analytically**. Narinyani, Semenov, *et al.* use a similar technique in their "subdefinite" computations. Their work is embodied in the software package Unicalc, introduced in [2].

A FORTRAN-90 software system for automated generation and use of the code list, as well as a complete description of the code list itself, will be described in [12].

3 A Heuristic for Variable Selection

In our algorithm, we do not apply (4) to each variable k, but we choose a particular k likely to lead to progress. Ideally, such an index k would satisfy

$$\frac{w(\tilde{\mathbf{x}}_k)}{w(\mathbf{x}_k)} = \min_{1 \le l \le n} \frac{w(\tilde{\mathbf{x}}_l)}{w(\mathbf{x}_l)}, \tag{5}$$

where $w(\mathbf{x}_k)$ is the width of the interval $\mathbf{x}_k$.

Here, we consider the relative decrease $\frac{w(\tilde{\mathbf{x}}_k)}{w(\mathbf{x}_k)}$, rather than $w(\tilde{\mathbf{x}}_k)$ itself, to take account of scaling: Otherwise, if $w(\mathbf{x}_k)$ were already small, k could possibly be chosen even though a preconditioned Gauss-Seidel step would not decrease the width much. If $w(\mathbf{x}_k)$ were large, $w(\tilde{\mathbf{x}}_k)$ may still be large, and hence k could possibly not be chosen, even if a Gauss–Seidel step would cause a large decrease in width.

From (4), we have

$$w(\tilde{\mathbf{x}}_l) = w\left(\sum_{\substack{j=1\\ j\neq l}}^{n} \left(\sum_{i=1}^{n} y_{li}\mathbf{f}'_{ij}\right)(\mathbf{x}_j - x_j)\right) \tag{6}$$

provided $lb(\sum_{i=1}^{n} y_{li}\mathbf{f}'_{il}) = 1$. Thus,

$$
\begin{aligned}
w(\tilde{\mathbf{x}}_l) &\leq \sum_{\substack{j=1\\ j\neq l}}^{n} \left| \sum_{i=1}^{n} y_{li}\mathbf{f}'_{ij} \right| w(\mathbf{x}_j) \\
&\leq \sum_{\substack{j=1\\ j\neq l}}^{n} \sum_{i=1}^{n} \left|\mathbf{f}'_{ij}\right| w(\mathbf{x}_j)\,\|Y_l\|_1 \\
&\leq \|\mathbf{Y}\| \sum_{\substack{j=1\\ j\neq l}}^{n} \sum_{i=1}^{n} \left|\mathbf{f}'_{ij}\right| w(\mathbf{x}_j), \qquad (7)
\end{aligned}
$$

where $\left|\mathbf{f}'_{ij}\right| = \max\{\left|\underline{\mathbf{f}}'_{ij}\right|, \left|\bar{\mathbf{f}}'_{ij}\right|\}$, $\mathbf{Y} = \left(Y_1^T, Y_2^T, ..., Y_n^T\right)^T$ is the preconditioner for the whole system, and where we assume $x_j \in \mathbf{x}_j$ for every j.

Our heuristic involves replacing the relative decrease of (5) by the right member in (7) divided by $w(\tilde{x}_l)$. Doing so, we do not need to obtain Y_k to determine the estimates of the relationships among the $\frac{w(\tilde{\mathbf{x}}_l)}{w(\mathbf{x}_l)}$. We may thus choose k such that

$$
\frac{\sum_{\substack{j=1\\ j\neq k}}^{n} \sum_{i=1}^{n} \left|\mathbf{f}'_{ij}\right| w(\mathbf{x}_j)}{w(\mathbf{x}_k)} = \min_{1\leq l\leq n} \frac{\sum_{\substack{j=1\\ j\neq l}}^{n} \sum_{i=1}^{n} \left|\mathbf{f}'_{ij}\right| w(\mathbf{x}_j)}{w(\mathbf{x}_l)}. \qquad (8)
$$

Another factor we should consider here is, if the intervals $\mathbf{f}'_{ik}$ all contain 0 for $i = 1, 2, ..., n$, then the preconditioner Y_k does not exist. Thus, if we denote by $\mathcal{J}$ the set of all indices l such that $\mathbf{f}'_{il}$ do not contain 0 for at least one i, then we choose k such that

$$
\frac{\sum_{\substack{j=1\\ j\neq k}}^{n} \sum_{i=1}^{n} \left|\ \mathbf{f}'_{ij}\right| w(\mathbf{x}_j)}{w(\mathbf{x}_k)} = \min_{l\in\mathcal{J}} \frac{\sum_{\substack{j=1\\ j\neq l}}^{n} \sum_{i=1}^{n} \left|\mathbf{f}_{ij}\right| w(\mathbf{x}_j)}{w(\mathbf{x}_l)}. \qquad (9)
$$

4 Our Algorithms

In this section and the next, we present two algorithms that combine the component solution process with an occasional Gauss–Seidel step on a single component. We then use numerical examples to compare these algorithms with the algorithm discussed in [10].

As mentioned in the introduction, our algorithm first computes ranges exact to within roundout error on the inverses of elementary operations and functions to solve for each variable in each equation containing it. Then, when no variable can be changed by this process, we apply a preconditioned Gauss–Seidel method to the linear system (2) for a selected variable x_{i_0}. If the width of x_{i_0} is decreased, we compute all x_i's in each equation which contains x_{i_0}. If some additional x_i are changed, we repeat this process. A generalized bisection method will be used if none of the variables can be changed by either the variable solution process or the Gauss–Seidel step on the selected variable.

Algorithm 4.1

```
0. (Input the initial data.)
     a) Input $M$, the number of equations in the original system;
     b) Input $N$, the number of equations in the expanded system;
     c) Input the code list and the initial box $\mathbf{B}$.
        As in the example in §2, we place the $M$ equations corresponding to
        the original system at the end of the code list.
     d) Input $\epsilon$, a convergence tolerance.
     e) Use forward substitution to compute $\mathbf{x}_{M+i}$ for $1 \le i \le N-M$, and
        thus get the initial box $\mathbf{X}_0$ for the expanded system.
1. (Scan all the variables in the last $M$ equations.)
   For $i = 1, 2, ..., n$, if $x_i$ is contained in one of the last $M$ equations in the
   expanded system, store $i$ in $\mathcal{V}$.
2. (Solve for some variables analytically from the equations containing them.)
     Do for $l \in \mathcal{V}$ while $\mathcal{V} \neq \emptyset$.
        For each equation index $j$ such that $x_l$ occurs in the $j$-th equation, do
        a) Solve the $j$-th equation analytically for each variable $x_i$ contained
           in the $j$-th equation[2], obtaining new bounds $\tilde{\mathbf{x}}_i$.
        b) If $\tilde{\mathbf{x}}_i \cap \mathbf{x}_i = \emptyset$, return, indicating that there is no root of $F(X)$
           within the box given by $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)^T$.
        c) If $[w(\mathbf{x}_i) - w(\tilde{\mathbf{x}}_i \cap \mathbf{x}_i)]/w(\mathbf{x}_i)$ is bigger than a tolerance, say
           $\epsilon_0$, then
            (i) Store $i$ in the stack $\mathcal{V}$.
           (ii) Replace[3] $\mathbf{x}_i$ by $\tilde{\mathbf{x}}_i \cap \mathbf{x}_i$.
        End do.
```

[2]We do not solve for x_i in the j-th equation if we had just solved for some other variable in that equation, because no improvement in $\mathbf{x}_i$ could then be made.

[3]Since $\tilde{\mathbf{x}}_i$ may be an extended interval, this step may produce two intervals, and two corresponding boxes $\mathbf{X}_k$. In this case, one of the boxes is pushed onto a stack $\mathcal{S}$, and will be considered later in step 3.

```
      End do.
   End if.
3. (**Complete processing a sub-box, get a new sub-box**.)
   If the widths $w(\mathbf{x}_i)$ are each less than $\epsilon$ for $i = 1, 2, ..., M$, then
      a) Store $\mathbf{X}_k$ on a list $\mathcal{L}$ of root-containing boxes.
      b) If the stack $\mathcal{S}$ of sub-boxes yet to be considered (and produced from bisection or arithmetic operations) is empty, return indicating convergence.
      c) If $\mathcal{S}$ is nonempty, take a box from it, put the corresponding components in the list $\mathcal{V}$ of changed variables, then return to step 2.
4. (**Continue derivative-free iteration if possible**.)
   If $\mathcal{V}$ is nonempty, repeat steps 2 and 3.
5. (**Apply a preconditioned Gauss–Seidel step if necessary**.)
   If $\mathcal{V}$ is empty, then
      a) Compute the Jacobi matrix $\mathbf{F}'$ over $\mathbf{X}_k$, and select index $k = k_0$ as described in §3.
      b) Compute the preconditioner $Y_{k_0}$, apply the interval Gauss–Seidel method to the preconditioned system, and get a bound $\tilde{\mathbf{x}}_{k_0}$ on $\mathbf{x}_{k_0}$.
      c) If $\tilde{\mathbf{x}}_{k_0} \cap \mathbf{x}_{k_0} = \emptyset$, return, indicating that there is no solution of the system $F(X) = 0$ within the box $\mathbf{X}_k$.
      d) If $[w(\mathbf{x}_{k_0}) - w(\tilde{\mathbf{x}}_{k_0} \cap \mathbf{x}_{k_0})]/w(\mathbf{x}_{k_0}) \geq \epsilon_0$, put $k_0$ into $\mathcal{V}$, and then go to step 2.
   End if.
6. (**Do a bisection as a last resort.**)
   If $\mathcal{V}$ is empty, then
      a) Bisect $\mathbf{x}_{k_0}$.
      b) Store one of the boxes, with corresponding changed coordinate information on $\mathcal{S}$.
      c) Put $k_0$ into $\mathcal{V}$, make the other box the current box, and then go to step 2.
   End if.
7. **Return**.
```

Remark 1. In the algorithm discussed in [10], a preconditioner was computed for all variables, after the substitution/iteration process. In contrast, in Algorithm 4.1 above, we only compute the preconditioner for a single variable. We thus save a factor of N in this method, where N is proportional to the number of operations required to evaluate the functions. Perhaps, however, a few more boxes may be produced.
Remark 2. The reason we only scan the last M equations in step 1 is because forward substitution is used to obtain initial intervals in the other equations. Hence no additional information relating the interval sizes could be obtained from them. For example, suppose we compute $\mathbf{x}_3$ by $\mathbf{x}_3 = \mathbf{x}_1 + \mathbf{x}_2$. Then

$$\mathbf{x}_1 \subseteq \tilde{\mathbf{x}}_1 = \mathbf{x}_3 - \mathbf{x}_2 \text{ and } \mathbf{x}_2 \subseteq \tilde{\mathbf{x}}_2 = \mathbf{x}_3 - \mathbf{x}_1.$$

Thus, we will get nothing new by setting

$$\mathbf{x}_1 = \mathbf{x}_1 \cap \tilde{\mathbf{x}}_1 \text{ and } \mathbf{x}_2 = \mathbf{x}_2 \cap \tilde{\mathbf{x}}_2.$$

Remark 3. Sometimes the derivative-free process converges slowly. Thus, to make the overall iteration more efficient, we introduce a tolerance ϵ_0 in part c) of step 2 and in part d) of step 5, and check the approximate condition

$$[w(\mathbf{x}_i) - w(\tilde{\mathbf{x}}_i \cap \mathbf{x}_i)]/w(\mathbf{x}_i) \geq \epsilon_0, \tag{10}$$

instead of checking the exact condition $\mathbf{x}_i \neq \tilde{\mathbf{x}}_i \cap \mathbf{x}_i$.

Numerical tests show that, if an L-P preconditioner has previously been computed for a particular variable, the same preconditioner usually works well if we wish to solve for the same variable later in the computation. Thus, in step 5, if Y_{i_0} has been computed when step 5 was previously entered, we use the previous Y_{i_0}. We then only recompute Y_{i_0} if its previous values do not work well. In other words, we replace step 5 by the following

Step 5$'$. **Apply the preconditioned Gauss–Seidel method with the previous preconditioner Y_{i_0}.**

a) Compute the Jacobi matrix $\mathbf{F}'$ over $\mathbf{X}_k$ and select index i_0 as described in §3;

b) Apply the interval Gauss–Seidel method to the preconditioned linear system with the previous preconditioner Y_{i_0}, if Y_{i_0} is available.

   (i) If Y_{i_0} has been computed before, use it to precondition the linear system, and get a bound $\tilde{\mathbf{x}}_{i_0}$ on $\mathbf{x}_{i_0}$;

   (ii) If $\tilde{\mathbf{x}}_{i_0} \cap \mathbf{x}_{i_0} = \emptyset$, return, indicating that there is no root of $F(X)$ within $\mathbf{X}_k$;

   (iii) If $[w(\mathbf{x}_{i_0}) - w(\tilde{\mathbf{x}}_{i_0})]/w(\mathbf{x}_{i_0}) \geq \epsilon_0$, store i_0 into $\mathcal{V}$, go to step 2;

(iv) If $\mathcal{V}$ is empty, compute Y_{i_0} and store it in the i_0-th row of matrix $\mathbf{Y}$. Apply the preconditioned Gauss–Seidel method to get a bound $\tilde{\mathbf{x}}_{i_0}$ on x_{i_0}.

We will refer to the algorithm so obtained as **Algorithm 4.2**.
Remark 4. Of course, the matrix $\mathbf{Y}$ could require too much storage if the system took a large number of operations to evaluate. In such an instance, we may wish to only store several preconditioner rows (say, the last three distinct ones that have been computed). This should lead to a reasonable algorithm, for our experiments indicate that often only several variables are selected as step 5 a) is repeatedly executed.

5 Experimental Results

Our preliminary results will be based on the following illustrative examples.

Example 1:

$$F(X) = \begin{bmatrix} f_1(x_1,x_2,x_3) \\ f_2(x_1,x_2,x_3) \\ f_3(x_1,x_2,x_3) \end{bmatrix} = \begin{bmatrix} 9x_1 + 10x_2 + 11x_3 - 30 \\ 12x_1 + 13x_2 + 11x_3 - 36 \\ 13x_1 + 11x_2 + 14x_3 - 38 \end{bmatrix} = 0$$

with initial box $\mathbf{B}=\begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} [-20,20] \\ [-20,20] \end{pmatrix}$. The system has one solution within the initial box $\mathbf{B}$. This example is interesting because, though linear, the matrix is not a permutation of a diagonally dominant matrix, or an interval H-matrix. Thus, iteration without preconditioning cannot succeed, even with the decomposition technique of §2. The situation is analogous to that of the classical Gauss–Seidel method applied without preconditioner. This is why we include this seemingly trivial example: our algorithm attempts to avoid preconditioner computation, but uses a heuristic to determine when to apply a preconditioner.

Example 2:

$$F(X) = \begin{bmatrix} f_1(x_1,x_2) \\ f_2(x_1,x_2) \end{bmatrix} = \begin{bmatrix} x_1^3 + x_1^2x_2 + x_2^2 + 1 \\ x_1^3 - 3x_1^2x_2 + x_2^2 + 1 \end{bmatrix} = 0$$

with initial box $\mathbf{B} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} [-200,200] \\ [-200,200] \end{pmatrix}$. The system has one solution within the initial box $\mathbf{B}$. This simple example's interest lies in the fact that there is substantial interval dependency in the individual equations in the system and between the equations, and was introduced in [10] to illustrate the advantages of decomposition of arithmetic expressions.

Example 3:

$$F(X) = \begin{bmatrix} f_1(x_1,x_2) \\ f_2(x_1,x_2) \end{bmatrix} = \begin{bmatrix} 4x_1^3 - 3x_1 - x_2 \\ x_1^2 - x_2 \end{bmatrix} = 0$$

with initial box $\mathbf{B} = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} [-2,2] \\ [-2,2] \end{pmatrix}$. The system has three solutions in the initial box $\mathbf{B}$. This is Example 2 in [10], and is thus useful for comparison of the present algorithm with the process of [10].

Example 4:

$$\begin{aligned} f_i(X) &= x_i + x_{n+1}\left(\sum_{1\le j\le n} x_j - n - 1\right), \quad 1 \le i \le n-1, \text{ and} \\ f_n(X) &= (1 - x_{n+1})x_n + x_{n+1}\left(\prod_{1\le j\le n} x_j - 1\right), \end{aligned}$$

with $n = 5$ and initial box $[-2,2]^5$. This is Brown's almost linear function, used in [8] to illustrate when the inverse-midpoint preconditioner is inadequate, but a linear programming preconditioner will work, and reported as problem 4 in [6]. It is also useful to test the behavior of implementations as the dimension increases.

Example 5:

$$\begin{aligned} f_1 &= 5x_1^9 - 6x_1^5x_2^2 + x_1x_2^4 + 2x_1x_3 \\ f_2 &= -2x_1^6x_2 + 2x_1^2x_2^3 + 2x_2x_3 \\ f_3 &= x_1^2 + x_2^2 - 0.265625 \end{aligned}$$

with initial box $[-0.6, 0.6] \times [-0.6, 0.6] \times [-5, 5]$. This is problem 12 in [6]. Since it has 12 solutions in the box, it tests the ability of enclosure algorithms to separate solutions.

Numerical Results:

The goal of the coordinate selection heuristic of §3 is to reduce the total number of expensive preconditioner row computations, without affecting the number of steps required in the overall root isolation algorithm. In fact, the primary difference between Algorithm 4.1 and the overall procedure in [10] is the use of this heuristic. The overall procecure in [10] computed preconditioners for *all* rows for, roughly speaking, n times as many preconditioner computations as in Algorithm 4.1 or Algorithm 4.2. There are other differences. For example, to simplify implementation, we did not use extended (Kahan) interval arithmetic in the forward substitution process in [10], so that the only place two boxes were produced was after bisection.

We have implemented Algorithm 4.1 and Algorithm 4.2 in Fortran–SC (also known as ACRITH–XSC, [15]), to be run on an IBM 3090. For Algorihm 4.2, we report

EDIM the dimension of the expanded system,

NBOX the total number of boxes **X** processed,

NBIS the total number of coordinate bisections (step 6 of Algorithm 4.1 or Algorithm 4.2),

NPRE the total number of preconditioner rows computed in step 5 of Algorithm 4.1 or in step $5'$ of Algorithm 4.2,

NJACROW the total number of evaluations of a row of the expanded Jacobi matrix,

NRESID the total number of re-computations of a component of a residual for an equation in the expanded system.

NCOMPONENT the total number of re-computations of a coordinate in the substitution-iteration process in step 2, and

CPURAT the percentage of CPU time spent in computing preconditioners.

Except for the CPU percentage, these performance measures are insensitive to details of the implementation and the machine, and furthermore indicate the effectiveness of the heuristic proposed in §3. Our preconditioner computations use an interior point method devised and programmed by our co-worker Milind Dawande, and similar to that in [3]. This code has been chosen to take advantage of the extreme sparsity in the Jacobi matrix for the expanded system, but is still under development. In our algorithms, beginning with that of [8], we have observed large differences in execution time depending on the linear programming solver used in the preconditioner computations.

Comparisons with previous implementations are difficult. Our algorithms here were implemented using ACRITH-XSC, but many of our previous experiments used the portable arithmetic and framework of INTBIS. In the experiments in [10], we coded the arithmetic in the function-specific routines by hand, using subroutine calls for each operation. Though we use a code list in the present experiments and generic function routines to interpret this code list, we must create the code list by hand. To compare our results with those of [8], we either need to use polynomial systems in power form representation, as in INTBIS [9] or hand-code the operations as function calls. For these reasons, we have only compared the present computations to those of [10] on examples 2 and 4. We will soon remedy these shortcomings; see §6.

In all of the experiments, we took the preconditioner decision tolerance to be $\epsilon_0 = 0.2$ and the domain tolerance (minimal box width) to be $\epsilon = 10^{-6}$.

Results for Algorithm 4.2 appear in Table 2. Several conclusions are evident. First, solving for one component in terms of one or two others in an equation in the expanded system is very inexpensive compared to preconditioner computations. Thus, the preconditioner heuristic appears crucial to the algorithm's efficiency. Second, the portion of time spent in preconditioner computations is

Prob. no.	1	2	3	4	5
EDIM	9	9	5	17	20
NBOX	1	15	13	10	314
NBIS	0	0	2	5	101
NPRE	1	0	3	25	208
NJACROW	9	0	15	425	4160
NRESID	9	0	15	1734	4220
NCOMPONENT	817	373	1218	5337	28492
CPURAT	58%	53%	40%	79%	92%

Table 2: Results of Algorithm 4.2 for the five examples.

Prob. / Algorithm	4 / 4.1	4 / 4.2	5 / 4.1	5 / 4.2
EDIM	17	17	20	20
NBOX	10	10	315	314
NBIS	5	5	100	101
NPRE	70	25	214	208
NJACROW	1190	425	4280	4160
NRESID	1190	1734	4280	4220
NCOMPONENT	2536	5337	20085	19900
CPURAT	95%	79%	93%	92%
4.2/4.1	40%		98%	

Table 3: Algorithm 4.1 versus Algorithm 4.2 for Example 4 and Example 5.

still excessive, and the preconditioner computation can possibly be improved. For example, in the results in [8], use of an optimized linear programming solver reduced total CPU times by a factor of 10. Also, the LP problems become singular as the widths of components tend to zero; this may be bothersome for our particular interior point method. A solution would be to use an inverse midpoint preconditioner, perhaps on a subsystem, when a number of variable widths are small. Another factor is that the advantages of interior point methods usually do not appear on small problems such as those from these examples.

Table 3 gives results analogous to Table 2, but for Algorithm 4.1. We only give results for problems 4 and 5, since the two algorithms gave identical results for the first three problems, and we repeat the results for Algorithm 4.2, for comparison. The last row of the table, labelled 4.2/4.1 gives the ratios of CPU times. In Example 4, using the old preconditioners resulted in substantially less CPU time, despite double the number of solutions for a component. In contrast,

Prob. no. / Method	2 / Prev.	2 / This	3 / Prev.	3 / This
EDIM	7	9	5	5
NBOX	1	15	4	13
NBIS	0	0	3	2
NPRE	35	0	35	3
NJACROW	110	0	34	15
NRESID	220	0	68	15
NCOMPONENT	64	373	470	1227
CPURAT	57.2%	53%	13%	41%

Table 4: Comparisons with the overall procedure in [10].

using the old preconditioners for Example 5 had little effect, but did not hurt.

Table 4 compares Algorithm 4.2 to the overall scheme in §4 of [10], for Example 2 and Example 3. In the previous implementation in [10], we used a dense linear program solver that was specially designed to solve preconditioner computation problems; that solver is much faster on these particular problems. Also note the difference in dimensions between the results from [10] and the present results: we used slightly different code lists, due to slight differences in our set of implemented elementary operations. Also, note that *no* preconditioners were required in Example 2 in the present method. This is due to the use of extended arithmetic in the component solution process in the present method, but not in the previous one. It is nonetheless evident from the table that the preconditioner selection heuristic is effective at increasing the algorithmic efficiency, despite increased numbers of solutions for components.

6 Conclusions and Future Work

Preconditioners are, in general, required for efficient interval solution of nonlinear systems of equations, unless the system exhibits a diagonally dominant or similar structure. We will discuss such conditions in a separate paper. However, because of their computation expense, such preconditioners should be computed only when necessary. We have proposed a heuristic in this paper to decide when to compute the preconditioners.

Our preliminary experiments presented above indicate that this heuristic is effective, but the effectiveness of the entire algorithm has not been conclusively demonstrated. Our present experiments are constrained by the present software environment. We used a very restricted programming environment in the experiments in [10]; essentially, we hand-coded each function in terms of subroutine calls. Thus, rerunning that code on other problems is constrained by the time we can spend programming. The present experiments, though in a somewhat more flexible environment, suffer from similar problems. We are presently developing a portable Fortran 90 environment where code lists are automatically generated (through operator overloading), and where algorithmic building blocks, such as preconditioner computation and solving for a component, are modularized in a simple way. When this environment is completed and tested, our experimentation and algorithmic research will proceed more quickly. In particular, we will be able to do flexible experimentation, and will be able to flexibly try different algorithmic combinations, such as preconditioning either the original system or the expanded system. We will also be able to directly and carefully compare using the expanded system to the basic algorithm in [9] and [8]. Finally, we hope to compare different ways of parsing the original system into elementary operations, and to explore possible user-defined operations for increased efficiency.

We expect to find the expanded system to be of use. We also expect to find the preconditioner heuristic to be of use in many problems, both with the expanded system and the original system.

References

[1] Alefeld, Götz, and Herzberger, Jürgen, *Introduction to Interval Computations*, Academic Press, New York,1983.

[2] Babichev, A. B., Kadyrova, O. B., Kashevarova, T. P., and Semenov, A. L., *UniCalc - A Tool for Solving Tasks with Inexact and Incompletely Defined Data*, in INTERVAL-92, Proceedings, Vol. 2, pp. 7–7, 1992.

[3] Fourer, R. and Mehrotra, S., *Solving Symmetric Indefinite Systems in an Interior-Point Method for Linear Programming*, Technical Report No. 92-01, 1992.

[4] Griewank, A. and Corliss, G. F., ed., *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, SIAM, Philadelphia, 1991.

[5] Iri, M. and Kubota, K., *Methods of Fast Automatic Differentiation and Applications*, Technical Report No. RMI 87-02, University of Tokyo, Department of Mathematical Engineering and Instrumentation Physics, 1987.

[6] Kearfott, R. B., *Some Tests of Generalized Bisection*, ACM Trans. Math. Software **13** (3), pp. 197–220, 1987.

[7] Kearfott, R. B., *Interval Newton / Generalized Bisection When There are Singularities near Roots*, Annals of Operations Research **25**, pp. 181–196, 1990.

[8] Kearfott, R. B., *Preconditioners for the Interval Gauss–Seidel Method*, SIAM J. Numer. Anal. **27** (3), pp. 804–822, 1990.

[9] Kearfott, R. B., and Novoa, M., *INTBIS, A Portable Interval Newton/Bisection Package (Algorithm 681)*, ACM Trans. Math. Software **16** (2), pp. 152–157, 1990.

[10] Kearfott, R. B., *Decomposition of Arithmetic Expressions to Improve the Behavior of Interval Iteration for Nonlinear Systems*, Computing **47**, pp. 169–191, 1991.

[11] Kearfott, R. B., Hu, C. Y., Novoa, M. III, *A Review of Preconditioners for the Interval Gauss–Seidel Method*, Interval Computations **1** (1), pp. 59–85, 1991.

[12] Kearfott, R. B., *A Fortran-90 Interval Arithmetic, Nonlinear Equations, Nonlinear Optimization, and List-Handling Environment*, preprint, 1993.

[13] Moore, R. E., *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, 1979.

[14] Neumaier, A., *Interval Methods for Systems of Equations*, Cambridge University Press, Cambridge, England, 1990.

[15] Walter, W. V., *ACRITH-XSC a Fortran-like language for verified scientific computing.*, in Scientific Computing with Automatic Result Verification, Academic Press, New York, 1993.