Bergische Universität
Wuppertal

Selfverifying Solvers for Dense Systems of Linear Equations Realized in C-XSC

Carlos Hölbig and Walter Krämer

Preprint 2003/1

Wissenschaftliches Rechnen/
Softwaretechnologie

This work was supported in part by DLR international bureau (BMBF, Germany), FAPERGS (Brazil) and LabTeC/Dell/II-UFRGS Project (Brazil).

Impressum

Herausgeber:	Prof. Dr. W. Krämer, Dr. W. Hofschuster Wissenschaftliches Rechnen/Softwaretechnologie Fachbereich 7 (Mathematik) Bergische Universität Wuppertal Gaußstr. 20 D-42097 Wuppertal

Internet-Zugriff

Die Berichte sind in elektronischer Form erhältlich über die World Wide Web Seiten

`http://www.math.uni-wuppertal.de/wrswt/literatur.html`

Autoren-Kontaktadresse

Walter Krämer
Bergische Universität Wuppertal
Gaußstr. 20
D-42097 Wuppertal

E-mail: `kraemer@math.uni-wuppertal.de`

Carlos Hölbig
PPGC at UFRGS and Universidade de Passo Fundo
Av. Bento Gonçalves 9500 - Campus do Vale - Bloco IV - Bairro Agronomia
CEP 91501-970 Porto Alegre - Brazil

E-mail: `holbig@inf.ufrgs.br`

Selfverifying Solvers for Dense Systems of Linear Equations Realized in C-XSC

Carlos Hölbig, Walter Krämer
Department of Mathematics, University of Wuppertal
holbig@inf.ufrgs.br, kraemer@math.uni-wuppertal.de

February 2003

Abstract

In this note selfverifying solvers for systems of linear equations $Ax = b$ with dense square and non square $n \times m$ coefficient matrices A are described. In the over-determined case $(m > n)$ a vector $x \in I\!\!R^n$ is sought whose residuum $b - Ax$ has minimal Euclidian norm whereas in the under-determined case $(n < m)$ a solution $x \in I\!\!R^n$ is sought which has minimal norm. C-XSC implementations of all algorithms are given. The source code of the routines is freely available. The presented description of the algorithms is taken from [14], Chapter 2 (see also [17]). Only some minor modifications have been introduced in view of the realization of the algorithms using the C++ class library C-XSC [7].

Acknowledgements

We would like to thank the students Bernardo Frederes Krämer Alcalde and Paulo Sérgio Morandi Júnior from the Mathematic of Computation and High Performance Computing Group at UFRGS (supervisor: Prof. Dr. T. Diverio) for their help in implementing and testing the C-XSC routines.

This work was supported in part by DLR international bureau (BMBF, Germany), FAPERGS (Brazil) and LabTeC/Dell/II-UFRGS Project (Brazil).

1 Introduction

One of the most frequent tasks in numerical analysis is the solution of systems of linear equations

$$Ax = b \tag{1}$$

with an $n \times n$ matrix A and a right hand side $b \in I\!\!R^n$. Many different numerical algorithms contain this task as a subproblem.

As a generalization to this problem with a square matrix A, we often encounter *over-* or *under-determined* systems, i.e. systems where A is not a square, but rather an $m \times n$ matrix with $m > n$ in the over- and $m < n$ in the under-determined case. In the over-determined case a vector $x \in I\!\!R^n$ is sought whose residuum $b - Ax$ has minimal Euclidian norm. In the under-determined case, a solution $x \in I\!\!R^n$ of (1) is sought which has minimal norm.

The inversion of the matrix A is also a problem of this type. Here, the right hand side of (1) has to be replaced by the $n \times n$ identity matrix I and the solution X, which is a matrix now, is the inverse A^{-1} of A. In the over- or under-determined case this matrix X is the Moore-Penrose pseudo inverse A^+ of A (if A has full rank).

There are numerous methods and algorithms computing approximations to the solution x in floating-point arithmetic. However, usually it is not clear how good these approximations are, or if there exists a unique solution at all. In general, it is not possible to answer these questions with mathematical rigour if only floating-point approximations are used. These problems become especially difficult if the matrix A is ill conditioned.

We present some algorithms which answer the questions about existence and accuracy automatically once their execution is completed successfully. Even very ill conditioned problems can be solved with these algorithms. Most of the algorithms presented here can be found in [24]. All algorithms work for all four basic numerical C-XSC data types: *real, interval, complex*, and *complex interval*.

We assume the coefficient matrix A in (1) to be dense, i.e. in a C-XSC program, we use a square matrix of type *rmatrix, imatrix, cmatrix*, or *cimatrix* to store A and we do not consider any special structure of the elements of A. In an additional paper we treat systems with banded coefficient matrices [8].

Our goal is to write a C-XSC program that *verifies the existence* of a solution and *computes an enclosure* for this solution for each of the following types of problems:

(s) compute an enclosure for the solution of system (1) for a *square* $n \times n$ matrix A.

(o) compute an enclosure for the solution of system (1) in the *over-determined* case, i.e. for an $m \times n$ matrix A where $m > n$.

(u) compute an enclosure for the solution of system (1) in the *under-determined* case, i.e. for an $m \times n$ matrix A where $m < n$.

(S) compute an enclosure of the *inverse* A^{-1} of A.

(O) compute an enclosure of the *pseudo inverse* A^+ of A in the *over-determined* case, i.e. for an $m \times n$ matrix A where $m > n$.

(U) compute an enclosure of the *pseudo inverse* A^+ of A in the *under-determined* case, i.e. for an $m \times n$ matrix A where $m < n$.

We also want these six problems to be solved for all four basic numerical C-XSC data types: *real, interval, complex*, and *complex interval*. In the following Section 2, we will briefly outline the solution methods. The corresponding algorithmic description can be found in Section 3. The C-XSC program code for real input data will be presented in Section 5. (C++ templates and the C++ exception handling [9] are not used in the actual implementation.)

2 Theoretical Background

In this section, we give a brief summary of the theory of the enclosure methods for our six problems. A more detailed presentation can be found in [24].

Starting with problem (s), we first assume that we have an approximate solution $\tilde{x}$ and an approximate inverse R of the square matrix A. Rather than computing an enclosure for the solution directly, we will try to enclose the error of the approximate solution, yielding a much higher accuracy. The error $y = x - \tilde{x}$ of the true solution x satisfies the equation

$$Ay = b - A\tilde{x}, \tag{2}$$

which can be multiplied by R and rewritten in the form

$$y = R(b - A\tilde{x}) + (I - RA)y. \tag{3}$$

Let $f(y) := R(b - A\tilde{x}) + (I - RA)y$. Then Equation (3) has the form

$$y = f(y) \tag{4}$$

of a fixed point equation for the error y. If R is a sufficiently good approximation of A^{-1}, then an iteration based on (4) can be expected to converge since then $I - RA$ will have a small spectral radius.

Therefore, we derive the following iteration from (4), where we use interval arithmetic and intervals $[y]_k$ for y:

$$[y]_{k+1} = R\Diamond(b - A\tilde{x}) + \Diamond(I - RA)[y]_k \tag{5}$$

or

$$[y]_{k+1} = F([y]_k), \tag{6}$$

where F is the interval extension of f.

Here $\Diamond$ means that the succeeding operations have to be executed exactly and the result is rounded to an enclosing interval (-vector or -matrix). Since in the computation of the defect $b - A\tilde{x}$ and of the iteration matrix $I - RA$, serious cancellations of leading digits must be expected. Hence, these should be computed using the exact scalar product. Each component is computed exactly and then rounded to a machine interval. For this purpose, the scalar product expressions of XSC-languages are used extensively in the implementations. With $z := R\Diamond(b - A\tilde{x})$ and $C := \Diamond(I - RA)$, Equation (5) can be written as:

$$[y]_{k+1} = z + C[y]_k\ .$$

In order to prove the existence of a solution of (2) and thus of (1), we use Brouwer's fixed point theorem, which applies as soon as we have at some iteration index $k + 1$ an inclusion of the form

$$[y]_{k+1} = F([y]_k) \subset [y]_k^\circ, \tag{7}$$

where $[y]_k^\circ$ means the interior of $[y]_k$. If this *inclusion test* (7) holds, then the iteration function f maps $[y]_k$ into itself. From Brouwer's fixed point theorem, it follows that f has a fixed point y^* which is contained in $[y]_k$ and in $[y]_{k+1}$. The requirement that $[y]_k$ is mapped into its interior ensures that this fixed point is also unique, i.e. (2) has an unique solution y^*, and thus (1) also has a unique solution $x^* = \tilde{x} + y^*$.

Remark: According to [24], if the inclusion test (7) is satisfied, the spectral radius of C (and even that of $|C|$, which is the matrix of absolute values of C) is less then 1, ensuring the convergence of the iteration (also in the interval case). Furthermore, this implies also the nonsingularity of R and of A and thus the uniqueness of the fixed point.

A problem which still remains is that we do not know whether we can succeed in achieving condition (7). For example, in the trivial case of $n = 1$ with $a_{11} = 0.1, b = 0$, and $\tilde{x} = 1$, the iteration converges to the unique solution $x^* = 0$. However, the convergence is monotonically decreasing, and (7) is never satisfied.

To force (7), we therefore introduce the concept of *ϵ-inflation*, which blows up the intervals somewhat, in order to "catch" a nearby fixed point. For a real interval $[w]$, we denote ϵ-inflation with the functional notation $\mathrm{blow}([w], \epsilon)$, where

$$\mathrm{blow}([w], \epsilon) := \begin{cases} (1+\epsilon)[w] - \epsilon[w] & \text{, if } \mathrm{diam}([w]) > 0, \\ [\mathrm{pred}(w), \mathrm{succ}(w)] & \text{, if } \mathrm{diam}([w]) = 0, \end{cases} \tag{8}$$

respectively, where $\mathrm{diam}([w])$ is the diameter of the interval ($\mathrm{diam}([w]) = \overline{w} - \underline{w}$) and in the case $\mathrm{diam}([w]) = 0$, $[w] = w$ is assumed to be a floating point number, and $\mathrm{pred}(w)$ and $\mathrm{succ}(w)$ are its predecessor and successor in the floating point screen. Similarly, we use the ϵ-inflation $\mathrm{blow}(\cdot)$ also for interval vectors and matrices, where it is applied componentwise.

It can be shown, e.g. [26] that (7) will always be satisfied after a finite number of iteration steps, whenever the absolute value $|C|$ of iteration matrix C has spectral radius less than 1.

We have not yet said how we compute our approximate solution $\tilde{x}$ and the approximate inverse R. In principle, there is no special requirement about these quantities, we could even just guess them. However, the results of the enclosure algorithm will of course depend on the quality of the approximations.

We now sketch the method we use in our C-XSC program for the computation of R and $\tilde{x}$.

To begin with, we do not use a special algorithm for the computation of the approximate solution, since we must compute an approximate inverse $R \approx A^{-1}$ anyway. Thus, we also have immediately an approximate solution $\tilde{x} := Rb$. However, the quality of this approximation is often not sufficient for the interval iteration to converge fast. Therefore, we first improve this approximation by use of an iterated defect correction:

$$\tilde{x}_{k+1} = \tilde{x}_k + R(b - A\tilde{x}_k) \tag{9}$$

using floating point arithmetic only and the exact scalar product for the defect $b - A\tilde{x}_k$. Improving the approximation in floating point arithmetic first is much cheaper than computing many interval iterations later.

For the computation of the approximate inverse R, we use the well-known Gauss-Jordan algorithm with column pivoting [28]. We do not repeat the algorithm here since it is well-known and it is listed as C-XSC code in Section 5 anyway. However, we explain a minor change in the algorithm which was included in order to make the algorithm more robust in the case of almost singular matrices. This modification concerns the elimination steps of the Gauss-Jordan algorithm which may produce exact zeros on the computer because of rounding errors. Later in the computation, when a pivot element is to be searched for, it could be the case that no pivot can be found since all candidates are exact zeros on the machine, thus causing a breakdown of the Gauss-Jordan algorithm, even when the true values of some pivot candidates are not zero. Therefore, we replace any exact zero value which was produced in an elimination step by a nonzero value in the order of magnitude of the roundoff error, i.e. we replace the result of $a - a = 0$ by ϵa where ϵ is the relative machine precision. This modification forces the Gauss-Jordan algorithm to execute completely for any nonsingular matrix (and also many singular matrices). Now the failure to find a pivot element also means that the zeros have been already in the input matrix, which therefore is surely singular.

In very ill conditioned cases, the quality of R computed this way will not be sufficient, i.e. the spectral radius of $|C|$ will not be less then 1, and the inclusion test will never be satisfied. In this case, we stop the interval iterations after a specified number of iterations (10, say) and recompute the approximate inverse in higher precision. The method we use is due to Rump, [23], so we call it Rump's device.

Assume we have an approximate inverse R (as we do from the Gauss-Jordan algorithm), then even if A is very ill conditioned, the matrix RA is usually very much better conditioned than A is. Now the simple relation

$$A^{-1} = (RA)^{-1}R \tag{10}$$

suggests that we compute another approximate inverse S of RA and take the product of S and R as a better approximation of A^{-1}. Since we can compute this product exactly with the aid of the exact scalar product in the long accumulator, it is also easy to approximate it by the sum of two matrices R_1 and R_2.

Summarizing, we can compute an approximate inverse $R_1 + R_2$ of double length (stored in two real floating point matrices $R_1 and R_2$) by the following steps:

1. compute an approximate inverse R of A with Gauss-Jordan, modified to replace zero valueslatex lsscxsc.

2. compute RA.

3. compute an approximate inverse S of RA with Gauss-Jordan.

4. compute SR in the long accumulator, and store it as sum of two floating point matrices R_1 and R_2.

Now that we have an approximate inverse $R := R_1 + R_2$ of double length, we can execute the algorithm described above with this R by using the exact scalar product whenever $R = R_1 + R_2$ is used in the algorithm.

The overall strategy of our method finally is:

- Compute an approximate inverse R of single length and execute the enclosure algorithm. If this fails then

- improve the approximate inverse by Rump's device and execute the enclosure algorithm with the double length approximate inverse $R = R_1 + R_2$.

Now that we have treated the case of one linear equation with one right hand side b, we turn to the inversion of a matrix A. The inverse $X = A^{-1}$ is the solution of the matrix-equation $AX = I$ with the $n \times n$ identity matrix on the right hand side. This equation can of course be solved column wise, thus obtaining the columns of $X = A^{-1}$ successively.

For the n individual matrix-vector equations which have to be solved, we use the algorithm just presented. In order to keep the computational overhead small, we use some simple observations to avoid the recomputation of intermediate quantities in the algorithm: Since all n equations have the same coefficient matrix A, it would be a big waste of computation time to recompute the approximate inverse R and the iteration matrix $C = \diamond(I - RA)$ for each of the n matrix-vector equations. Rather, we will implement the algorithm for the square case (s) in such a way that the approximate inverse R (or $R_1 and R_2$ if a double length inverse was needed) as well as its residual matrix $C = \diamond(I - RA)$ (or $C = \diamond(I - R_1A - R_2A)$) are saved after the first computation and are reused in the solution of the following equations.

In the C-XSC code, the function that computes solutions of the square matrix-vector equation (i.e. the local function `LSS`) must communicate with its calling functions for matrix-equations which quantities of the algorithm have already been computed (R, R_1, R_2, $\diamond(I - RA)$, and $\diamond(I - R_1A - R_2A)$) and can be reused in a future call. For this communication a flag (named `FLAGS`) is used in the C-XSC program code in Section 5 to avoid these unnecessary recomputations.

Next, we consider the cases of over- and of under-determined systems. (For more details on the theory see, e.g. [28].) In both cases, we assume the $m \times n$-matrix A to have full rank, i.e. in the over-determined case (case (o), $m \geq n$), A has rank n, and in the under-determined case (case (u), $m \leq n$), A has rank m. Here x is an n-vector, and b is an m-vector.

In case(o) the system (1) has no solution in general. Therefore, we are rather interested in a vector x which minimizes the Euclidian norm of the residual vector $r = b - Ax$, or, equivalently, the square of this norm. That is, we seek the solution of the linear least squares problem:

$$\|Ax - b\|_2^2 = min \ .$$

It is well known (e.g. [28], [27]) that such an x is uniquely determined (if A has full rank), and that it is the solution of the system of *normal equations*

$$A^H Ax = A^H b, \tag{11}$$

where A^H is the Hermitian matrix of A, i.e. the transposed matrix in the real case.

We could now proceed to compute $A^H A$ and $A^H b$ and to solve the resulting square $n \times n$ system using the previously presented method. However, as is well known, $A^H A$ usually has a very bad condition. Moreover, on the computer $A^H A$ can only be obtained with roundoff errors or as an interval matrix, which makes the solution of this system difficult.

Instead, we follow the suggestion of [24] and rewrite (11) as a larger square $(n+m) \times (n+m)$-system, which can be solved by the previous method to very high accuracy (but also with much higher computational effort if $m >> n$). Introducing a new m-vector $y = Ax - b$, we immediately obtain $A^H y = 0$ from (11). We write these two equations in block form

$$\begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b \\ 0 \end{pmatrix}, \tag{12}$$

a square $(n+m) \times (n+m)$ system (I is the $m \times m$ identity matrix here). This system has much better condition than the original normal equations.

Now it is straightforward to solve (12) by the method for square systems. The x-part of the resulting enclosure then is an enclosure for the solution x of the normal equations (11).

In a very similar way, we also proceed in the case of under-determined systems. Here, the system (1) usually has infinitely many solutions, and we are interested in the vector y among these solutions x which has minimal Euclidian norm. This vector can be determined as $y = A^H x$, where x is the solution of $AA^H x = b$. Again we write these two equations in block form

$$\begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 \\ b \end{pmatrix}, \tag{13}$$

again a square $(n+m) \times (n+m)$ system (here I is the $n \times n$ identity matrix). Again, we solve this system (13) by use of the method for square systems. The y-part of the resulting enclosure then yields an enclosure for the solution of our problem.

Finally, we treat the two cases (O) and (U), i.e. the computation of the Moore-Penrose pseudo inverse A^+ of A in the over- and under-determined cases (here we also assume A to have full rank).

In case (O), the $n \times m$-matrix A^+ ($m \geq n$) is given by (see e.g. [28], [27]).

$$A^+ = (A^H A)^{-1} A^H \tag{14}$$

or, equivalently, by the solution $Y = A^+$ of the matrix equation $A^H A Y = A^H$. Introducing the $m \times m$-matrix $X = AY - I$, we see that $A^H X = 0$ and, as previously, we can write these two equations in block form:

$$\begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \cdot \begin{pmatrix} Y \\ X \end{pmatrix} = \begin{pmatrix} I \\ 0 \end{pmatrix}, \tag{15}$$

where both identity matrices are $m \times m$.

Solving this equation column-wise with the algorithm for square matrices we get an enclosure for the pseudo inverse A^+ by extracting the Y-block from the computed enclosure.

Since the pseudo inverse has the property $(A^+)^H = (A^H)^+$, it follows from (14) that in case (U) ($m \leq n$), we have

$$A^+ = A^H (AA^H)^{-1}$$

or, equivalently, A^+ is the solution $Y = A^+$ of $YAA^H = A^H$. For the $n \times n$-matrix $X = YA - I$, we see that $XA^H = 0$. Taking the Hermitian of the two latter equations, we can again write these in a block form:

$$\begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \cdot \begin{pmatrix} Y^H \\ X^H \end{pmatrix} = \begin{pmatrix} I \\ 0 \end{pmatrix}, \tag{16}$$

with the identity matrices being $n \times n$.

Solving this equation column-wise with the algorithm for square matrices, we get an enclosure for the pseudo inverse A^+ by extracting the Y^H-block from the computed enclosure and taking its Hermitian matrix as the result.

Remark: We should stress that the algorithms which were presented here for the cases (o), (u), (O), and (U) generally deliver very narrow bounds for the solution, but that they are computationally very expensive, at least in the case where $|m - n|$ is large since the solution is always computed by use of a system of dimension $n + m$. There are other methods which can enclose the solution more efficiently (though perhaps less accurately), see Section 7. The reasons for the presentation of our methods are their high accuracy and that they are straightforward and can be implemented in C-XSC in some 10-20 lines of code.

According to a remark in [24], the number of operations can be reduced substantially in our algorithms for the cases (o), (u), (O), and (U) if we carefully observe the special structure of the block coefficient matrices B and compute only those elements of the approximate inverse R and its residual matrix $I - RB$ which are actually needed.

3 Algorithms

Following the theoretical discussion in Section 2, we can outline the following algorithms in a pseudo Pascal notation.

Algorithm 3.1: Solution of problem (s): {procedure}
{Compute an enclosure for the solution of the square linear system $Ax = b$}

```
Part I  (approximate inverse of single length)
   I.1  compute an approximate inverse $R$ of $A$ (e.g. using a Gauss-Jordan algorithm)
   I.2  compute an approximation $\tilde{x} := Rb$ of $x$
        improve $\tilde{x}$ by an iterated defect correction :
        repeat
            $\tilde{x} := \tilde{x} + R(b - A\tilde{x})$
        until $\tilde{x}$ accurate enough or maximum iteration count exceeded
   I.3  compute enclosures for the residuum:
        $Z := R\Diamond(b - A\tilde{x})$
        and for the iteration matrix:
        $C := \Diamond(I - RA)$
   I.4  interval iteration
        $Y := Z$
        repeat
            $Y_A := \mathrm{blow}(Y, \epsilon)$ {$\epsilon$ - inflation }
            $Y := Z + C \cdot Y_A$
        until $Y \subset \mathrm{int}(Y_A)$ or maximum iteration count exceeded
   I.5  if $Y \subset \mathrm{int}(Y_A)$ then
            a unique solution $x$ exists and $x \in \tilde{x} + Y$
        else
            if in Part I then
                Part I failed, goto Part II with $R_1 := R$
            else
                algorithm failed, the matrix $A$ is ill conditioned or singular
```

```
Part II  (approximate inverse of double length)
  II.1   (compute an approximate inverse $R := R_1 + R_2$ of $A$:)
         $S := R_1 \cdot A$
         compute an approximate inverse $S_1$ for $S$ (e.g. as in I.1)
         $S := S_1 \cdot R_1$
         $R_2 := S_1 \cdot R_1 - S$
         $R_1 := S$
  II.2   goto step I.2 of Part I.
```

Algorithm 3.2: Solution of problem (o): {procedure}
{Compute an enclosure for the solution of the over-determined linear system $Ax = b$}

1. $A_{big} := \begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \in I\!R^{(n+m)\times(n+m)}$

   $B_{big} := \begin{pmatrix} b \\ 0 \end{pmatrix} \in I\!R^{n+m}$

   $Y_{big} := \begin{pmatrix} x \\ y \end{pmatrix} \in I\!R^{n+m}$
2. solve $A_{big}Y_{big} = B_{big}$ using Algorithm 3.1
3. vector x from the vector Y_{big} is the desired enclosure.

Algorithm 3.3: Solution of problem (u): {procedure}
{Compute an enclosure for the solution of the under-determined linear system $Ax = b$}

1. $A_{big} := \begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \in I\!R^{(n+m)\times(n+m)}$

   $B_{big} := \begin{pmatrix} 0 \\ b \end{pmatrix} \in I\!R^{n+m}$

   $Y_{big} := \begin{pmatrix} x \\ y \end{pmatrix} \in I\!R^{n+m}$
2. solve $A_{big}Y_{big} = B_{big}$ using Algorithm 3.1
3. vector x from the vector Y_{big} is the desired enclosure.

Algorithm 3.4: Solution of problem (S): {procedure}
{Compute an enclosure for the inverse of the square matrix A}

```
1. (solve $AX = I$ column-wise)
   for $i := 1$ to $n$ do
   begin
       $b_i := e_i$ (= $i$-th unit vector)
       solve $Ax_i = b_i$ using Algorithm 3.1
   end
2. $X = (x_1, \ldots, x_n)$ is the desired enclosure.
```

Algorithm 3.5: Solution of problem (O): {procedure}
{Compute an enclosure for the pseudo inverse of the $m \times n$ matrix A, $m > n$}

```
1. $A_{big} := \begin{pmatrix} A & -I \\ 0 & A^H \end{pmatrix} \in I\!R^{(n+m)\times(n+m)}$, $I \in I\!R^{m\times m}$
   $B_{big} := \begin{pmatrix} I \\ 0 \end{pmatrix} \in I\!R^{n+m\times m}$, $I \in I\!R^{m\times m}, 0 \in I\!R^{n\times m}$
   $Y_{big} := \begin{pmatrix} X \\ Y \end{pmatrix} \in I\!R^{n+m\times m}$, $X \in I\!R^{m\times m}, Y \in I\!R^{n\times m}$
2. solve $A_{big}Y_{big} = B_{big}$ using Algorithm 3.4
3. block $X$ from the matrix $Y_{big}$ is the desired enclosure.
```

Algorithm 3.6: Solution of problem (U): {procedure}
{Compute an enclosure for the pseudo inverse of the $m \times n$ matrix A, $m < n$}

```
1. $A_{big} := \begin{pmatrix} A^H & -I \\ 0 & A \end{pmatrix} \in I\!R^{(n+m)\times(n+m)}$, $I \in I\!R^{m\times m}$
   $B_{big} := \begin{pmatrix} I \\ 0 \end{pmatrix} \in I\!R^{n+m\times n}$, $I \in I\!R^{n\times n}, 0 \in I\!R^{m\times n}$
   $Y_{big} := \begin{pmatrix} Y^H \\ X^H \end{pmatrix} \in I\!R^{n+m\times n}$, $X \in I\!R^{n\times n}, Y \in I\!R^{n\times m}$
2. solve $A_{big}Y_{big} = B_{big}$ using Algorithm 3.4
3. block $X$ from the matrix $Y_{big}$ is the desired enclosure.
```

4 Applicability of the Algorithms

The algorithms presented in Section 3 can be applied to any system of linear equations which can be stored in the floating point system on the computer. They will, in general, succeed in finding and enclosing a solution or, if they do not succeed, will tell the user so. In the latter case, the user will know that the problem is very ill conditioned or that the matrix A is singular.

In the following implementation in C-XSC, there is the chance that if the input data contains large numbers or if the inverse of A or the solution itself contain large numbers, an overflow may occur, in which case the algorithms may crash. In practical applications, this has never been observed, however. This could also be avoided by including the floating point exception handling which C-XSC offers for IEEE floating point arithmetic.

If the problem contains interval data in the matrix A or in the right hand side b, then the algorithms can be applied. However, the exact solution set is usually overestimated by the enclosure which is computed by the algorithm. Usually this overestimation becomes larger as the diameter of the input interval increases. For certain classes of matrices for which the optimal interval hull of the solution set can be computed (e.g. interval M-matrices), this algorithm will usually overestimate the solution set. This overestimation can be estimated by use of certain quantities appearing in the algorithm. We will not discuss this here, however. For details see e.g. [25] and [6].

5 C-XSC Program Code

The C-XSC program which is given here is written for the case of real input data, i.e. A is of type *rmatrix* and b is of type *rvector*. However, because of the clear structure of C-XSC, it is easy to transform this program code to make use of the other data types *interval, complex*, and *complex interval*. The changes in the program are mainly changes of the data type of certain variables and functions. We will indicate the necessary changes at the end of this section. (In a future implementation the data type should be a template parameter.)

The following module `lss_aprx` contains the function `MINV` which computes an approximate inverse of the input matrix `A` of type *rmatrix* using the Gauss-Jordan algorithm (see e.g. [28]). The parameter `A` is for input and output. The second parameter `err` indicates whether an approximate inverse could be computed (`error=0`) or if the computation failed (`error=1`).

There is, however, one trick included in this algorithm: in order to be able to complete the algorithm even for ill conditioned matrices, we avoid computing matrix elements which are exactly zero in the elimination step. For ill conditioned matrices, these elements could appear later as pivot elements, thus forcing the computations to break down. If the elimination step would yield an element `a[l][k]` which is rounded to an exact zero, we rather replace it by `eps*a[l][k]` with the old value of `a[l][k]` and an `eps` close to the relative machine precision. If we then still find a zero pivot element, then it must have been in the input matrix. Hence, the input matrix was actually singular. (This is, however, only almost true, since we did not consider the possibility of underflow in the multiplication `eps*a[l][k]`).

The reason why the array index computations look somewhat complicated is that we do not assume that the matrix `A` has the same index ranges in both dimensions, it only has to be square. E.g. if `A` is 10×10 the first index might run from 0 to 9, whereas the second index could run from -5 to 4.

```
//----------------------------------------------------------------------------
// File: lss_aprx (header)
// Purpose: Compute an approximate inverse of the square matrix A
// Global functions:
//   MINV(): Computes the approximate inverse of a square matrix A using
//           the Gauss-Jordan algorithm.
//----------------------------------------------------------------------------

#ifndef __LSS_APRX_HPP
#define __LSS_APRX_HPP

#include <l_rmatrix.hpp>   // Long Real matrix/vector arithmetic
#include <intvecto.hpp>    // Integer vector type

using namespace cxsc;
using namespace std;

extern void MINV ( rmatrix&, int& );

#endif

//----------------------------------------------------------------------------
// File: lss_aprx (implementation)
// Purpose: Compute an approximate inverse of the square matrix A
// Global functions:
//   MINV(): Computes the approximate inverse of a square matrix A
//           using the Gauss-Jordan algorithm.
//------------------------------------------------------------------------


#include <lss_aprx.hpp>

using namespace cxsc;
using namespace std;

const real eps = 1e-15;

//------------------------------------------------------------------------

// MINV computes approximate inverse of the matrix A by use of the
// Gauss-Jordan algorithm.
// A   : input matrix and outpout of approximate inverse
// err : Error indicator = 0 everything ok, = 1 matrix singular
//------------------------------------------------------------------------

void MINV( rmatrix& A, int& error )
{
  int       i, j, k, au1, au2, ao1, ao2;
  bool      ok;
  real      h;
  rvector   aux( Lb(A,ROW),Ub(A,ROW) );
  intvector v( Lb(A,ROW),Ub(A,ROW) );
  int       l;

  au1 = Lb(A,ROW); ao1 = Ub(A,ROW);
  au2 = Lb(A,COL); ao2 = Ub(A,COL);

  ok = ((ao1 - au1) == (ao2 - au2))? true : false; // square matrix ?

  // Vector that will control what's happening with the column
  for(i = au1; i <= ao1; i++) v[i] = i;

  i = au1-1;  // Row index
  j = au2-1;  // Column index

  while ( (j < ao2) && ok )
  {
    i++; j++;  // i and j run simultaniously

    // pivot search
    if ( j < ao2)
    {
      l = i;
      for( k = i+1; k <= ao1; k++)
        if( abs(A[k][j] ) > abs( A[l][j] ) )
          l = k;

      // row interchange
      if(i != l)
      {
        aux = A[l]; A[l] = A[i]; A[i] = aux;
        k = v[l]; v[l] = v[i]; v[i] = k;
      }
    }

    // transformation
    if( A[i][j] == 0 ) ok = false;
```

```
        else
        {
          h = 1.0 / A[i][j];
          Col(A,j) = h * rvector( Col(A,j) );
          A[i][j] = h;
          for( k = au2; k <= ao2; k++)
            if( k != j)
            {
              for( l = au1; l <= ao1; l++)
                if( l != i )
                {
                  h = A[l][k] - A[l][j] * A[i][k];
                  if( h == 0.0) A[l][k] *= eps;
                    else A[l][k] = h;
                }
              A[i][k] *= -A[i][j];  // A[i][k] = -A[i][k] * A[i][j]
            }
        }
    }

    // column interchange
    if(ok)
      for( k = au1; k <= ao1; k++)
      {
        l = au2 + k - au1;
        while( v[k] != k )
        {
          i = v[k]; v[k] = v[i]; v[i] = i;
          j = au2 + i - au1;
          aux = rvector( Col(A,j) ); Col(A,j) = Col(A,l); Col(A,l) = aux;
        }
      }
    error = !ok;

  } // end MINV
```

The following module `lss.cpp` contains the functions which solve the problems (s), (o), (u), (S), (O), and (U) stated in Section 1 for a coefficient matrix of type *rmatrix* and a right hand side of type *rvector*.

There are only two global functions contained in `lss` – `LSS` and `INV` – which can be called from modules other than `lss`. `LSS` accepts linear systems of the form (1) with any matrix, checks the dimensions of the coefficient matrix `A`, and calls other functions which handle the square or rectangular cases. Similarly, `INV` checks its parameter `A` and calls other routines for the square or rectangular cases.

The functions have the following meanings (in order of their static appearance):

`REL`:

Computes the maximum relative error of the components of the two parameter vectors `A` and `B`. It is used in the stopping criterion of the real defect iteration for an improvement of the real approximations in the parts `USE_SINGLE_R` and `USE_DOUBLE_R` (both local to the first function `LSS`).

`TOO_BAD`:

Checks the accuracy of the parameter `A`. If at least one component is larger than $[-10^{20}, 10^{20}]$ then the function yields `TRUE` else `FALSE`. It is used to abort the interval iteration if it seems to diverge (again in `USE_SINGLE_R` and `USE_DOUBLE_R`).

`GUESS_ZEROES`:

Tries to guess whether a component of the solution is equal to zero. This is assumed to be the case if this component (i) has changed sign in the last two successive iterations or if (ii) it has decreased by more than a factor of 10^{-6} from the previous iterate to the current iterate and if it is less in magnitude than 10^{-6} times the maximum magnitude of all other components of the current iterate. Replacing such components by zero often improves the enclosures by many orders of magnitudes. On the other hand, if the guess was wrong, then zero is still a good approximation such that the enclosures will stay good.

`LSS`:

It implements the algorithm for a square system $Ax = b$ by first trying an iteration with an approximate inverse of single length (`USE_SINGLE_R`). In case we are not successful, compute an approximate inverse of double length (`USE_DOUBLE_R`). This function is the central function of the module. Input parameters are `A` and `b`. Output parameters are `Y` and `errcode`. The other parameters are for input as well as for output. Since `LSS` is called by the functions for matrix inversion several times, the computation of the approximate inverses (`R1,R2`) and the iteration matrix `C` is controlled by the parameter `FLAGS` which indicates in successive calls to `LSS` which of the matrices `R1, R2, C` have already been computed in order to avoid unnecessary and costly recomputation of these quantities. For details, see the comment at the beginning of `LSS`.

`USE_SINGLE_R`: (local to `LSS`)

This part implements the algorithm by using an approximate inverse `R1` of single length which is computed by the function `MINV`.

After the computation of `R1`, we compute an approximate solution `x1 = R1*b` and execute a defect iteration in floating point arithmetic to improve this approximation `x1`. It is essential to use scalar product expressions since the computation of the residuum `b - A*x0` must necessarily suffer from severe cancellation. This iteration is stopped if (i) the maximal relative error `p` is less than `delta` ($= 10^{-15}$) or if the relative error decreases very slowly only, such that `p` becomes larger than `bound`, which is multiplied by a factor of $\approx \sqrt{0.1}$ in each iteration starting after the 5th iteration. This means roughly that after the 5th iteration, we expect the approximation to gain at least one decimal digit each two iterations, otherwise we stop the iteration process.

Subsequent to this floating point iteration, we try to guess whether some of the solution components are exactly zero by calling function `GUESS_ZEROES`.

Next, we compute an enclosure of the residual `b - A*x1` in double length (real vector `y0` plus interval vector `Y1`) in order to get still high accuracy for the product with `R1`. Here the repeated computation of `A*x1` inside the #-expressions could be avoided if these expressions were computed component-wise by use of a variable of type *dotprecision*. However, since these computations are cheap relative to the rest of the algorithm, we prefer this version which is much easier to read.

If the residual is exactly the zero vector, `x1` is an exact solution of the system. Thus we can stop the algorithm at this point and need not continue with the costly interval iteration However, stopping here means that we have an exact solution, but that we do not know if it is unique. If we need uniqueness we have to continue with the interval iteration.

Finally the iteration matrix `C` is computed (again with scalar product expression because of cancellations), and the interval iteration for the defect `Y1` of the approximation `x1` is executed. Beginning with the 5th iteration, the inflation parameter `eps` is multiplied by 5 in each iteration in order to accelerate convergence or divergence. This iteration is stopped (i) if it was successful, i.e. if an iterate `Y1` is contained in its predecessor `YA`, (ii) after a maximum iteration count of 10, or (iii) if the accuracy becomes too bad (indicating divergence).

In the case of a successful interval iteration, the resulting enclosure `x1+Y1` of the solution is returned in `Y`, and the error code is set to zero.

`USE_DOUBLE_R`: (local to `LSS`)

This part implements the algorithm by using an approximate inverse `R1+R2` of double length (i.e. `R1` and `R2` are of type *rmatrix*), which is computed by the use of Rump's device as explained in Section 2.

The following steps are completely analogous to those in `USE_SINGLE_R` except that wherever the approximate inverse appears, the double length representation has to be used. Note that in the floating point defect iteration, the iterates are computed in double length also (`x1+x0`) but subsequently only the most significant part `x1` is used. Also, note that `y0` is an auxiliary variable in the floating point iteration. Here again we make use of a repeated computation of `A*x1` only to make the program easier to read.

`SQUARE_LSS`:

This function treats the case of a square coefficient matrix. The trivial case of a 1×1 matrix is solved explicitly. For $n > 1$, the storage for `R1, R2, C`, and `FLAGS` is allocated for a call of the function `LSS` described above.

`OVER_LSS`, `UNDER_LSS`:

These functions handle over- and under-determined linear systems, respectively. Both functions allocate variables `BIG_A,BIG_B`, and `BIG_Y` for the augmented problems and initialize `BIG_A` and `BIG_B` appropriately according to Section 2. Then this augmented square system is solved by a call to function `SQUARE_LSS`. Finally, those components of the big solution vector needed for the solution of the original problem are extracted.

`LSS`:

This function is the only global entry for the solution of linear systems in this module `lss`. It first checks the parameter for consistent dimensions, then decides which case (s), (o), or (u) has to be treated, and calls the appropriate function `SQUARE_LSS`, `OVER_LSS`, or `UNDER_LSS`.

`SQUARE_INV`, `OVER_INV`, `UNDER_INV`, `INV`:

These functions compute enclosures for the inverse matrices (or the pseudo inverse) and are completely analogous to those for linear systems. As mentioned above, `INV` is the only global function for this purpose and calls the other ones appropriately. The only difference from the corresponding `-LSS` functions is that they compute the inverse column-wise and therefore have to call the local linear system solving function `LSS` several times. Therefore, the matrices `R1, R2`, and `C` are defined locally in these functions and their computation is controlled by the variable `FLAG` as discussed previously (see function local `LSS`). In function `SQUARE_INV`, the resulting columns of the inverse matrix `Y` are first written into the rows of the result matrix. Therefore, `Y` has to be transposed at the end of the function.

```
//----------------------------------------------------------------------------

// File: lss (header)
// Purpose: Compute approximations to the solution x of Ax = b and an
//    approximations of the inverse of A. In both case the system (m x n)
//    can be square (m = n), over-determined (m > n) and under-determined
//    (m < n).
// Global functions:
//    LSS(): General entry for linear system solver.
//    INV(): General entry for matrix inversion.
//----------------------------------------------------------------------------


#ifndef _LSS_HPP
#define _LSS_HPP

#include <imatrix.hpp>   // Include real and interval types
                         // for Vector/Matrix
#include <mvi_util.hpp>  // Include real and interval utilities
                         // for Vector/Matrix
#include <lss_aprx.hpp>  // Include library for matrix inversion
#include <iostream>
#include <iomanip>       // for I/O manipulation

using namespace cxsc;
using namespace std;

extern  void LSS( rmatrix&, rvector&, ivector&, int& );
extern  void INV( rmatrix&, imatrix&, int& );

#endif

//----------------------------------------------------------------------------

// File: lss (implementation)
// Purpose: Compute approximations to the solution x of Ax = b and an
//   approximations of the inverse of A. In both case the system (m x n)
//   can be square (m = n), over-determined (m > n) and under-determined
//   (m < n).
// Global functions:
//   LSS(): General entry for linear system solver.
//   INV(): General entry for matrix inversion.
//----------------------------------------------------------------------------


#include <lss.hpp>

using namespace cxsc;
using namespace std;

const real zerotest = 1e6;
const real delta    = 1e-15;
const real eps1     = 1e-15;
const real sqrt_01  = 0.31622777;
const real limit    = 1e20;

static int m, n, dim;

static ivector null(ivector A)
{
  return (A = 0);
}


//------------------------------------------------------------------------

// REL computes componentwise the maximum relative error of A w.r.t B.
// if A[i] and B[i] do not have the same sign or if B[i] = 0, then
// rel. error = 0 for this component.
// A is always the new value of an iteration, B the old one.
//------------------------------------------------------------------------

real REL( rvector A, rvector B)
{
  int  i;
  real p,r,ai,bi;

  p=0;
  for(i=Lb(A);i<=Ub(A);i++) // A,B must have same index range
  {
    ai = A[i];
    bi = B[i];
    if( ai*bi <= 0.0 || zerotest*abs(ai) < abs(bi) ) r = 0.0;
      else r = abs( (ai-bi)/bi );
    if (r>p) p = r;
  }
  return p;
} // end REL


//------------------------------------------------------------------------

// TOO_BAD = accuracy of A is far too bad
// note: 0 for false, 1 for true;
//------------------------------------------------------------------------

bool TOO_BAD( ivector &A )
{
  int  i;
  bool bad;

  bad = false;
  for(i=Lb(A);i<=Ub(A);i++)
  {
    bad = bad || Inf(A[i]) < -limit && Sup(A[i]) > limit;
  }
  return bad;
} // end TOO_BAD


//------------------------------------------------------------------------

// x1 is the new, x0 the old value of an iteration. If a component of x1
// has decreased by more than a factor of zerotest, then this component
// is set to 0. The same is done if the sign of a component has changed.
//------------------------------------------------------------------------
```

```
void GUESS_ZEROES(rvector& x0, rvector& x1)
{
  int  i;
  real MAXX;

  MAXX = 0.0;
  for(i=Lb(x1);i<=Ub(x1);i++)
    if(abs(x1[i])>MAXX)
      for(i=Lb(x1);i<=Ub(x1);i++)
        if( x0[i]*x1[i] < 0.0 || zerotest*abs(x1[i]) < abs(x0[i])
            && MAXX > zerotest*abs(x1[i]) )
          x1[i] = 0.0;
} // end GUESS_ZEROES


//-------------------------------------------------------------------------

// The result of Y is an enclosure of the solution
// errcode = 0: Y is enclosure of the solution
// errcode = 1: no enclosure obtained, bad condition (?)
// errcode = 2: no enclosure obtained, matrix A singular (?)
//
// FLAGS = 0: R1,R2,C have not yet been computed
// FLAGS = 1: only R1 has been computed
// FLAGS = 2: R1 and corresponding C have been computed
// FLAGS = 3: R1 and R2 are computedbut not the corresponding C
// FLAGS = 4: R1, R2 and the corresponding C have been computed
//-------------------------------------------------------------------------

void LSS(rmatrix& A, rvector& b, ivector& Y, int& errcode, rmatrix& R1,
         rmatrix& R2, imatrix& C, int& FLAGS)
{
  rmatrix       D( Lb(A,1),Ub(A,1),Lb(A,2),Ub(A,2) ),
                R2temp( Lb(A,1),Ub(A,1),Lb(A,2),Ub(A,2) );
  rvector       x0( Lb(A,1),Ub(A,1) ), x1( Lb(A,1),Ub(A,1) ),
                y0( Lb(A,1),Ub(A,1) ), x1temp( Lb(A,1),Ub(A,1) );
  ivector       Y1( Lb(A,1),Ub(A,1) ), YA( Lb(A,1),Ub(A,1) ),
                Z( Lb(A,1),Ub(A,1) ),Y1temp( Lb(A,1),Ub(A,1) );
  int           i,j,k,err;
  bool          ready;
  real          p,bound,eps1;
  dotprecision  accu;
  idotprecision iaccu;

  ready = false;

// begin USE_SINGLE_R - compute approximate inverse of A
  if (FLAGS<3)
  {
    err = 0;
    if (FLAGS<1)
    {
      R1 = A;
      MINV( R1, err );
      if (err==0) FLAGS = 1;
    }
    if (err==0)
// floating point defect iteration: result is x1
    {
      bound = 100.0*sqrt_01;
      x1 = R1*b;
      k = 0;
      do
```

```
        // iterate x = x + R*(b-Ax)
        {
          k = k + 1;
          x0 = x1;

          // x1 := #*(b - A*x0)
          for (i=Lb(x0);i<=Ub(x0);i++)
          {
            accu = b[i];
            accumulate(accu,-A[Row(i)],x0);
            x1[i] = rnd(accu);
          }

          // x1 := #*(x0 + R1*x1)
          for (i=Lb(x1);i<=Ub(x1);i++)
          {
            accu = x0[i];
            accumulate(accu,R1[Row(i)],x1);
            x1temp[i] = rnd(accu);
          }
          x1 = x1temp;

          p = REL(x1,x0);
          if(k>5) bound = bound*sqrt_01;
        } while ((p<bound || k<=5) && p>=delta);

        GUESS_ZEROES(x1,x0);

  // Compute enclosure y0+y1 of the residuum b-A*x1 of the aproximation x1
  // and initialize Y1:=Z:= R1*(b-A*x1), C:= I-R1*A

        // y0 := #*(b-A*x1)
        for (i=Lb(x1);i<=Ub(x1);i++)
        {
          accu = b[i];
          accumulate(accu,-A[Row(i)],x1);
          y0[i] = rnd(accu);
        }

        // Y1 := ##(b-A*x1-y0)
        for (i=Lb(x1);i<=Ub(x1);i++)
        {
          accu = b[i];
          accumulate(accu,-A[Row(i)],x1);
          accu = accu - y0[i];
          rnd(accu,Y1[i]);
        }

        // Y1 := ##( R1*y0 + R1*Y1 );
        for (i=Lb(R1,1);i<=Ub(R1,1);i++)
        {
          iaccu = 0.0;
          accumulate(iaccu,R1[Row(i)],y0);
          accumulate(iaccu,R1[Row(i)],Y1);
          rnd(iaccu,Y1temp[i]);
        }
        Y1 = Y1temp;
        Z = Y1;

        if (Z==null(Z))
        {
          Y = x1; // exact solution! (however, not necessarily unique!)
          errcode = 0;
          ready = true;
        }
          else
          {
            if (FLAGS<2)
            {
              // C := ##( ID(A) - R1*A );
              for (i=Lb(A,1);i<=Ub(A,1);i++)
                for (j=Lb(A,2);j<=Ub(A,2);j++)
                {
                  accu = ( i == j) ? 1.0 : 0.0;
                  accumulate(accu,-R1[i],A[Col(j)]);
                  rnd(accu,C[i][j]);
                }

              FLAGS = 2;
            }

  // interval iteration until inclusion is obtained
  // (or max. iteration count)
            k = 0;
            eps1 = 0.1;
            do
            {
              if (k>=5) eps1 = 5*eps1;
              k = k+1;
              YA = Blow(Y1,eps1);
              Y1 = Z + C*YA;
              ready = in(Y1,YA);
            } while ( !ready && k<10 && !TOO_BAD(Y1) );

            // output of the result
            if (ready)
            {
              Y = x1 + Y1;
              errcode = 0;
            }
          }
      }
        else ready = false;
    } // end USE_SINGLE_R

  // if no success: try again with approximate inverse
  // R = R1+R2 of double length
  // USE_DOUBLE_R to try again with approximate inverse
  // R = R1+R2 of double length
    if (!ready)
    {
      err = 0;
      if (FLAGS<3)
      {
        R2 = R1*A;
        MINV(R2,err);
        if (err==0)
        {
          FLAGS = 3;
          D = R2*R1;

          // R2 := #* (R2*R1 - D);
          for (i=Lb(R1,1);i<=Ub(R1,1);i++)
            for (j=Lb(R1,2);j<=Ub(R1,2);j++)
            {
              accu = -D[i][j];
```

```
              accumulate(accu,R2[i],R1[Col(j)]);
              R2temp[i][j] = rnd(accu);
            }
          R2 = R2temp;
          R1 = D;
        }
      }

      if (err==0)
      // floating point defect iteration: result is x1+x0
      {
        bound = 100.0*sqrt_01;

        // x1 := #*( R1*b + R2*b );
        for (i=Lb(R1,1);i<=Ub(R1,1);i++)
        {
          accu = 0.0;
          accumulate(accu,R1[Row(i)],b);
          accumulate(accu,R2[Row(i)],b);
          x1[i] = rnd(accu);
        }

        // x0 = #*( R1*b + R2*b - x1):
        for (i=Lb(R1,1);i<=Ub(R1,1);i++)
        {
          accu = -x1[i];
          accumulate(accu,R1[Row(i)],b);
          accumulate(accu,R2[Row(i)],b);
          x0[i] = rnd(accu);
        }

        k = 0;
        do
        // iteration x = x + (R1+R2)*(b-Ax), x = x1 + x0)
        {
          k = k+1;

          // y0 = #*(b - A*x1 - A*x0)
          for (i=Lb(A,1);i<=Ub(A,1);i++)
          {
            accu = b[i];
            accumulate(accu,-A[Row(i)],x1);
            accumulate(accu,-A[Row(i)],x0);
            y0[i] = rnd(accu);
          }

          // y0 := #*(x0 + R1*y0 + R2*y0);
          for (i=Lb(R1,1);i<=Ub(R1,1);i++)
          {
            accu = x0[i];
            accumulate(accu,R1[Row(i)],y0);
            accumulate(accu,R2[Row(i)],y0);
            x1temp[i] = rnd(accu);
          }
          y0 = x1temp;

          p = REL (x1+y0,x1+x0);
          y0 = x1 + y0;

          // x0 := #*(x1 + x0 - y0)
          for (i=Lb(x1);i<=Lb(x1);i++)
          {
            accu = x1[i] + x0[i] - y0[i];
            x0[i] = rnd(accu);
          }
          x1 = y0;

          if (k>5) bound = bound * sqrt_01;
        } while ( (p<bound || k<=5) && p>=delta );

  // compute enclosure y0+Y1 of the residuum b-A*x1 of the approximation
  // x1 and initialize Y1:= Z:= (R1+R2)*(b-A*x1), C:= I-(R1+R2)*A

        // y0 := #*(b-A*x1)
        for (i=Lb(x1);i<=Ub(x1);i++)
        {
          accu = b[i];
          accumulate(accu,-A[Row(i)],x1);
          y0[i] = rnd(accu);
        }

        // Y1 := ##(b-A*x1-y0)
        for (i=Lb(x1);i<=Ub(x1);i++)
        {
          accu = b[i];
          accumulate(accu,-A[Row(i)],x1);
          accu = accu - y0[i];
          rnd(accu,Y1[i]);
        }

        // Y1 := ##(R1*y0 + R2*y0 + R1*Y1 + R2*Y1 );
        for (i=Lb(R1,1);i<=Ub(R1,1);i++)
        {
          accu = 0.0;
          accumulate(accu,R1[Row(i)],y0);
          accumulate(accu,R2[Row(i)],y0);
          iaccu = accu;
          accumulate(iaccu,R1[Row(i)],Y1);
          accumulate(iaccu,R2[Row(i)],Y1);
          rnd(iaccu,Y1temp[i]);
        }
        Y1 = Y1temp;
        Z = Y1;

        if (Z==null(Z))
        {
          Y = x1; // exact solution! (however, not necessarily unique!)
          errcode = 0;
          ready = true;
        }
          else
          {
            if (FLAGS<4)
            {
              // C:= ## (ID(A) - R1*A - R2*A);
              for (i=Lb(A,1);i<=Ub(A,1);i++)
                for (j=Lb(A,2);j<=Ub(A,2);j++)
                {
                  accu = ( i == j) ? 1.0 : 0.0;
                  accumulate(accu,-R1[i],A[Col(j)]);
                  accumulate(accu,-R2[i],A[Col(j)]);
                  rnd(accu,C[i][j]);
                }

              FLAGS = 4;
            }
```

```
// interval iteration until inclusion is obtained
// (or max. iteration count)
          k = 0;
          eps1 = 0.1;
          do
          {
            if (k>=5) eps1 = 5*eps1;
            k = k+1;
            YA = Blow( Y1, eps1);
            Y1 = Z + C*YA;
            ready = in(Y1,YA);
          } while ( !ready && k<10 && !TOO_BAD(Y1) );

// output of the result
          if (ready)
          {
            Y = x1 + Y1;
            errcode = 0;
          }
            else errcode = 1;
        }
    }
      else errcode = 2;
  } // end USE_DOUBLE_R
}  // end LSS


//------------------------------------------------------------------------
// Linear system: square matrix
// The result y is an enclosure of the solution of Ax = b
//------------------------------------------------------------------------
void SQUARE_LSS( rmatrix& A, rvector& b, ivector& y, int& errcode)
{
  rmatrix       R1(Lb(A,ROW),Ub(A,ROW),Lb(A,ROW),Ub(A,ROW)),
                R2(Lb(A,ROW),Ub(A,ROW),Lb(A,ROW),Ub(A,ROW));
  imatrix       C(Lb(A,ROW),Ub(A,ROW),Lb(A,ROW),Ub(A,ROW));
  idotprecision Accu;
  int           FLAGS;

  if( dim == 1 )     // Treat trivial case separately
    if( A[Lb(A,ROW)][Lb(A,COL)] == 0.0 ) errcode = 2;
      else
      {
        y[Lb(y)] = interval( b[Lb(b)] ) / A[Lb(A,ROW)][Lb(A,COL)];
        errcode = 0;
      }
    else
    {
      FLAGS = 0;
      LSS( A, b, y, errcode, R1, R2, C, FLAGS );
    }
} // end SQUARE_LSS


//------------------------------------------------------------------------
// Linear system: over-determined case
// The result Y is an enclosure of the solution x of AH*A*x = AH*b,
// i.e. x is least squares solution of Ax = b.
//                                                 | A  -I | |x|   |b|
// From Ax = b we generate the (n+m)x(n+m)-system  |        | | | = | |
//                                                 | 0  AH | |y|   |0|
//
// Here, AH is the hermitian of A (transpose in the real case)
//-------------------------------------------------------------------------

void OVER_LSS( rmatrix& A, rvector& b, ivector& y, int& errcode)
{
  rmatrix BIG_A(1,dim,1,dim);
  rvector BIG_b(1,dim);
  ivector BIG_y(1,dim);
  int     i, j;

  BIG_A( 1,m, 1,n ) = A;

  // BIG_A( 1,m, n+1,n+m ) = -Id(m);
  for( i = 1; i <= m; i++)
    for( j = n+1; j <= n+m; j++)
      (j == i+2)? BIG_A[i][j]=-1 : BIG_A[i][j] = 0;

  BIG_A( m+1,m+n, 1,n ) = 0.0;
  BIG_A( m+1,m+n, n+1,n+m ) = transp(A);
  BIG_b( 1,m ) = b;
  BIG_b( m+1,m+n ) = 0.0;

  SQUARE_LSS( BIG_A, BIG_b, BIG_y, errcode );

  y = BIG_y( 1,n );
} // end OVER_LSS


//-------------------------------------------------------------------------
// Linear system: under-determined case
// The result Y is an enclosure of Y = AH*x with A*AH*x = b,
// i.e. y is solution of Ay = b with minimal Euklidian norm.
//                                                 |AH  -I | |x|   |0|
// From Ax = b we generate the (n+m)x(n+m)-system  |       | | | = | |
//                                                 | 0   A | |y|   |b|
//
// Here, AH is the hermitian of A (transpose in the real case)
//-------------------------------------------------------------------------

void UNDER_LSS( rmatrix& A, rvector& b, ivector& y, int& errcode)
{
  rmatrix BIG_A(1,dim,1,dim);
  rvector BIG_b(1,dim);
  ivector BIG_y(1,dim);
  int     i, j;

  BIG_A( 1,n, 1,m ) = transp(A);

  //BIG_A( 1,n, m+1,m+n ) = -Id(m);
  for( i = 1; i <= n; i++)
    for( j = m+1; j <= n+m; j++)
      (j == i+2)? BIG_A[i][j]=-1 : BIG_A[i][j] = 0;

  BIG_A( n+1,n+m, 1,m ) = 0.0;
  BIG_A( n+1,n+m, m+1,m+n ) = A;
  BIG_b( 1,n ) = 0.0;
  BIG_b( n+1,n+m ) = b;

  SQUARE_LSS( BIG_A, BIG_b, BIG_y, errcode );
  y = BIG_y( m+1,m+n );
} // end UNDER_LSS


//------------------------------------------------------------------------
// General entry for linear system solver: decides which case to treat
//------------------------------------------------------------------------

void LSS( rmatrix& A, rvector& b, ivector& y, int& errcode )
{
  errcode = 0;
  m = Ub(A,ROW) - Lb(A,ROW) + 1;
  n = Ub(A,COL) - Lb(A,COL) + 1;

  dim = m+n;

  if( m != Ub(b) - Lb(b) + 1 ) errcode = 3;  // b : wrong dimension
  if( n != Ub(y) - Lb(y) + 1 ) errcode = 4;  // y : wrong dimension

  if( errcode == 0 )
    if( m > n ) OVER_LSS( A, b, y, errcode ); // over-determined system
      else                                    // under-determined system
        if( m < n ) UNDER_LSS( A, b, y, errcode );
          else
          {
            dim = n;
            SQUARE_LSS( A, b, y, errcode ); // square system
          }
} // end LSS


//------------------------------------------------------------------------
// Inverse matrix: square case
// The result Y is an enclosure of the solution of AY = I
//------------------------------------------------------------------------

void SQUARE_INV( rmatrix& A, imatrix& Y, int& errcode)
{
  rmatrix R1(Lb(A,ROW),Ub(A,ROW),Lb(A,ROW),Ub(A,ROW)),
          R2(Lb(A,ROW),Ub(A,ROW),Lb(A,ROW),Ub(A,ROW));
  imatrix C(Lb(A,ROW),Ub(A,ROW),Lb(A,ROW),Ub(A,ROW));
  imatrix temp(Lb(A,ROW),Ub(A,ROW),Lb(A,ROW),Ub(A,ROW));
  rvector b(Lb(A,ROW),Ub(A,ROW));
  ivector xTemp(Lb(A,ROW),Ub(A,ROW));
  int     i, err, FLAGS;

  if( dim == 1 )     // Treat trivial case separately
    if( A[Lb(A,ROW)][Lb(A,COL)] == 0.0 ) errcode = 2;
      else
      {
        Y[Lb(Y,ROW)][Lb(Y,COL)] = interval(1.0) / A[Lb(A,ROW)][Lb(A,COL)];
        errcode = 0;
      }
    else
    {
      FLAGS = 0;
      err = 0;
      b = 0.0;
      for(i = Lb(A,ROW); i <= Ub(A,COL); i++)
      {
        b[i] = 1.0;
        LSS( A,b,xTemp,errcode,R1,R2,C,FLAGS );
        Y[Lb(Y,ROW)+i-Lb(A,ROW)] = xTemp;
        if( errcode > err ) err = errcode;
        b[i] = 0.0;
      }
      errcode = err;

      //Computing: Y = transp(Y);
      for( i = Lb(Y,ROW); i <= Ub(Y,COL); i++)
        Row(temp,i) = Col(Y,i);
      Y = temp;
    }
} // end SQUARE_INV


//--------------------------------------------------------------------------

// Inverse matrix: over-determined case
// The result Y is an enclosure of the pseudo inverse A+ of A.
//                               -1
// If m > n then Y = A+ = (AH*A)  * AH is solution of:
//
// | A  -I | |Y|   |I|  ( A: mxn           )
// |       | | | = | |  ( X = AY - I: mxm  )
// | 0  AH | |X|   |0|  ( Y = A+    : nxm  )
//                      ( right hand side: )
//                      ( I : mxm, 0 : nxm )
//--------------------------------------------------------------------------

void OVER_INV( rmatrix& A, imatrix& Y, int& errcode)
{
  rmatrix R1(1,dim,1,dim), R2(1,dim,1,dim), BIG_A(1,dim,1,dim);
  imatrix C(1,dim,1,dim);
  rvector BIG_b(1,dim);
  ivector BIG_Y(1,dim);
  int     i, j, err, FLAGS;

  BIG_A( 1,m, 1,n ) = A;

  // BIG_A( 1,m, n+1,n+m ) = -Id(m);
  for( i = 1; i <= m; i++)
    for( j = n+1; j <= n+m; j++ )
      (j == i+2)? BIG_A[i][j]=-1 : BIG_A[i][j] = 0;

  BIG_A( m+1,m+n, 1,n ) = 0.0;
  BIG_A( m+1,m+n, n+1,n+m ) = transp(A);
  BIG_b = 0.0;
  FLAGS = 0;
  err = 0;
  for( j = 1; j <= m; j++ )
  {
    BIG_b[j] = 1.0;
    LSS( BIG_A,BIG_b,BIG_Y,errcode,R1,R2,C,FLAGS );
    if( errcode > err ) err = errcode;
    BIG_b[j] = 0.0;
    Col(Y,Lb(Y,COL)+j-1) = BIG_Y(1,n);
  }
  errcode = err;
} // end OVER_INV


//--------------------------------------------------------------------------
```

```
// Inverse matrix: under-determined case
// The result Y is an enclosure of the pseudo inverse A+ of A.
//                                  -1
// If m < n then Y = A+ = AH*(A*AH)  is solution of:
//
// |AH  -I | |YH|   |I|  ( A: mxn           )
// |       | |  | = | |  ( X = AY - I: nxn  )
// | 0   A | |XH|   |0|  ( Y = A+    : nxm  )
//                       ( right hand side: )
//                       ( I : nxn, 0 : mxn )
//------------------------------------------------------------------------

void UNDER_INV( rmatrix& A, imatrix& Y, int& errcode)
{
  rmatrix R1(1,dim,1,dim), R2(1,dim,1,dim), BIG_A(1,dim,1,dim);
  imatrix C(1,dim,1,dim);
  rvector BIG_b(1,dim);
  ivector BIG_Y(1,dim);
  int     i, j, err, FLAGS;

  BIG_A( 1,n, 1,m ) = transp(A);

  //BIG_A( 1,n, m+1,m+n ) = -Id(m);
  for( i = 1; i <= n; i++)
    for( j = m+1; j <= n+m; j++)
      (j == i+2)? BIG_A[i][j]=-1 : BIG_A[i][j] = 0;

  BIG_A( n+1,n+m, 1,m ) = 0.0;
  BIG_A( n+1,n+m, m+1,m+n ) = A;
  BIG_b = 0.0;
  FLAGS = 0;
  err = 0;
  for( i = 1; i <= n; i++ )
  {
    BIG_b[i] = 1.0;
    LSS( BIG_A,BIG_b,BIG_Y,errcode,R1,R2,C,FLAGS );
    if( errcode > err ) err = errcode;
    BIG_b[i] = 0.0;
    Y[Lb(Y,ROW)+i-1] = BIG_Y(1,m);
  }
  errcode = err;
} // end UNDER_INV


//------------------------------------------------------------------------

// General entry for matrix inversion: decides which case to treat
//------------------------------------------------------------------------

void INV( rmatrix& A, imatrix& Y, int& errcode )
{
  errcode = 0;
  m = Ub(A,ROW) - Lb(A,ROW) + 1;
  n = Ub(A,COL) - Lb(A,COL) + 1;
  dim = m+n;

  if(n!= Ub(Y,ROW)-Lb(Y,ROW)+1 ) errcode = 3;
  // Y : wrong number of rows
  if(m!= Ub(Y,COL)-Lb(Y,COL)+1 ) errcode = 4;
  // Y : wrong number of columns

  if( errcode == 0 )
```

```
        if( m > n ) OVER_INV( A, Y, errcode ); // over-determined system
          else
            if( m < n ) UNDER_INV( A, Y, errcode ); // under-determined system
              else
              {
                dim = n;
                SQUARE_INV( A, Y, errcode ); // square system
              }
    } // end INV
```

To conclude this section, we want to indicate changes required in the preceding C-XSC program code to yield programs that treat the cases of *interval, complex*, and *complex interval* input data.

First we consider the case of interval data, i.e. the input matrix A is an interval matrix, and the right hand side b is an interval vector.

We construct a new file header `ilss.hpp` from the file header `lss.hpp` by applying the following changes:

- Change the statements
  `#ifndef _LSS_HPP` and `#define _LSS_HPP` by
  `#ifndef _ILSS_HPP` and `#define _ILSS_HPP`.

- Change the statements
  `extern void LSS( rmatrix&, rvector&, ivector&, int& );` and
  `extern void INV( rmatrix&, imatrix&, int& );` by
  `extern void LSS( imatrix&, ivector&, ivector&, int& );` and
  `extern void INV( imatrix&, imatrix&,int& );`.

We construct a new module `ilss` from the module `lss` by applying the changes which we describe now.

These are the changes in the first (local) function `LSS`:

- In the head of the function make `A` an `imatrix` and `b` an `ivector`.

- In the declaration part, add variables `AM` of type `rmatrix` and `bm` of type `rvector` which will be used to hold the midpoints of `A` and `b` and the variable `iaccu` of type `idotprecision`.

- In this function, replace the variable `accu` by `iaccu` in *all* statements that need to use interval variables of type `imatrix` and `ivector`.

- At the beginning of the function body, add the statements
  `AM = mid(A);` and `bm = mid(b);`.

- In the part `USE_SINGLE_R`, replace the variables `A` by `AM` and `b` by `bm` in each of the four statements:
  `R1 = A;`
  `x1 = R1*b;`
  `accu = b[i];` and
  `accumulate(accu,-A[Row(i)],x0);`

- Also in `USE_SINGLE_R`, replace (the variable `itemp` is of type `interval`) `y0[i]=rnd(accu);` by `rnd(iaccu,itemp); y0[i] = mid(itemp);`

- In the part `USE_DOUBLE_R`, replace the variables `A` by `AM` and `b` by `bm` in each of the six statements:
  `R2 = R1*A;` `accumulate(accu,R1[Row(i)],b);`
  `accumulate(accu,R2[Row(i)],b);` `accu = b[i];`
  `accumulate(accu,-A[Row(i)],x1);` and
  `accumulate(accu,-A[Row(i)],x0);`

- Also in `USE_DOUBLE_R`, replace (the variable `itemp` is of type `interval`) `y0[i]=rnd(accu);` by `rnd(iaccu, itemp);` `y0[i] = mid(itemp);`

The other functions in the module must be modified as follows:

- Make `A` an `imatrix` and `b` an `ivector` in the heads of the functions: `SQUARE_LSS`, `OVER_LSS`, `UNDER_LSS`, `LSS`, `SQUARE_INV`, `OVER_INV`, `INV` and `UNDER_INV`.

- In `SQUARE_LSS` and in `SQUARE_INV` delete the conversion function `interval` in the special case `dim = 1` and in both functions replace the test `A[Lb(A,ROW)][Lb(A,COL)] == 0.0` by the test `in(0.0,A[Lb(A,ROW)][Lb(A,COL)])`.

- In the functions `OVER_LSS`, `UNDER_LSS`, `OVER_INV` and `UNDER_INV` change the types of the variables `BIG_A` and `BIG_b` to the types `imatrix` and `ivector`.

- In function `SQUARE_INV`, change the type of the local variable `b` from `rvector` to `ivector`.

Next, we turn to the case of a complex input matrix A and a complex input vector b. In this case, we also have to modify the function `MINV`to compute an approximate inverse of a complex matrix.

We construct a new header file `clssaprx.hpp` from the header file `lss_aprx.hpp` by applying the following changes:

- Change the statements
  `#ifndef _LSS_HPP`, and `#define _LSS_HPP` to
  `#ifndef _CLSS_HPP`, and `#define _CLSS_HPP`.

- Delete *all* statements `#include` and include the following statements:
  `#include <cimatrix.hpp>`,
  `#include <intvector.hpp>`,
  `#include <ci_util.hpp>`, and
  `#include <iostream>`.

- Change the statement
  `extern void MINV(rmatrix&, int& );` to
  `extern void MINV( cmatrix&, int& );`.

Now we write a new module `clssaprx` by copying `lss_aprx` with the following modifications:

- The types of the parameter `A` of `MINV` and the local variables `aux` and `h` must be replaced by their equivalent complex types `cmatrix`, `cvector`, and `complex`.
- The two occurrences of the conversion function `rvector` in the body of `MINV` must be replaced by the conversion function `cvector`.

We construct a new header file `clss.hpp` from the header file `lss.hpp` by applying the following changes:

- Change the statements
  `#ifndef _LSS_HPP`, and `#define _LSS_HPP` to
  `#ifndef _CLSS_HPP`, and `#define _CLSS_HPP`.
- Delete *all* statements `#include` and include the following statements:
  `#include <clssaprx.hpp>`, and
  `#include <iomanip.hpp>`.
- Change the statements
  `extern void LSS( rmatrix&, rvector&, ivector&, int& );` and
  `extern void INV( rmatrix&, imatrix&, int& );` to
  `extern void LSS( cmatrix&, cvector&, civector&, int& );` and
  `extern void INV( cmatrix&, cimatrix&, int& );`.

A module `clss` will now be constructed as a copy of the module `lss` by applying the following changes:

- Replace *all* occurrences of `rmatrix` by `cmatrix`, `rvector` by `cvector`, `imatrix` by `cimatrix`, and `ivector` by `civector` throughout the module.
- In function `REL`, change the type of `ai, bi` from `real` to `complex` and delete the test `ai*bi <= 0.0` without replacement.
- In function `TOO_BAD`, replace statement
  ```
  bad = bad || Inf(A[i]) < -limit && Sup(A[i]) > limit;
  ```
  by
  ```
  bad = bad || InfRe(A[i]) < -limit && SupRe(A[i]) > limit
            || InfIm(A[i]) < -limit && SupIm(A[i]) > limit;
  ```
- In function `GUESS_ZEROES`, delete the test `x0[i]*x1[i]<0.0` without replacement.
- In the functions `OVER_LSS`, `UNDER_LSS`, `OVER_INV`, and `UNDER_INV` we must replace all function calls to the function `TRANSP` by calls to the function `HERM`. This function must be included in the module `clss`. The code of the function `HERM` is as follows:

```
cmatrix Herm( cmatrix &A )
{
 int i;
 cmatrix herm(Lb(A,COL),Ub(A,COL),Lb(A,ROW),Ub(A,ROW));
 for(i=Lb(A,ROW);i<=Ub(A,ROW);i++)
 {
     Col(herm,i) = Row(A,i);
 }
 return SetIm(herm, -1*Im(herm) );
}
```

- Additionally, in `UNDER_INV`, the statement `Y[Lb(Y,ROW)+i-1] = BIG_Y(1,m);` must be replaced by `Y[Lb(Y,ROW)+i-1] = conj( BIG_Y(1,m) );`.

Finally, a new module `cilss` can be constructed from the module `ilss` by applying exactly the same changes as for the conversion from `lss` to `clss`.

6 Test Results

A very well known set of ill conditioned test matrices for linear system solvers are the $n \times n$ Hilbert matrices H_n with entries $(H_n)_{i,j} := \dfrac{1}{i+j-1}$. As a test problem, we report the results of our program for the linear systems $H_n x = e_1$, where e_1 is the first canonical unit vector. Thus the solution x is the first column of the inverse H_n^{-1} of the Hilbert matrix H_n. We give results for the cases $n = 10$ and $n = 20$. Since the elements of these matrices are rational numbers which can not be stored exactly in floating point, we do not solve the given problems directly but rather we multiply the system by the least common multiple lcm_n of all denominators in H_n. Then the matrices will have integer entries which makes the problem exactly storable in IEEE floating point arithmetic. For $n = 10$, we have $lcm_{10} = 232792560$ and for $n = 20$, we have $lcm_{20} = 5342931457063200$.

For the system $(lcm_{10}H_{10})x = (lcm_{10}e_1)$, the program computes the result

x_1	1.000000000000000E+002
x_2	-4.950000000000000E+003
x_3	7.920000000000000E+004
x_4	-6.006000000000000E+005
x_5	2.522520000000000E+006
x_6	-6.306300000000000E+006
x_7	9.609600000000000E+006
x_8	-8.751600000000000E+006
x_9	4.375800000000000E+006
x_{10}	-9.237800000000000E+005

which is the exact solution of this ill conditioned system.

For the system $(lcm_{20}H_{20})x = (lcm_{20}e_1)$, the program computes the enclosures (here an obvious short notation for intervals is used)

x_1	${}^{4.000000000000001}_{3.999999999999999}E+002$
x_2	$-7.9^{79999999999998}_{80000000000002}E+004$
x_3	$5.266^{800000000001}_{799999999999}E+006$
x_4	$-1.71609^{8999999999}_{9000000001}E+008$
x_5	$3.2949100^{80000001}_{79999999}E+009$
x_6	$-4.118637^{599999999}_{600000001}E+010$
x_7	$3.5694859^{20000001}_{19999999}E+011$
x_8	$-2.23730278^{1999999}_{2000001}E+012$
x_9	$1.044074631^{600001}_{599999}E+013$
x_{10}	$-3.7006645275^{59999}_{60001}E+013$
x_{11}	$1.0092721438^{80001}_{79999}E+014$
x_{12}	$-2.1332343041^{09999}_{10001}E+014$
x_{13}	$3.5006921913^{60001}_{59999}E+014$
x_{14}	$-4.4431862428^{79999}_{80001}E+014$
x_{15}	$4.31623806451^{2001}_{1999}E+014$
x_{16}	$-3.1472569220^{39999}_{40001}E+014$
x_{17}	$1.6661948410^{80001}_{79999}E+014$
x_{18}	$-6.044040109^{799999}_{800001}E+013$
x_{19}	$1.343120024^{400001}_{399999}E+013$
x_{20}	$-1.378465288^{199999}_{200001}E+012$

which is an extremely accurate enclosure for the exact solution. (The exact solution components are the integers within the computed intervals).

7 Notes and References

In the program the case $n = 2$ should of course be computed separately, as it is done for $n = 1$. This concerns the functions `SQUARE_LSS` and `SQUARE_INV` in module `lss` as well as `MINV` in `lss_aprx`. We have omitted these cases to keep the procedures somewhat shorter. Even $n = 3$ might be profitably programmed directly.

The form of the result of iterated defect correction in the algorithm

$$x \in \tilde{x} + [y]$$

is a special case of a staggered correction format [14], which has the more general form

$$x \in \sum_{i=1}^{k} x_i + [y],$$

where the sum of the x_i is an approximation of x, and $[y]$ is an enclosure of its error. Such a representation can easily be obtained by a modification of our algorithm in such a way that several iterated defect correction steps are performed.

If the algorithm is modified in the way that it produces output in staggered correction format, it is almost trivial to make a further modification to allow input of the coefficient matrix A and the right hand side b in staggered form. Then we have a verifying multi-precision linear system solver.

Such a linear system solver in staggered format can be applied to solve the least squares problem and compute the pseudo inverse in a different and cheaper way from the method in our program. For the least squares problem, we proceed as follows (see [27]): In the case $m \geq n$ we compute $B := A^H A$ and $d := A^H b$ in double length (i.e. B and d have staggered format) and solve $Bx = d$ with the staggered linear system solver. In the case $m \leq n$, we compute $B := AA^H$ in double length and solve $Bx = b$ with the staggered linear system solver. Then, finally, $y = A^H x$ is the desired solution (which must be computed in double length).

The methods presented in this section were originally developed by Rump [23], [24], some ideas go back to Wilkinson [32]. Some newer methods exist using LU-Factorization with backward error estimation for the solution of full triangular systems [11]. These algorithms can be adapted to the condition of the system and use less computing time than ours in the case of well conditioned problems.

Special methods for symmetric matrices also exist. Such special methods give better results especially in the case of an interval matrix A and have somewhat less overhead.

For symmetric and positive definite matrices, a variant of the Cholesky method has been investigated in [5].

Similarly, different methods exist for M-matrices and H-matrices. For example, the interval Gauss elimination can be applied in these cases. See [3], [4], or [21].

Another popular class of methods are iterative methods not directly based on defect correction. See also [18], [19], or [20].

Finally, we note that our algorithm is not well suited for large systems which are sparse or which have banded structure since the approximate inverse R will be in general a full matrix. A special C-XSC implementation for these cases is discussed in [8] (see also [14], or [17]).

References

[1] Adams, E., Kulisch, U. (Eds.): *Scientific Computing with Automatic Result Verification*. I. Language and Programming Support for Verified Scientific Computation, II. Enclosure Methods and Algorithms with Automatic Result Verification, III. Applications in the Engineering Sciences. Academic Press, San Diego, 1993.

[2] Albrecht, R., Kulisch, U. (Eds.): *Grundlagen der Computerarithmetik*. Computing Supplementum **1**, Springer-Verlag, Wien, New York, 1977.

[3] Alefeld, G.: *Über die Durchführbarkeit des Gaußschen Algorithmus bei Gleichungen mit Intervallen als Koeffizienten*. In [2], pp 15–19, 1977.

[4] Alefeld, G., Herzberger, J.: *Introduction to Interval Computations*. Academic Press, New York, 1983.

[5] Alefeld, G., Mayer, G.: *The Cholesky Method for Interval Data*. Presentation at the International Conference on "Numerical Analysis with Automatic Result Verification". Lafayette, Louisiana, USA, Feb. 25 – March 1, 1993.

[6] Cornelius, H., Lohner, R.: *Computing the Range of Values of Real Functions with Accuracy Higher Than Second Order*. Computing **33**, pp 331–347, 1984.

[7] Hofschuster, W., Krämer, W., Wedner, S., Wiethoff, A.: *C-XSC 2.0: A C++ Class Library for Extended Scientific Computing*. Universität Wuppertal, Preprint BUGHW - WRSWT 2001/1, 2001.

[8] Hölbig, C., Krämer, W.: *A Selfverifying Solver for Linear Systems with Banded Coefficient Matrix Realized in C-XSC*. Universität Wuppertal, Preprint BUGHW - WRSWT 2003/2, 2003.

[9] ISO/IEC 14882 International Standard: *Programming languages – C++*, 1998.

[10] Jahn, K.-U. (Ed.): *Computernumerik mit Ergebnisverifikation*. Problemseminar, Technische Hochschule Leipzig, 13.-15. März 1991. Proceedings in Wissenschaftliche Zeitschrift der Technischen Hochschule Leipzig, Jahrgang 15, Heft 6, 1991.

[11] Jansson, C.: *A Fast Direct Method for Computing Verified Inclusions*. Berichte des Forschungsschwerpunktes Informations- und Kommunikationstechnik, Technische Universität Hamburg-Harburg, Bericht 90.4, 1990.

[12] Kaucher, E., Kulisch, U., Ullrich, Ch. (Eds.): *Computerarithmetic: Scientific Computation and Programming Languages*. B. G. Teubner Verlag, Stuttgart, 1987

[13] Kaucher, E., Mayer, G., Markov, S. M. (Eds.): *Computer Arithmetic, Scientific Computation and Mathematical Modelling*. Proceedings of SCAN-90. IMACS Annals on Computing and Applied Mathematics, Vol. **12** (1992), published Oct. 1991. J. C. Baltzer AG, Basel, 1991.

[14] Krämer, W., Kulisch, U., Lohner, R.: *Numerical Toolbox for Verified Computing II - Advanced Numerical Problems*. Universität Karlsruhe, 1994, *http://www.uni-karlsruhe.de/˜Rudolf.Lohner/papers/tb2.ps.gz*.

[15] Kulisch, U., Miranker, W. L. (Eds.): *A New Approach to Scientific Computation*. Proceedings of Symposium held at IBM Research Center, Yorktown Heights, N.Y., 1982. Academic Press, New York, 1983.

[16] Kulisch, U., Stetter, H. J. (Eds.): *Scientific Computation with Automatic Result Verification*. Computing Supplementum **6**. Springer-Verlag, Wien / New York, 1988.

[17] Lohner, R.: *Verified computing and programs in PASCAL-XSC*. Universitaet Karlsruhe, Habilitationsschrift, pp 230, 1994.

[18] Mayer, G.: *Reguläre Zerlegungen und der Satz von Stein und Rosenberg für Intervallmatrizen*. Habilitationsschrift, Universität Karlsruhe, 1986.

[19] Mayer, G.: *Enclosing the Solutions of Linear Equations by Interval Iterative Processes*. In [16], pp 47–58, 1988.

[20] Mayer, G., Frommer, A.: *A Multisplitting Method for Verification and Enclosure on a Parallel Computer*. In [31], pp 483–497, 1990.

[21] Mayer, G.: *Old and New Aspects for the Interval Gaussian Algorithm*. In [13], pp 329–349, 1991.

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

[23] Rump, S. M.: *Kleine Fehlerschranken bei Matrixproblemen*. Dissertation, Universität Karlsruhe, 1980.

[24] Rump, S. M.: *Solving Algebraic Problems with High Accuracy*. Habilitationsschrift. In [15], pp 51–120, 1983.

[25] Rump, S. M.: *Estimation of the Sensitivity of Linear and Nonlinear Algebraic Problems*. Linear Algebra and its Applications **153**, pp 1–34, 1991.

[26] Rump, S. M.: *Convergence Properties of Iterations Using Sets*. In [10], pp 427–431, 1991.

[27] Stewart, G. H.: *Introduction to Matrix Computations*. Academic Press, New York, 1973.

[28] Stoer, J.; Bulirsch, R.: *Introduction to Numerical Analysis*. Springer-Verlag, New York, 1980.

[29] Ullrich, Ch., Wolff v. Gudenberg, J. (Eds.): *Accurate Numerical Algorithms, A Collection of DIAMOND Research Papers*. Springer-Verlag, Berlin, 1989.

[30] Ullrich, Ch. (Ed.): *Computer Arithmetic and Self-Validating Numerical Methods*. (Proceedings of SCAN-89, invited papers). Academic Press, San Diego, 1990.

[31] Ullrich, Ch.: *Programming Languages for Enclosure Methods*. In [30], pp 115–136, 1990.

[32] Wilkinson, J.: *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1964.