Design of Generic Direct Sparse Linear System Solver in C++ for Power System Analysis

Shubha Pandit, Student Member, IEEE, S. A. Soman, and S. A. Khaparde, Senior Member, IEEE

Abstract—This paper presents design of generic Linear System Solver (LSS) for a class of large sparse symmetric matrices over real and complex numbers. These matrices correspond to either of the following 1) Symmetric Positive Definite (SPD) matrices, 2) Complex Hermitian matrices, 3) Complex matrices with SPD real and imaginary matrices. Such matrices arise in various power system analysis applications like load flow analysis and short circuit analysis. Template facility of C++ is used to write a generic program on float, double and complex data types. Design of algorithm guarantees numerical stability and efficient sparsity implementation. A reusable class SET is defined to cater to graph theoretic computations. LSS problems with matrices up to 20,000 nodes have been tested. Another feature of the proposed LSS is implementation of associative array, which allows subscribing an array with character strings, such as bus names. This helps in making the power system analysis software user friendly. The proposed LSS reflects an important development toward a truly object oriented power system analysis software.

Index Terms—Graph theoretic applications, linear system solver, LU decomposition, object oriented programming, sparse matrix computations.

I. INTRODUCTION

OBJECT Oriented Programming (OOP) has been accepted by the power industry as a viable alternative to traditional procedural programming. In all these developments, C++ has been almost the universal choice for implementation. An OOP language like C++ exhibits following features which are not supported by procedural languages [1].

• Definition of classes and their methods: This involves designing classes by encapsulating data and methods. Neyer et al. [2] discuss definition of classes for power system apparatus like lines and transformers.

• Operator overloading: Methods for arithmetic and logical operations such as addition, subtraction, etc. on user defined classes like matrix, vector can be implemented by overloading the corresponding arithmetic operators.

• Encapsulation: Methods written in other languages like FORTRAN can be encapsulated in C++, thereby reducing the software development time. For example, FORTRAN NAG library routines have been encapsulated in C++ for implementation of load flow analysis in reference [3]. However, such an implementation lacks flexibility and may not correspond to a good OO implementation. Defining reusable classes and methods is necessary to take full advantage of OO paradigm. For example, it may be advisable to write OO software for library functions like LSS, matrix computations, etc.

• Inheritance: In C++, a generic base class can be defined on which more specific application classes can be derived. Zhou [3] has defined a network container base class which contains classes for power system apparatus. Virtual functions are also defined on the base class. AC load flow is then a derived class which has load flow as a method. While derived classes and use of virtual functions improve flexibility of a large program, it may result in loss of speed as choice of implementation is done at run time rather than at compile time.

• Use of template facility: In C++, templates provide another level of generalization. In other words, a unique code which caters to many data types can be written by using template facility [3]. For example, a class matrix may be required to support data types such as float, double and complex; complex being a user defined data type available in standard library. This means that all methods like arithmetic operations defined for this class should work equally well on all these data types, without duplication of code. The algorithm written using template definition for class matrix is therefore written with data type T. User program then specifies required data type. Use of templates add flexibility at the cost of compile time. Run time efficiency, however, is not sacrificed.

OOP for sparse matrix computations is discussed in reference [4]. Reference [5] identifies two important computational classes, one for sparse matrix computations and the other for graph theoretic computations. Recently, OO library developed for sparse matrix computations has received significant attention of numerical analysts and researchers in scientific computations. To quote Dongarra et al. [6], “besides simplifying the subroutine interface, the OO design allows driving code to be used for various sparse matrix formats, thus addressing many of the difficulties encountered with the typical approach to sparse matrix libraries. Using C++ for numerical computation can greatly enhance clarity, reuse, and portability.” Another important area of research is development of library for ordering sparse matrices. Such library permits user to choose ordering methodology depending upon problem requirements. However, our experience suggests that learning curve involved in such work may be steep. Dobrian et al. [7] discuss design of sparse direct solvers using OO technique. The implementation is in C++. They identify design goals as managing complexity, simplicity of integration, flexibility, extensible, reusable and efficient coding. Ashcraft et al. [8] have designed an OO code...
called \textit{SMOOTH} for reduced fill ordering. An OO package of direct and iterative solvers, called \textit{SPOOLES}, has been reported in reference [9]. Both these packages are implemented in C.

In this paper, we discuss design and implementation of sparse LSS in C++, for computations in power system applications. This provides a focused development of methods that meet requirements such as speed, stability, simplified interface and scalability. Consider implementation of \textit{LDL}$^T$ algorithm for a symmetric sparse matrix. Such matrices arise in power system analysis \textit{e.g.}, real matrices $B, B^o$ in Fast Decoupled Load Flow (FDLF), complex admittance matrix (YBUS) in short circuit analysis. The requirement of basic data type is float (or double) and complex respectively. One of the challenges in designing a good C++ program lies in exploiting the \textit{template} facility for non trivial problems. The question is, \textit{can there be a unique, sparse, efficient implementation of \textit{LDL}$^T$ algorithm using this feature over the said data types?} An efficient implementation implies a numerically stable implementation with minimum computation complexity and a memory requirement proportional to the number of nonzeros in the original matrix. In fact, it is difficult to write an efficient unique \textit{LDL}$^T$ factorize routine for a general class of symmetric matrices over real numbers as the implementation issues associated with SPD matrices differ a lot from indefinite symmetric matrices. However, by restricting the class of matrices to either SPD or symmetric indefinite, efficient implementation can be developed.

This paper proposes design of \textit{LDL}$^T$ decomposition based LSS using template, for a class of real and complex matrices. The class of matrices considered need to have symmetry and diagonal dominance. Since YBUS matrix and matrices usually derived from it are diagonally dominant, the proposed implementation is best suited for power system applications like load flow and short circuit analysis. Another advantage of the proposed solver is that it works directly with bus names as subscripts, thereby making user free from assigning bus numbers for each bus.

The paper is organized as follows. Sections II and III discuss the issues regarding LU decomposition of real and complex matrices respectively. Implementation of Analyze and Factorize phases is developed in Sections IV and V. While a class \textit{sparse matrix} is defined in Section VI, Section VII relates the character strings to integers using the associative arrays. Results are compiled in Section VIII. Section IX concludes the discussion.

II. LU DECOMPOSITION WITH REAL MATRICES

Consider direct solution of a sparse linear system $Ax = b$ where, $A$ is a real symmetric matrix and $b$ is a real vector. We restrict the discussion to LU decomposition based solvers. The matrices arising in LSS can be grouped into following two categories [10]:

$R_1$: when matrix $A$ is real sparse SPD.

$R_2$: when matrix $A$ is sparse indefinite.

Typically, matrices in FDLF, normal equation approach for power system state estimation belong to first category, while matrices in Hatchel’s methods for state estimation, etc. belong to the later. If application algorithms are so chosen that they lead to sparse LSS with real sparse SPD matrices, significant computational savings can be achieved.

When the matrix $A$ is indefinite, the choice of pivot has to be undertaken considering numerical magnitude (size) of pivot and the number of fills that will result. Thus, for this category, choice of pivot has to be determined during numerical factorization and therefore fills can not be predicted \textit{apriori}. This implies use of dynamic data structure (preferred one being linked list). Use of dynamic data structure tends to distribute data in memory instead of storing it in a continuous block. This involves, extra cache fetches, leading to problems like page thrashing, which together slow down the process of implementation. Bunch \textit{et al.} [11] discuss various complete and partial pivoting strategies.

In contrast, for SPD matrices, it can be shown that diagonal pivots are always stable. An important consequence is that diagonal pivots can be selected \textit{apriori} to actual factorization. Minimum Degree Algorithm (MDA) [12] is one of the most commonly used strategy for ordering nodes so as to reduce/minimize fills. A static data structure can then be set up in a continuous block of memory, \textit{apriori}, to numerical or actual factorization. Typically, this is known as \textit{Analyze phase}. \textit{Factorize phase} corresponds to actual numerical factorization.

III. LINEAR SYSTEM SOLVER WITH COMPLEX MATRICES

Let $A$ be a complex matrix given by $A = B + iC$. We can group $A$ into following three categories:

$C_1$: $A$ is a complex symmetric $n \times n$ matrix with matrices $B$ and $C$ real SPD;

$C_2$: $A$ is a complex Hermitian matrix;

$C_3$: $A$ does not belong to category $C_1$ and $C_2$ above.

If a matrix is complex Hermitian (category $C_2$), it is known that Cholesky factorization can be performed. Because of diagonal dominance of partially factored matrices, diagonal pivots are stable. However, complex symmetric matrices ($A = A^T$) do not admit to such simple methods, an example being YBUS (category $C_2$). Consider the admittance model formulation for short circuit analysis:

$$[I] = [YBUS][V].$$

One way to solve equation (1), though not elegant, is to convert the problem into a real LSS problem as follows:

$$\begin{bmatrix} \hat{\varphi} \\ \hat{i} \end{bmatrix} = \begin{bmatrix} G & -B \\ B & G \end{bmatrix} \begin{bmatrix} \hat{\varphi} \\ \hat{i} \end{bmatrix}$$

(Superscript $\tau$ and $i$ refer to the real and imaginary parts.)

It is known that such a scheme can be inefficient in storage and time [13]. One of the best available schemes to factorize symmetric complex matrix is a version of Bunch's diagonal pivoting method for symmetric indefinite LSS [11]. The essential step at each stage of the elimination is to choose $1 \times 1$ or $2 \times 2$ blocks for pivoting. This choice is dictated by magnitude of the pivot which affects numerical stability of implementation.

S. M. Serbin [14] showed that for category $C_1$ matrix, LINPACK diagonal pivoting decomposition $U^{-1}AX^{-T}$ proceeds without necessity for pivoting. The analysis also holds for $A =$
VT decomposition. Though this reference did not deal with sparse LSS, we can summarize the following:

- The VT algorithm for sparse SPD LSS can be used by replacing data type float to data type complex, without affecting numerical stability.
- For matrices of category \( R_2 \) and \( C_3 \), a VT algorithm may be implemented based on Bunch’s partial pivoting strategy.

This paper discusses design and implementation of a generic sparse LSS, using the template facility in C++, over the following category of matrices:

1) SPD real matrices (category \( R_1 \))
2) complex symmetric matrix with real and imaginary component as SPD matrices (category \( C_1 \))
3) complex Hermitian matrices (category \( C_2 \)).

The section next discusses the design of Analyze phase.

### IV. Analyze Phase

Implementation of Analyze phase involves ordering to reduce fills (implemented through MDA) and setting up of data structure for sparse matrix. The data structure is developed by an algorithm described in [10].

**Implementation of MDA**: Being one of the most successful ordering algorithms for SPD matrices, various enhancements to speed up MDA have been suggested in literature. While designing efficient implementation of MDA, following design goals have been specified.

1) As MDA involves graph theoretic operations, class and methods designed for MDA should be useful in other graph theoretic operations (like finding connectivity of graph, tree, loop etc.), which are involved in the observability analysis and relay coordination of mesh systems.
2) Graph theoretic applications including MDA routinely require finding one or more nodes, edges, union of nodes/edges etc. Implementation of operations like intersection, union, should be optimal, both in time as well as space. The implementation should provide an elegant interface to user.

The discussion so far clearly suggests suitability of a class set with methods for intersection, union etc. defined on it. The use of class set to model topological relationships/graph traversing was suggested in [2]. Class set is also implemented in the standard template library, as a generalized class map which serves various applications like dictionary, telephone directory etc. [1]. However, the proposed class Set is tailored only for sparse graph theoretic operations required in sparse matrix computations. Object oriented technologies are leveraged through reuse. Class Set can be reused in applications like Network Topology Processing.

**Introduction to Class Set**: Class Set is defined as follows:

```cpp
class Set {
    int *v, cardin, vertex, vsize;
};
```

A graph of \( n \) vertices can be represented by collection of \( n \) sets. Each set for a vertex contains adjacent vertices to it. For convenience, the vertex is also included as a member of the set. Set operations like union, intersection, difference and logical operations are implemented by overloading operators as shown in Table I.

**Optimal Implementation of Class Set**: A set, as per definition, is a collection of distinct elements. From the point of view of implementation, set can be classified as Unordered set, Bit set or Ordered set [15].

**Unordered Set**: A set is said to be an unordered set when elements in the set do not appear in any particular order. The union operation requires an effort of \( O(s1.cardin \times s2.cardin) \) which is much more than linear effort. Therefore, it is not advisable to use unordered set.

**Bit Set**: Let the vertices of original graph \( G \) be represented by set \( V; V = \{1, 2, \ldots, n\} \). All sets involved in a graph theoretic implementation are subsets of \( V \). Such sets can be represented by a collection of \( n \) bits. If vertex \( v \) is present in a set, then, bit \( v \) is set to 1 else, it is set to 0. The advantage of bit set representation is that set operations like union operation can correspond to OR operation, difference to an exclusive OR and intersection to the AND operation on bit sets. C++ supports bit operations on a word length. The bit set implementation is very convenient when the order of \( n \) is small, typically \( n < 32, 64 \). However, when \( n \) is large, say 10,000, bit set implementation requires \( n \) integers to replicate \( n \) bits. Thus, memory requirement becomes prohibitive. As such bit set implementation is not very efficient for large scale systems.

**Ordered Set**: A set is said to be ordered, when elements of the set are arranged in ascending (or descending) order. Here union operation can be performed in \( O(s1.cardin + s2.cardin) \), typically much less than \( O(n) \) effort for bit set implementation. This justifies use of ordered sets for large scale problems.

**Intersection in Ordered Sets**: Two efficient implementations for intersection are possible with ordered sets. One of the implementations would involve an algorithm similar to the union, the complexity of which would be \( O(s1.cardin + s2.cardin) \). However, a single find operation to search an element in a set \( S \) can be performed in \( O[log(S.cardin)] \) using the divide and conquer strategy. In this strategy, a set \( S \) is partitioned into ordered subsets with cardinality \( S.cardin/2 \). Only one of two subsets can have the element being searched. Recursive application leads to the solution. At each step of recursion, the effort involved to solve the problem is halved; hence, the logarithmic order of complexity. Intersection of two sets \( s1 \) and \( s2 \) can be computed, using this technique, with complexity of \( O(s1.cardin \times [log
When $sL_{cardin}$ is much less than $s2.L_{cardin}$ it is preferable to use divide and conquer algorithm.

The factorize and solve phases are discussed in brief in the next section.

V. FACTORIZE PHASE

LU factorization involves following computations

\[
L_{i,j} = A_{i,j} - \sum_{k=j+1}^{\infty} L_{i,k} * L_{j,k}
\]

\[
D_{i,i} = A_{i,i} - \sum_{k=1}^{i-1} L_{i,k} * L_{i,k}
\]

With user defined data type complex and overloaded arithmetic and logical operators, a generic code can be written for factorize phase using template facility. Such an implementation is numerically stable for matrices mentioned in Section III. The following section discusses the implementation of a class sparse\_matrix in which the methods like MDA, factorize and solve(forward-backward substitution) are implemented.

VI. IMPLEMENTATION OF CLASS sparse\_matrix

Data Structure: The memory of a large sparse system can be optimized by storing only the nonzero elements of the matrix. In order to minimize searches of elements in the matrix, a data structure spmat which allows creation of a static linked list is defined. The link\_row and link\_col contain the index to the next element in a row or column in a matrix.

\[
\text{template}(\text{class} T) \text{struct spmat} \{
\text{int row.col, link.row, link.col;}
\text{Tvalue;}
\};
\]

The class sparse\_matrix can be defined as follows:

\[
\text{template}(\text{class} T) \text{class sparse\_matrix} \{
\text{protected:}
\text{int rows.cols.nz, *row.begin, *col.begin;}
\text{spmat(T)* A;}
\text{public:}
\text{void sort(); // establish links}
\text{int * mda(); // ordering by MDA}
\text{void permute(); // renumber the matrix}
\text{void factor(); // numerical factorization}
\text{friend void solver(sparse\_matrix}(T) & S,}
\text{vector(T) & V); // forwardbackward substitution}
\};
\]

The following section discusses mechanism to use character strings as array subscripts instead of integers.

VII. IMPLEMENTING ASSOCIATIVE ARRAY

In the graphs associated with power systems, buses or nodes have physical significance. Typically, reference by name reflects this information, e.g., name of the place, KV class, etc. In contrast, bus numbers do not convey any information. While it was not possible to have character subscript in languages like FORTRAN and C, it is possible to implement this feature in C++ by defining an associative array and overloading the operator [ ].

We define a data structure pair to implement class assso\_array. The struct pair stores bus name as a character string and bus number as an integer. Given a bus name as the key, the corresponding bus number is made available by methods defined on class assso\_array. In simplest sense, an associative array defines a map between the character string and integer. Use of associative array makes numerical computations more friendly.

VIII. RESULTS

The results presented are obtained on Intel Pentium Pro Processor at 200 MHz with 32 MB RAM running a Linux kernel (version 2.0.34). All programs are compiled using a gnu C++ compiler (version egcs 1.0.2-8), using the third level of optimization (O3).

The timing reported for solution of a linear system includes:

Analyze Phase:
1) Setting up a static linked list using sort routine.
2) Extracting sparse matrix connectivity data from step 1 to create SET representation.
3) New ordering using MDA.
4) Setting up LU data-structure (symbolic factorization).

Numerical Computation Phase:
1) Numerical factorization.
2) Forward backward substitution.

All the overheads accompanying class SET, sparse\_matrix and assso\_array have been accounted for. The proposed MDA implementation has been used successfully for ordering nodes in power systems upto 1044 nodes. Computation time for Fast Decoupled Load Flow and short circuit analysis, on these systems, using the proposed LSS, are presented in Table II.
In addition, more than hundred large sparse matrices were generated using random sparse matrix generator available on MATLAB. The test systems correspond to an average connectivity of 7 per node. Table III compares timing (in msec) of MDA on these systems, where class SET is implemented by bit set (column 4) and ordered set (columns 5 to 7). The \( \text{div}_\text{con} \) stands for implementation of operators \( \% = \) and \( \% = \) using divide and conquer strategy. Similarly, without \( \text{div}_\text{con} \) refers to an implementation with linear worst case complexity. Results using both these strategies are compared in Table III, columns 6 and 7. The results clearly show that complexity of bit set implementation is worse than ordered set (refer Section IV). It can be seen that the best implementation of MDA is obtained using a) ordered set, b) with divide and conquer strategy for intersection and difference operation, c) returning variables by reference.

Table IV shows breakup of total solution time in msec, required for implementing MDA, symbolic and numerical factorization on test systems of Table III. Results for the same test systems with a standard implementation \textit{SPARSPAK} available on \textit{netlib} are presented in Table V. FORTRAN subroutine \textit{GENQMD} of \textit{SPARSPAK} uses quotient graph model with enhancements like indistinguishable nodes for ordering the nodes. Function \textit{SMBFCT} for symbolic factorization, \textit{GSFCT} for numerical factorization and \textit{GSSL} for forward backward substitution have been implemented in this package.

Table VI summarizes the performance using \textit{SPOOLES} package [9]. This package is designed to solve two types of real or complex LSS. Driver program \textit{allInOne.c} uses default parameters for multiple elimination of vertices with minimum priority, with exact external degree for each vertex. Only two adjacent nodes are used for graph compression required in finding indistinguishable vertices. Pivoting is not used. The package has an OO design but is implemented in C. Therefore, computations with this package do not involve overheads inherent with OOP languages like C++. Even though \textit{struct} is used for encapsulating data, designing simple interfaces, creating multiple objects, developing polymorphic code, using templates, etc. is not straightforward. All the same, it has utility in objectively evaluating efficacy of proposed implementation.

Table VII compares performance of the three packages on the 20,000 node test system, which is illustrative of the trends seen in Tables IV–VI. It is observed that:

1) In Analyze phase, C based \textit{SPOOLES} implementation gives optimal results. Performance (time and fills) of the proposed implementation is in between \textit{SPOOLES} and \textit{SPARSPAK}. \textit{SPOOLES} incorporates the best quality of ordering reflecting the ongoing progress in ordering technology.

2) FORTRAN based \textit{SPARSPAK} implementation provides the best results in Numerical Computation phase. It can be observed that, \textit{SPOOLES} implementation has a very slow solver. In \textit{SPOOLES}, factorization and solve phases are based on front matrices. As direct factorization is implemented, dense fronts are stored. Proposed implementation ranks next to \textit{SPARSPAK} implementation in this phase.

3) Overall performance of the proposed implementation is superior to \textit{SPARSPAK} and \textit{SPOOLES}. It can be seen that, total computation time is the least for the proposed implementation. \textit{Hence, we conclude that the proposed OO design and C++ implementation is competitive!}
IX. CONCLUSION

In reply to the question raised in Section I, we state that it is possible to have a unique efficient implementation of $LDLT^T$ factorization for a class of SPD and complex SPD like matrices. While it is difficult to numerically verify symmetric positive definite nature of matrices in power system analysis, YBUS and matrices in FDLF exhibit diagonal dominance. The proposed class sparse_matrix can be augmented with methods for sparse QR decomposition wherein MDA function can be reused. An important conclusion is that, this OO implementation is competitive enough to a procedural implementation. Some noteworthy OO design and C++ implementation contributions are:

1) Design of class SET to facilitate and simplify MDA implementation. The class SET is reusable in various graph theoretic applications like observability analysis and relay coordination in mesh systems.

2) A carefully crafted C++ implementation using return by reference, templates and operator overloading. This enables speed as well as simplifies interface and programming. Without any run time penalty, templates integrate factorize and solve phases of LSS over a class of real and complex matrices.

3) Matching of matrices in power systems with sparse LSS technology (data structure-algorithms-numerical analysis). Successful usage of the proposed LSS in power system application is therefore justified.

REFERENCES


Shubha Pandit is Assistant Professor at S.P. College of Engineering, Mumbai. She is working for Ph.D. at I.I.T. Bombay.

S. A. Soman is Assistant Professor at I.I.T. Bombay. His research interest include sparse matrix computations, power system analysis, OOP, and power system automation.

S. A. Khaparde is Professor at I.I.T. Bombay. His research interests include power system computations, analysis and deregulation.