Object-oriented Programming, C++ and Power System Simulation

E.Z. Zhou, MIEEE
EDSA Micro Corp.
200 East Long Lake Rd., Suite 177
Bloomfield Hills, MI 48302

***Abstract*—Application of OOP concepts and the C++ programming language to power system simulation has been explored. An efficient platform for power system simulation applications has been proposed. By emulating a physical power lab, a generic power network container has been build. A module, which can be used to solve sparse matrix equations of any data types, has been implemented. Two load flow applications: a dc load flow and an ac load flow have been discussed. It is found that the OOP based C++ load flow programs are as efficient as their non-OOP counterparts.**

1. Introduction

The planning, design and operation of electrical power systems require simulation analyses to evaluate the current and future system performance, reliability, safety, and ability to grow with production or operating requirements. To perform computer simulation analysis, a power system is represented by a set of data with certain structures. Analysis algorithms are implemented to process the data to produce simulation results. For different applications the data structures are different. Different programs (load flow, short circuit, stability...) are built for different applications. With conventional programming languages (Fortran, C, Pascal) the coupling between the data structures and the algorithm procedures are very strong. A minor change may propagate through a whole program[1]. As a result, software modification and evolution may require a time period proportional to the size of the software rather than the magnitude of the changes.

There are serious problems with today's approaches to build Energy Management Systems (EMS)[2]. The present EMS architecture is inflexible. It precludes a logical and phased replacement of its subsystems and usually ends up with an entire replacement in about ten years[3]. The solution to the problem is to use open system architectures. Standards are used and key interfaces are disclosed to enable incremental growth. Multiple suppliers are available for various upgrades. The utility can select the best hardware and software for each phase of the upgrade. One of the key features of an open EMS is modular compatible application software[11]. Third parties will be able to offer new application software that function with the existing applications. One promising approach to achieve the modularity is to use object-oriented programming (OOP) approaches which are designed to address the problems associated with large-scale software developments.

The potential of OOP for power system simulation applications has been well recognized. There are OOP applications to database management[4], graphical user interface[5] and power system simulation[1]. So far these applications are based on some existing platforms (class library and programming environment). On top of a well-developed platform one can easily build his application programs through inheriting the features in the library. One shortcoming of this approach is that one has to tailor his problem to be fitted into the existing platform. This may sometimes result in a very complicated, hard-to-understanding representation of an originally simple concept. For example transmission lines are represented as a Line class, a subclass of class NPort, which is then a subclass of class Physical in ref.[1]. The complicated structure will often create heavy overhead. The program runs slower because of large amount of unnecessary message passing between objects and requires more memory. A test case in ref.[1] indicates that about 40% of the total CPU time was spent on message passing.

Application of OOP concepts and the C++ programming language[6] to power system simulation will be explored in this paper. An efficient platform for power system simulation applications will be developed by using C++ as a programming language (not as an existing platform). The author believes that platforms for power system simulation applications should be created by power engineers. A brief review of C++ is presented in Section-2. By emulating a physical power lab, a generic power network container is build in Section-3. In Section-4 a sparse equation module is introduced which can be used to solve sparse equations of any data structures. Load flow application problems are discussed in Section-5. The efficiency of C++ implementation of an ac load flow program and a dc load flow program is compared with a Fortran version and a Pascal version in Sec.6.

2. A Review of C++

C++ is a very popular and efficient OOP language. Some of important C++ features will be briefly discussed in this section.

2.1 Template

One of the most important improvements of OOP approaches over non-OOP approaches is the flexibility of reusing existing code. Beginning with the AT&T 3.0 Standard, C++ offers a feature, called template, that extends code reuse even further. Let us use the following simple example to show the usefulness of the template feature. The Ohm's law in power system analysis can be expressed as follows:

$$V = Z \times I \qquad (1)$$

where, Z is the impedance of a device, I the current through the device and V the voltage drop. For dc power distribution system analysis, V, I, Z are real numbers. For ac system analysis, V, Z, I are complex numbers. If it is for 3-phase unbalance situation analysis by using the *abc* coordinates[9], V, I would be 3×1 complex vectors and Z 3×3 complex matrix. With the introduction of template, it is possible to write a generic function to cover all possible Ohm's law applications, as follows:

```
template<class Tz, class Tvi>
  Tvi voltage( Tz z, Tvi i )
       { return z*i; }
```

The template statement basically tells C++ compiler that `Tz` and `Tvi` represent any data types. The compiler will generate code for as many different functions as it needs to satisfy the calls made to the `voltage` function.

2.2 User-defined Data Types

For the analysis of small disturbances (changes) around a steady-state operating point, eqn(1) in complex form would become:

$$\begin{vmatrix} \Delta V_x \\ \Delta V_y \end{vmatrix} = \begin{vmatrix} r & -x \\ x & r \end{vmatrix} \begin{vmatrix} \Delta I_x \\ \Delta I_y \end{vmatrix} \qquad (2)$$

C++ allows the users to define data types that are most suitable for their own applications. To represent the relationship of eqn(2) the following two data types are defined:

```
class Vect_xy {
        Double x, y;
        ...
      };
class Matr_xy {
        double xx, xy, yx, yy;
        ...
      };
```

After the data types `Vect_xy`, `Matr_xy` are properly defined and implemented, the `voltage` function can be called to compute the voltage vector in eqn(2).

2.3 Operator Overloading

When the `voltage` function is called to calculate eqn(2), a multiplication of a 2×2 matrix (`Matr_xy`) and a 2×1 vector (`Vect_xy`) is performed inside the function. This is possible because C++ allows the math operators (+,-,×,/...) being overloaded, called operator overloading. With the operators properly overloaded, user-defined data types in C++ could be used as if they were compiler build-in data types (int ...).

2.4 Inline Functions

One of the criticisms that is often addressed to OOP languages is that they are much less efficient than the traditional non-OOP languages. This is unquestionably true for some OOP languages. The load flow program implemented in Objective-C is about 2-3 times slower than a Fortran implementation[1]. But this is far less apparent in the case of C++. In fact, efficiency is one of the goals stated by the inventor of C++[6]. For example, one of the properties that can make an OOP program slower than a non-OOP program is the heavy use of data access functions, which is a direct consequence of the application of the information-hiding principle. C++ lets you make inline substitutions of functions, which means that every function call is actually replaced at compiling by the function's executable code to avoid the overhead due to the actual calling of the function.

3. A Network Container

Before the introduction of computers to power system simulation, the analysis was usually done on an ac network analyzer in a power lab. A power lab is conceptually a place (a house) where there are positions that can be defined as buses. Branch elements (lines, transformers) can be connected to the buses to form a power network. A power lab is designed and built to be generic so that different kinds of experiments (load flow, short circuit, ...) can be performed. To emulate this concept, a network container will be designed in this section. The design objective is that the container should be generic, flexible and reusable for different kinds of applications.

For identification purpose, a unique name or id needs to be assigned to each element in a network. To serve this purpose the following data type is defined.

```
class NameTag {
    char* id;         // name id
    int   noSort;     // number for sorting
    ...
  };
```

The `noSort` is a number for element sorting purpose.

3.1 Bus Class

In a power lab, a bus is a position where a branch or branches can be connected to. To emulate this idea, the following data type is defined as a base bus class.

```
class Bus : public NameTag  {
    ListOfPtr<NameTag> branchConnected;
      ...
  };
```

`Bus` is a subclass of `NameTag`, inheriting all its features. `Bus` has an `id` for identification purpose, a number `noSort` for sorting purpose and a list of pointers, pointing to the branches connected to the bus. Any number of branches can be con-

nected to a bus object (an instance of `Bus`) by adding pointers to the branches into the `branchConnected` list.

3.2 Branch Class

A branch is a device with two terminals (from terminal and to terminal) which can be connected between two buses (from-bus and to-bus). The following data type is defined as a base branch class:

```
template<class TBus>  // TBus: bus template
  class Branch : public NameTag {
    TBus*  ptrFromBus;  // ptr to from bus
    TBus*  ptrToBus;    // ptr to to bus
      ...
  };
```

`Branch` has two pointers: one points to the from-bus and the other points to the to-bus. `TBus` is a template for the bus objects, to which the branch objects (instances of `Branch`) will be connected.

3.3 A Network Container

A network container for the simulation purpose should be designed as if it is a place where buses can be defined, and branches can be connected between the defined buses to form a network. The following data type is defined as a base network class, a generic network container.

```
template<class TBus,class TBra>
  class Network : public NameTag {
    LinkedList<TBus>  busList;
    LinkedList<TBra>  branchList;
  public:
    void  addBus( ... );
    TBus& getBus( ... );
    void  addBranch( ... );
    TBra& getBranch( ... );
    void  resortBus( ... );
    void  resortBranch( ... );
    void  arrangeBusno( ... );
    ...
  };
```

At the center of the container there are two linked lists, one for storing bus objects and the other for branch objects. The `LinkedList<T>`, a linked list class of data type `T`[13], knows how to manage memory dynamically. With the help of the templates `TBus, TBra`, any kinds of bus and branch objects can be stored into the container. Network utility functions (`arrangeBusno(), resortBus(), ...`) are implemented in the container which will be available through inheritance to all network application programs. The `network` can arrange bus number of the buses in the container according to one of the rules: Tinney1, Tinney2 or Tinney3[7].

The following is a simple example of storing information into and then retrieving from the container.

```
Network<Bus, Branch> myNet;
Bus    myBus1, myBus2;
Branch myBranch;
  ...
myNet.addBus( 'bus1', myBus1 );
myNet.addBus( 'bus2', myBus2 );
myNet.addBranch('bus1', 'bus2', myBranch);
  ...
Bus& aBus=myNet.getBus('bus1');
Branch& aLine=myNet.getBranch('bus1','bus2');
  ...
myNet.resortBus(By_id);
myNet.arrangeBusno(Tinney2);
```

The `Bus` class, the `Branch` class and the `Network` container so far are very simple and cannot do any real simulation work, just like an empty power lab. They have a few key features common to, and will serve as core code for, all power network simulations. They can be easily maintained because of the simplicity. We will see in later sections how these simple modules can be combined with other modules to solve load flow problems. This is the evolutionary way of the OOP approach.

4. Sparse Equation Solution

At the center of power network solution is the solution of the following sparse matrix equation:

$$[A] \times [x] = [B] \qquad (3)$$

where, [A] is a nonsingular sparse matrix, [B] is a given vector and [x] is an unknown vector to be found. Sparse matrix method[7] are now used for solving almost all large power network problems. Although the data types of eqn(3) for different applications may be different, the logic of LU factorization of [A], the forward and backward substitution processes of solving eqn(3) are independent of its data types. The following is a generic class for solving sparse equations in terms of two templates `TAij, TBi`.

```
template<class TAij,class TBi>
  class SparseEqn {
     SortedList<A_Elem<TAij>> Amatrix;
  public:
     void setBvector( ... );
     void addAmatrix( ... );
     int  solveEquation( ... );
      ...
  };
```

Only non-zero elements of [A] are stored in the list `Amatrix`. The `SortedList<T>` is a sorted linked list class of data type `T`[13]. The sorting is according to element relative positions in [A]. Also the LU factors are stored in the list, overriding the [A] matrix to reduce memory requirements. With the help of templates `TAij, TBi, SparseEqn` can be used to solve sparse matrix equations of any data types. The following is a simple example:

$$\begin{vmatrix} 2.5 & 0.5 & 0 & 0 \\ 0.5 & 0.6 & 0 & 0 \\ 0 & 0 & 3.1 & 1.7 \\ 0 & 0 & 0 & 0.7 \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{vmatrix} = \begin{vmatrix} 1.5 \\ 0.6 \\ 3.5 \\ 2.7 \end{vmatrix} \qquad (4)$$

If the [A] is treated as a 4×4 matrix of real numbers and [B] a 4×1 vector, the problem can be solved as follows:

```
SparseEqn<double,double>
             myEqn1(4,NonSymmetric);
myEqn1.setBvector( ... );
  ...
```

```
myEqn1.addAmatrix( ... );
  ...
myEqn1.solveEquation( 1.e-30 );
```

Also the [A] can be partitioned into a 2×2 matrix (of 2×2 elements) and [B] into a 2×1 vector. By using the data types `Matr_xy, Vect_xy` defined in Section-2, the problem of eqn(4) can be alternatively solved as follows:

```
SparseEqn<Matr_xy,Vect_xy>
            myEqn2( 2, Symmetric );
  ...
myEqn2.solveEquation( 1.e-30 );
```

One would assume that the second 2×2 matrix approach might be slower than the first 4×4 matrix approach because the user defined data types (OOP feature) are involved in the 2×2 approach. The following is the CPU time used in solving eqn(4) by the two approaches for 1000 times on a 486DX/33Mz PC:

4×4 approach: 0.55 sec. 2×2 approach: 0.44 sec.

The performance tests indicate that the 2×2 approach is even faster than the 4×4 approach. The main reason is that all member functions of `Matr_xy, Vect_xy` are declared as inline functions. This makes them as efficient as the compiler build-in data types.

The key points of `SparseEqn` are modularity and flexibility. Sparse matrix equations of any data types can be solved by using the module. It is a self-contained module, managing its memory dynamically. To the user it is a "black box" with the sparse matrix solution capability. The sparse matrix method is now under intensive investigation[10,12]. If any future modifications or improvements are needed the modifications can be contained to `SparseEqn` itself and would not propagate to the outside.

5. Load Flow Applications

Load flow is one of the most commonly use power system analyses. From computation point of view, the problem can be simply stated as follows:

> For specified voltage (V, θ and/or generation (P_g, Q_g) at generator buses and specified load (P_l,Q_l) at load buses, find a set of bus voltage by iteration such that the mismatch between specified quantities and calculated quantities is within a required tolerance.

The convergence of a load flow study is tested by the following equation:

$$\max_{1\le i\le n} \left| \text{ mismatch at bus \# i } \right| \le \text{tolerance} \qquad (5)$$

The mismatch is computed based on bus Y-matrix and bus voltage.

So far, a `Bus` class, a `Branch` class and a `Network` container have been implemented. Also there is a `SparseEqn` class for sparse matrix equation solutions. In this section these modules will be combined together with some new modules to solve load flow problems.

5.1 Load Flow Base Classes

For different load flow applications (dc load flow, ac load flow ...) the details of implementations, especially the data structures, might be quite different. But all load flows share some common features. These common features are extracted and implemented in the following three load flow base classes: `LFBus, LFBranch, LFNet`.

```
template<class T>
  class LFBus : public Bus {
    T voltage;       // bus voltage
  public:
    virtual T yii( void )      = 0;
    virtual T mismatch( void ) = 0;
     ...
  };
```

`LFBus` is a subclass of `Bus`, inheriting all its features. The `voltage` is for bus voltage. The `voltage` data type is defined in terms of a template `T` so that it can be used for different applications. The `yii()` function is for finding the diagonal element yii of Y-matrix corresponding to the bus. The `mismatch()` is for bus mismatch calculation.

```
template<class T,class Tbus>
  class LFBranch : public Branch<TBus> {
  public:
    virtual T yij( void ) = 0;
     ...
  };
```

`LFBranch` is a subclass of `Branch`. The `yij()` function is for finding the off-diagonal elements yij corresponding to the branch. `TBus` is the template for the buses to which the branch will be connected.

```
template<class TV,class TM,
   class Tmis, class Tbus, class TBra>
     // TM,TV: for sparse equation
     // TMis : for mismatch information
class LFNet: public Network<TBus,TBra> {
  public:
    virtual TMis maxMismatch(void) = 0;
    virtual void formJmatrix(
              SparseEqn<TM,TV>& eqn) = 0;
    int loadFlow( void );
     ...
  };
```

`LFNet` is a subclass of `Network`. Therefore `LFNet` is also a network container. Bus objects, branch objects and their connection configuration as a network can be put into the container. In addition, the `LFNet` container has load flow application features. The `maxMismatch()` is for finding the maximum bus mismatch and the location where the maximum occurs. The `formJmatrix()` is for forming Jacobian matrix. The J-matrix is stored into the sparse matrix equation object `eqn`. The data types of J-matrix are specified with two templates `TM, TV` because they might be different for different load flow applications.

LFBus, LFBranch, LFNet contain pure virtual functions (indicated by virtual...=0)[13]. These functions are purely virtual in that they do not define any code but act as a pattern of interface for all subclasses. For example, formJmatrix() of LFNet class is a pure virtual function. It tells the subclass of LFNet that, although the details of actual implementation of formJamtrix() might be different for different applications, the interface of the actual functions must be the same as the pure virtual function so that the actual functions can be properly plugged into the system to perform load flow computation.

In the LFNet class a generic Newton-Raphson load flow function loadFlow() is implemented. The main part is as follows:

```
SparseEqn<TM,TV> lfEqn( get_noBus() );
for (int cnt=0;cnt<=maxIteration;cnt++) {
  if ( maxMismatch() <= tolerance ) {
    return CONVERGED;
  } else {
    formJmatrix( lfEqn );
    forEachBusInNet(bus)
      { ... // set B vector }
    lfEqn.solveEquation( 1.0e-20 );
    forEachBusInNet(bus)
      { ... // update bus voltage }
  }
}
return NOT_CONVERGED;
```

The loadFlow() function is generic, only includes those steps common to all load flow applications. The function is implemented in terms of bus, branch, mismatch, Jacobian matrix element templates and the pure virtual functions that define the function interface. Any class that contains one or more pure virtual functions is called an abstract base class[13]. It can be only used as a base class for other classes. No objects of an abstract class can be created. LFBus, LFBranch, LFNet are abstract base classes for load flow applications.

5.2 AC Load Flow

Ac load flow has been the most commonly used load flow analysis program. In ac load flow analysis bus voltage, generation, load, and branch impedance are complex numbers in nature. Based on the load flow base classes LFBus, LFBranch, LFNet, the following three subclasses are derived for ac load flow analysis.

```
class ACLFBus : public
  LFBus<complex> { ... };
```

ACLFBus is a subclass of LFBus with complex substituting the template T for bus data type. Therefore the voltage, yii(), mismatch() which ACLFBus inherits from LFBus are of type complex. The yii(), mismatch() functions defined in LFBus are pure virtual functions. The actual complex version of these functions has to be implemented inside the subclass ACLFBus.

```
class ACLFBra : public
   LFBranch<complex,ACLFBus> { ... };
```

ACLFBra is a subclass of LFBranch with complex substituting the template T for branch data type and ACLFBus substituting TBus, so that ac load flow branch objects will be connected to ac load flow bus objects.

```
class ACLFNet : public
  LFNet< Vect_xy, Matr_xy, AC_misStruct,
         ACLFBus, ACLFBra > {
    void formJmatrix(
       SparseEqn<Matr_xy,Vect_xy>& eqn);
    ...
  };
```

ACLFNet is a subclass of LFNet. The substitutions of the templates indicate that ac load flow bus objets and branch objects are to be put into the ac load flow network container ACLFNet. For ac load flow using the Newton-Raphson method in the rectangular coordinates, the Jacobian matrix elements are 2×2 matrices and voltage increments $[\Delta V_x, \Delta V_y]$ are 2×1 vectors[9]. The data structure is similar to that of eqn(2). Therefore the 2×2 matrix data type Matr_xy, and 2×1 vector data type Vect_xy defined in Section-2 are used for the sparse matrix equation data types.

With the ac load flow classes properly implemented, load flow analysis of an ac power system is quite simple, as follows:

```
ACLFNet myACLFNet;
cin >> myACLFNet;
myACLFNet.arrangeBusno( Tinney2 );
if ( myACLFNet.loadFlow() == CONVERGED )
  cout << myACLFNet;
```

Where, an ac load flow network object (container) myACLFNet is first defined. By the overloaded I/O operator >>, data of a study case is input into the container. Depending on how the operator is overloaded, the data might come from a text file, a binary file, from other simulation process or directly from the screen of a graphic editor. Then the bus number is rearranged by using the arrangeBusno() inherited from the Network class. Load flow is calculated by calling the loadFlow() inherited from the LFNet class. If the load flow converges, the I/O operator << will direct the results to somewhere depending on how the operator << is overloaded.

5.3 DC Load Flow

Direct current (dc) power distribution systems have been extensively used in nuclear power plants and underground transit systems. To find voltage and voltage drop at load buses of a dc distribution system dc load flow studies are needed[14]. For dc system analysis all quantities involved are real numbers. Based on the load flow base classes LFBus, LFBranch, LFNet, the following three subclasses are derived for dc load flow analysis.

```
class DCLFBus : public
   LFBus<double> { ... };

class DCLFBra : public
   LFBranch<double,DCLFBus> { ... };

class DCLFNet : public
   LFNet<double,double,DC_misStruct,
         DCLFBus, DCLFBra > {
```

```
    void formJmatrix(
          SparseEqn<double,double>& eqn);
    ...
};
```

For dc load flow using the Newton-Raphson method, the Jacobian matrix elements and voltage increments ΔV are real numbers. Therefore `double` is used for the sparse matrix equation data types. With the dc load flow classes properly implemented, load flow analysis of a dc power system is as follows:

```
DCLFNet myDCLFNet;
cin >> myDCLFNet;
myDCLFNet.arrangeBusno( Tinney2 );
if ( myDCLFNet.loadFlow() == CONVERGED )
  cout << myDCLFNet;
```

Inheritance is one of the most powerful tools available in OOP. It allows you to abstract common behavior between similar objects into a base class and define derived subclasses, not from scratch, but in terms of the base class without having to rewrite the similar parts every time. The class hierarchy of load flow applications is shown in fig.1. There are four layers in the hierarchy. The first layer is the `NameTag` class. The second layer consists of `Bus`, `Branch` and `Network`, and handles the common features of power networks. The third layer consists of `LFBus`, `LFBranch` and `LFNet`, and adds the common features of load flow studies to the hierarchy. The fourth layer deals with the details of different load flow applications. Fast decoupled load flow method and 3-phase unbalanced load flow are also implemented. Their discussion has been omitted due to the space limitation.

6. Performance Test Results

It is desirable to design a flexible software project that is easy to build and easy to maintain, but it also needs to be concerned that the software must be efficient. For power system simulation software the top priority is their performance. It is very hard to justify to sacrifice efficiency for flexibility[8]. C++ itself is a very efficient programming language. The performance of a C++ application is mainly dependent on the way by which one programs (speaks) the language. An example in Sec.4 indicates that adding OOP features to a routine does not necessary mean slow in speed. In the following the performance of a C++ implementation of the ac load flow (Sec.5.2) is compared with a Fortran load flow program, and a C++ implementation of the dc load flow (Sec.5.3) with a Pascal version.

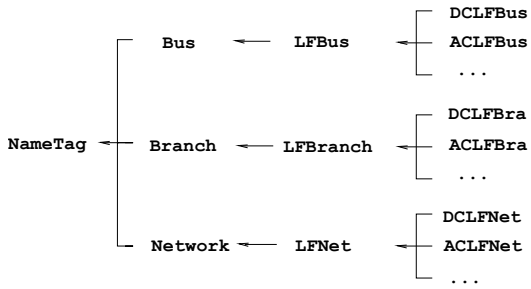

Fig.1 The class hierarchy

Table-1 Performance tests, CPU time in seconds

Cases	Fortran	Pascal	C++
ac 35-bus (6 iter.)	1.53	-	0.66
ac 69-bus (6 iter.)	5.26	-	1.65
dc 100-bus (4 iter.)	-	0.33	0.48(0.38)
ac 69-bus (100 iter.)	51.15	-	27.97
ac 35-bus (100 iter.)	15.33	-	11.70
dc 100-bus (100 iter.)	-	10.55	13.30(11.12)

Two ac power systems (35-bus and 69-bus) and a 100-bus dc system are used to measure the performance. The load flow tests were performed on a 486DX/33Mz PC. The Fortran compiler used is the Lahey Fortran-77 (v3.00), the Pascal compiler: Turbo Pascal (v6.00) and the C++ compiler: Borland C++ (v3.1). The load flows converge in 6 iterations for the two ac systems and in 4 iterations for the dc system. The CPU time used by different programs is shown in Table-1. To measure the "true" numerical performance of the programs the tolerance for convergence was set to 0.0 to make the load flow computation impossible to converge and the maximum iteration was set to 100 so that the computation stops at the 100th iteration.

The CPU time test results in the table indicate that the C++ implementation is slower than the Pascal version and faster than the Fortran version. The reason for the C++ version slower than the Pascal version is that in the Pascal implementation the J-matrix and its LU factors are stored in two separate fixed arrays, while in the C++ implementation the J-matrix and its LU factors are stored in a sorted list with memory dynamically allocated (see Sec.4). The improvement in memory efficiency results in slower in speed. If the same fixed array storage scheme is used, C++ implementation of the dc load flow is as faster as the Pascal version. The CPU time is shown in the brackets in Table-1. The reason why the Fortran implementation of the ac load flow is slower than the C++ version is not clear. It may be because the Fortran compiler used is an old version (1988), which may not be able to take full advantages of a 486 machine.

Memory requirement is also important when the efficiency is concerned. There are mainly three chunks of data: bus data, branch data and LU factor table in the load flow applications. They are stored in the network container and the `SparseEqn` class in three linked lists instead of three fixed arrays. The linked list classes used are from the Borland C++ library[13], that knows how to allocate and free memory dynamically. With the fixed array approach, a statement would always come with a load flow program, for example, this is a 2000-bus version. With the linked list approach, a program only uses an amount of memory necessary to simulation a particular power network. If the program is running under Windows or Dos protected mode[13] one could practically simulate power networks of any size.

7. Conclusions and Comments

Application of OOP concepts and the C++ programming language to power system simulation has been thoroughly explored. An efficient platform for power system simulation applications has been proposed. It is found that the OOP approaches and C++ are very flexible and highly efficient, as efficient as non-OOP programming languages if properly implemented.

The most difficult part of power system simulation programming is to deal with the network relationship. By emulating a physical power lab, a generic power network container has been build. Any types of bus objects can be put into and retrieved from the container, and any kinds of branch objects with two terminals can be connected between the buses in the container. The network container can be used as the base class for all power network simulation applications.

A module for sparse matrix equation solution has been implemented. The module is very flexible. Sparse equations of any data type can be solved by the module. It is a self-contained module, managing its memory requirement dynamically by itself.

Two load flow applications: a dc load flow and an ac load flow have been discussed. The implementation of the load flow algorithms takes full advantage of inheritance. The C++ implementation of the load flow algorithms has been compared with non-OOP load flow programs for performance evaluations. It is found that the C++ implementation is as efficient as the non-OOP counterparts.

Author's experience with OOP and C++ has been very positive. The key advantage of using C++ is not so much that it can do things that the non-OOP languages can not do, but rather that it can help you think and approach your problems in ways that otherwise might not be possible or considered. The author predicts that OOP and C++ will compete with the non-OOP programming languages in the near future as an alternative way to program power system simulation software.

8. References

[1] A.F. Neyer, F.F. Wu and K. Imhof, "Objected-Oriented Programming For Flexible Software: Example of a Load Flow", IEEE Tran. on Power Systems, Vol.5, No.3, Aug. 1990, pp 689-696.

[2] A.M. Sasson, "Open systems procurement: a migration strategy", IEEE/PES 1992 Winter Meeting (92WM158-6), New York, New York, Jan. 1992

[3] J.L. Scheidt, M.E. Robertson, "The Problem of Upgrading Energy Management Systems", IEEE Tran. on Power Systems, Vol.3, No.1, Feb. 1988. pp118-126

[4] D.G. Flinn, R.C. Dugan, "A Database for Diverse Power System Simulation Applications", IEEE Tran. on Power Systems, Vol.7, No.2, May 1992, pp784-790.

[5] M. Foley, A. Bose, W. Mitchell and A. Faustni, "An Object Based Graphical User Interface for Power Systems", IEEE/PES 1992 Winter Meeting, New York, New York, Jan. 1992

[6] B. Stroustrup, *The C++ Programming Language*, 2nd edition, Reading, MA, Addison-Wesley.

[7] W.F. Tinney, I.W. Walker, "Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization", Proc. of the IEEE, vol.55, Nov. 1967, pp 1801-1809.

[8] D. Kirschen, G. Irisarri, Discussion to [1]

[9] J. Arrillaga, C.P. Arnold, *Computer Analysis of Power Systems*, John Wiley & Sons, 1990.

[10] M.K. Enns, W.F. Tinney, F.L.Alvarada, "Sparse Matrix Inverse Factors", IEEE Tran. on Power Systems, Vol.5, No.2, May. 1990.

[11] R. Podmore, "Criteria for Evaluating Open Energy Management Systems", IEEE Tran. on Power Systems, Vol.8, No.2, May. 1993.

[12] W.F. Tinney, V.Brandwajn, S.M. Chan, "Sparse Vector Method", IEEE Tran. on PAS, Vol.104, No.2, Feb. 1985, pp295-301.

[13] Borland International, *Borland C++ Programmer's Guide*, version 3.1, 1992.

[14] E. Zhou, A. Nalse, "Simulation of DC Power Distribution Systems", 1994 IEEE I&CPS Tech. Conf., Irvine, CA, May 1994, pp 191-195.

Erzhuan Zhou (IEEE member 1990) received his B.Sc. degree in EE from Hunan University, Hunan, China in 1982; his M.Sc and his Ph.D. degrees in EE from Tsinghua University, Beijing, China, in 1984 and 1987 respectively. From 1990 to 1992 he was an assistant professor with the department of EE, the University of Saskatchewan. He is now the vice president of EDSA Micro Corp., Bloomfield Hills, Michigan. Dr. Zhou's current research interests are power system simulation, OOP applications, oscillations in power systems and application of PC to power engineering.