******

SMV – An Object-Oriented Sparse Matrix-Vector Computation Library

Programming Manual

Zhijun Wu and Yin Zhang

June, 2000

Department of Computational and Applied Mathematics

Rice University

******

Contents

Preface **iii**

1 Overview **1**
1.1 Object-Oriented Principles 2
1.2 Organization of Matrix-Vector Objects 3
1.3 Matrix-Vector Operations 4
1.4 Object-Oriented Matrix Computation 4
1.5 Programming in SMV 5

2 SMV Objects **7**
2.1 Vectors 8
2.2 Dense-Row-Vectors 9
2.3 Dense-Column-Vectors 13
2.4 Sparse-Row-Vectors 16
2.5 Sparse-Column-Vectors 19
2.6 Matrices 22
2.7 Dense-Row-Matrices 23
2.8 Dense-Column-Matrices 26
2.9 Sparse-Row-Matrices 29
2.10 Sparse-Column-Matrices 32

3 Example Programs **35**
3.1 Vector Inner Product 36
3.2 Saxpy 38
3.3 Matrix Multiplication 41
3.4 Gaxpy 43

4 Future Development **46**
4.1 Performance Issues 47
4.2 Parallelization 47
4.3 Object-Oriented Optimization 48

Preface

Vectors and matrices are primary elements of modern scientific computation. Libraries for matrix-vector computation have been key components of many scientific computing applications including optimization, control, and simulation. In order to support such applications, the libraries are required to be efficient, portable, and easy to use. In large-scale applications, it is desirable to run them on parallel platforms as well.

The SMV library is an object-oriented matrix-vector class library and is developed to support dense and sparse matrix-vector computation in optimization applications. The library is written in C++. The current version runs only on single processor platforms. A parallel version will be developed in future efforts.

There are many similar libraries developed in the past, for dense, sparse, parallel (shared or distributed), and object-oriented matrix-vector computation. Most of them focus on one or two of the features. The purpose of the SMV library is to unify all the features and organize them in a single object-oriented system. However, since the library is targeted for special applications in optimization, it is restricted to a small set of matrix-vector classes and operations and to only a few simple storage formats.

One of the advantages of SMV is that users can write a program with SMV in a familiar mathematical language such as $C = A * B$ without implementing the real code and distinguishing A, B, and C to be vectors or matrices, sparse or dense, and serial or parallel (in future version). This allows the users to concentrate on their algorithms rather than matrix-vector level computational details.

Another advantage of SMV over other similar libraries is that the matrix and vector objects and their relationships are defined and understood more in terms of their mathematical meanings. The users can therefore use or manipulate the objects more easily based on their known conventions. They

do not have to follow many implementation-specific rules. For example, in SMV, a vector can be a row or column vector as often defined in linear algebra. Then a row vector times a column vector is the inner product of the two vectors, while the other way around is the outer product of them. Also, in SMV, a row matrix is defined as a set of row vectors, and a column matrix as a set of column vectors. If A is a row matrix, then $A[i]$ is the ith row vector of A, and $A[i][j]$ is naturally the jth nonzero element of the ith vector of A.

The design principle for the SMV library is object-oriented. This is not only for practicing the object-oriented programming methodology in design of scientific computing libraries, but also for supporting higher-level object-oriented software development such as software for engineering and scientific optimization. The library is organized around matrix and vector objects, and for each of the objects, certain operations are defined. The internal data structure is encapsulated, and so are the complicated matrix-vector level calculations such as parallel and sparse calculations. There are certainly overheads for building-in these calculations, but the average users are expected to benefit from the library for easier programming and better performance.

The development of the SMV library is part of the joint effort in large-scale scientific computation by Rice CRPC and Los Almos Advanced Computing Laboratory. The support from the DOE/LANL contract 03891-99-23 and the access to the computing facility at Rice CAAM Department are hereby acknowledged.

Zhijun Wu and Yin Zhang
Houston, Texas, June, 2000

Chapter 1

Overview

1.1 Object-Oriented Principles

The object-oriented programming technique is used for organizing and developing large-scale software systems. In traditional programming methods, a software package typically contains a set of programs each having a set of data objects. The objects in different programs usually are "segregated". The relationships among them are unclear. In object-oriented approaches, the programs are organized around the data objects: The data objects in a given problem domain are first identified. The properties of the objects are defined along with their relationships such as shared properties, functions, etc. The actions upon the objects are then programmed. They are associated with and defined for the objects.

For example, in traditional matrix-vector libraries, the system consists of a set of subroutines each corresponding to some specific matrix-vector operations such as vector inner-product, saxpy, gaxpy, etc. Each of them may be applied to certain types of vectors or matrices. Changes in vector or matrix types may affect the design of the operations. However, an object-oriented matrix-vector library will have only vector and matrix objects. The operations are defined around the objects and associated with their specific types. The objects can be modified without affecting the operations at all.

An object is a data abstraction. It refers to a group of physical or conceptual identities in the problem domain. An object has private and public data. It has parent and children objects to inherit and pass on properties. Operations that may be applied to the object are defined in the object. They can only be invoked through the object.

Since complex systems usually consist of numerous different data objects while there are only a few routine operations, it is more natural to organize a software system around objects instead of operations, and the system will be easier to update and maintain. More importantly, object-oriented programming supports data abstraction, inheritance, and encapsulation. By using objects, data can be organized in different abstract levels, shared through family relationships, and hidden from the users. These features offer a great deal of advantages for modern computer programming and software development.

1.2 Organization of Matrix-Vector Objects

For matrix-vector computation, matrices and vectors are basic objects. However, in order to include sparse as well as regular dense computation, more specific object types are required, for example, dense and sparse matrices and vectors. For the storage reason, we also distinguish in between row and column matrices and vectors. A vector can be defined as a row or column vector. A matrix can be defined as a row or column matrix. A row matrix consists of a set of row vectors. A column matrix consists of a set of column vectors.

So, at the top level, we have the vector and matrix objects. The vector object has four children objects, dense row vector, sparse row vector, dense column vector, and sparse column vector. Similarly, the matrix object has four children objects, dense row matrix, sparse row matrix, dense column matrix, and sparse column matrix.

Matrices are defined in terms of vectors. Therefore, a dense row matrix consists of a set of dense row vectors. A sparse row matrix consists a set of sparse row vectors. A dense column matrix consists of a set of dense column vectors. A sparse column matrix consists of a set of sparse column vectors.

A vector has a set of data entries, including an integer for the vector's dimension, an integer for the number of nonzero elements in the vector, an index set for the nonzero elements, and an array of the nonzero elements. A dense vector has default values for the number of nonzero elements and the index set. The values for a sparse vector are set to agree with the sparse structure of the vector.

The data entries for a matrix include two integers for the numbers of rows and columns in the matrix, respectively, and an array of vectors. If the matrix is dense, the vectors are dense. Otherwise, they are sparse.

Each of the vector and matrix objects have associated operations such as addition, subtraction, multiplication, inner-product, etc. There are also operations for data access, transfer, and reorganization, such as getting the index or value of an element, transposing vectors or matrices, and changing the storage format from row matrix to column matrix, or vice versa.

1.3 Matrix-Vector Operations

The definitions of row and column vectors are consistent with those in linear algebra. A row vector can only be added to or subtracted from a row vector, and a column vector can only be added to or subtracted from a column vector. A row vector times a column vector is an inner-product of the two vectors. A column vector times a row vector is an outer-product of the two vectors.

Similarly, a row matrix can only be added to or subtracted from a row matrix, and a column matrix can only be added to or subtracted from a column matrix. A row matrix times a column matrix is the same as a column matrix times a row matrix, but computationally, the former is done through inner-products while the latter through outer-products.

For matrix-vector operations, a row matrix can only time a column vector and the product is a column vector, and a row vector can only time a column matrix and the product is a row vector.

Naturally, the transpose of a row vector is a column vector. The transpose of a column vector is a row vector. The transpose of a row matrix is a column matrix, and the transpose of a column matrix is a row matrix.

If v is a vector, either row or column, then $v[i]$ is the value of ith nonzero element of v, and $v(i)$ is the index of the element. Let A be a row matrix. Since it is defined as a set of row vectors, $A[i]$ is referred to as the ith row vector of A, and $A[i][j]$ and $A[i](j)$ are the value and index of the jth nonzero element in $A[i]$, respectively. If A is a column matrix, $A[i]$ is the ith column vector, and $A[i][j]$ and $A[i](j)$ are the value and index of the jth nonzero element in $A[i]$.

1.4 Object-Oriented Matrix Computation

Traditionally, matrix-vector computation is done with detailed calculations, loops, and procedures. For sparse or parallel computation, the programs are even more complicated. Instead, with well-defined matrix-vector objects, the high-level matrix-vector computation can be coded easily in an object-oriented fashion. The operations upon the objects can be expressed in simple and familiar mathematical forms. The logical structure of the program is clear and independent of low level implementation.

For example, a traditional C code for sparse matrix-vector multiplication looks more or less like the following, where A is a sparse matrix, and u and v are dense vectors. Suppose that A is $n \times n$, and u and v are n-dimensional. Also, suppose that A has m nonzero elements and they are stored in A as an array, and their row and column indices are collected in two integer arrays I and J.

```
for (int i = 0; i < n; i ++) u [i] = 0;

for (int i = 0, j = 0, l = 0; l < m; l++) {
    i = I [l];
    j = J [l];
    u [i] = u [i] + A [l] * v [j];
    }
```

The above code can be improved if a better storage scheme is used, but it will look even more complicated. However, if we use the objects we have discussed in previous section, the code will be much simpler. Let A be defined as a row matrix, and u and v as column vectors. Then we only need a single statement for the above calculation:

```
u = A * v;
```

The code is not only simple, but also in a familiar mathematical expression which anyone can read and write without worrying about its low level programming details whatever they are.

1.5 Programming in SMV

SMV is written in C++. All objects are implemented in C++ classes. There are ten general objects with eight of them accessible to the users. They are classified into two classes of objects, vector and matrix objects. Vector objects include general vector, dense row vector, dense column vector, sparse row vector, and sparse column vector. Matrix objects include general matrix, dense row matrix, dense column matrix, sparse row matrix, and sparse column matrix. These objects are implemented as the following C++ classes in SMV.

Vector	Matrix
DenseRowVector	**DenseRowMatrix**
DenseColVector	**DenseColMatrix**
SparseRowVector	**SparseRowMatrix**
SparseColVector	**SparseColMatrix**

Vector and **Matrix** are two general classes. The specific vector and matrix objects are implemented as subclasses of **Vector** or **Matrix**. More detailed descriptions of the objects and their C++ implementation are given in the following chapter.

In order to use SMV library, a header file **smv.h** must be included in the beginning of the program. The SMV objects can then be defined, and the operations on the objects can be done directly by using corresponding member operators or functions. Some example SMV programs are given in Chapter 3.

Chapter 2

SMV Objects

2.1 Vectors

General Description:

Vector is a general object with some data shared by specific vector types, sparse or dense, row or column. A vector has two integer data, **dimension**, the dimension of the vector, and **numNonZeros**, the number of the nonzeros in the vector. The two integers are protected and can only be accessed through the member functions of the vector.

There are two member functions open to public. One is **getDimension** () and the other **getNumNonZeros** (). They can be called to obtain the dimension and the number of the nonzeros of the vector.

A **Vector** can be of int, float, or double types. We define **Vector** as a template with its type as a parameter.

C++ Definition:

```
template <class Type>
class Vector {

public:

// Member Functions:

   int getDimension ();
   int getNumNonZeros ();

protected:

// Internal Data:

   int dimension;
   int numNonZeros;

};
```

2.2 Dense-Row-Vectors

General Description:

DenseRowVector is a vector subclass. Every vector subclass has two sets of data, **index** and **array**. The nonzero elements are stored in **array**. Their positions in the vector are recorded in **index**. For dense vectors, **index** is not really so useful although it still is set to default values.

Every vector subclass has a basic set of member functions that can be used to read or write vector elements and compute vector norms. The functions include overloaded operators [], (), and function $norm()$. Suppose that v is defined as a vector. Then, in general, $v[i]$ returns the ith nonzero element of v, while $v(i)$ the index of $v[i]$. The function $norm()$ returns the l_2 norm of v.

Every vector subclass has a large number of member functions used for all arithmetic operations upon the vector objects. They include vector addition, subtraction, and multiplication, vector inner and outer products, etc. We implement the operations by using the conventional operators $+$, $-$, and $*$. They are also overloaded for all different vector subclasses.

There are also some other types of member functions, basically used for changing the storage format of the vector, for example, changing a row vector to a column vector, or a sparse vector to a dense vector, or vice versa.

A vector subclass can also be of int, float, or double types. We define a vector subclass as a template with its type as a parameter.

A **DenseRowVector** is a dense vector, and therefore, the number of nonzero elements is equal to the dimension or the length of the vector. It is also a row vector and certain rules apply, for example, it cannot be added to or subtracted from a column vector.

C++ Definition:

```
template <class Type>
class DenseRowVector : public Vector<Type> {

protected:

// Internal Data:
```

```
  int *index; Type *array;

public:

//Member Functions:

// Initialization / Termination:

  DenseRowVector ( const int = 1 );
  DenseRowVector ( const int, const Type* );
  DenseRowVector ( const DenseRowVector& );

 ~DenseRowVector () { delete [] index, array; }

// Basic Operations:

  double norm ();

  int& operator () ( const int );
  Type& operator [] ( const int );

  DenseRowVector& operator = ( const DenseRowVector& );

// Vector Addition / Subtraction:

  friend DenseRowVector operator + ( const DenseRowVector& );
  friend DenseRowVector operator - ( const DenseRowVector& );

  friend DenseRowVector operator +
  ( const DenseRowVector&, const DenseRowVector& );
  friend DenseRowVector operator -
  ( const DenseRowVector&, const DenseRowVector& );

  friend DenseRowVector operator +
  ( const DenseRowVector&, const SparseRowVector<Type>& );
  friend DenseRowVector operator -
  ( const DenseRowVector&, const SparseRowVector<Type>& );
```

```
  friend DenseRowVector operator +
  ( const SparseRowVector<Type>&, const DenseRowVector& );
  friend DenseRowVector operator -
  ( const SparseRowVector<Type>&, const DenseRowVector& );

// Vector-Scaler Multiplication:

  friend DenseRowVector operator *
  ( const Type, const DenseRowVector& );
  friend DenseRowVector operator *
  ( const DenseRowVector&, const Type );

// Vector-Vector Multiplication:

  friend Type operator *
  ( const DenseRowVector&, const DenseColVector<Type>& );
  friend Type operator * (
   const DenseRowVector&, const SparseColVector<Type>& );

// Matrix-Vector Multiplication:

  friend DenseRowVector operator *
  ( const DenseRowVector&, const DenseColMatrix<Type>& );
  friend DenseRowVector operator *
  ( const DenseRowVector&, const SparseColMatrix<Type>& );

  friend DenseColMatrix<Type> operator *
  ( const DenseColVector<Type>&, const DenseRowVector& );
  friend SparseColMatrix<Type> operator *
  ( const SparseColVector<Type>&, const DenseRowVector& );

// Row-Column Exchange:

  friend DenseRowVector trans ( const DenseColVector<Type>& );
  friend DenseColVector<Type> trans ( const DenseRowVector& );
```

```
// Dense-Sparse Transform:

  friend DenseRowVector
  sparse2dense ( const SparseRowVector<Type>& );
  friend SparseRowVector<Type>
  dense2sparse ( const DenseRowVector& );

};
```

2.3 Dense-Column-Vectors

General Description:

DenseColVetor is very similar to **DenseRowVector** except that it can only be used as a column vector. The properties of a vector subclass as described in previous section can all apply to **DenseColVector**.

C++ Definition:

```
template <class Type>
class DenseColVector : public Vector<Type> {

protected:

// Internal Data:

  int *index; Type *array;

public:

// Member Functions:

// Initialization / Termination:

  DenseColVector ( const int = 1 );
  DenseColVector ( const int, const Type* );
  DenseColVector ( const DenseColVector& );

 ~DenseColVector () { delete [] index, array; }

// Basic Operations:

  double norm ();

  int& operator () ( const int );
  Type& operator [] ( const int );
```

```
  DenseColVector& operator = ( const DenseColVector& );

// Vector Addition / Subtraction:

  friend DenseColVector operator + ( const DenseColVector& );
  friend DenseColVector operator - ( const DenseColVector& );

  friend DenseColVector operator +
  ( const DenseColVector&, const DenseColVector& );
  friend DenseColVector operator -
  ( const DenseColVector&, const DenseColVector& );

  friend DenseColVector operator +
  ( const DenseColVector&, const SparseColVector<Type>& );
  friend DenseColVector operator -
  ( const DenseColVector&, const SparseColVector<Type>& );

  friend DenseColVector operator +
  ( const SparseColVector<Type>&, const DenseColVector& );
  friend DenseColVector operator -
  ( const SparseColVector<Type>&, const DenseColVector& );

// Vector-Scaler Multiplication:

  friend DenseColVector operator *
  ( const Type, const DenseColVector& );
  friend DenseColVector operator *
  ( const DenseColVector&, const Type );

// Vector-Vector Multiplication:

  friend Type operator *
  ( const DenseRowVector<Type>&, const DenseColVector& );
  friend Type operator *
  ( const SparseRowVector<Type>&, const DenseColVector& );
```

```
// Matrix-Vector Multiplication:

  friend DenseColVector operator *
  ( const DenseRowMatrix<Type>&, const DenseColVector& );
  friend DenseColVector operator *
  ( const SparseRowMatrix<Type>&, const DenseColVector& );

  friend DenseColMatrix<Type> operator *
  ( const DenseColVector&, const DenseRowVector<Type>& );
  friend SparseRowMatrix<Type> operator *
  ( const DenseColVector&, const SparseRowVector<Type>& );

// Row-Column Exchange:

  friend DenseRowVector<Type> trans ( const DenseColVector& );
  friend DenseColVector trans ( const DenseRowVector<Type>& );

// Dense-Sparse Transform:

  friend DenseColVector
  sparse2dense ( const SparseColVector<Type>& );
  friend SparseColVector<Type>
  dense2sparse ( const DenseColVector& );

};
```

2.4 Sparse-Row-Vectors

General Description:

SparseRowVector is a vector subclass similar to **DenseRowVector** except that it is sparse, that is, it has only a few nonzero elements, and the number of nonzero elements is less than the dimension or the length of the vector. For sparse vectors, calculations usually are complicated since only those among nonzero elements are required and they need to be found out by examining the indices of the nonzero elements. However, at the object level, the structure of a sparse vector is not so different from a dense vector because the sparse calculations are hid in the lower computational levels.

C++ Definition:

```
template <class Type>
class SparseRowVector : public Vector<Type>
{

protected:

// Internal Data:

  int *index; Type *array;

public:

// Member Functions:

// Initialization / Termination:

  SparseRowVector ( const int = 1, const int = 1 );
  SparseRowVector ( const SparseRowVector& );

  SparseRowVector
  ( const int, const int, const int*, const Type* );
```

```
 ~SparseRowVector () { delete [] index, array; }

// Basic Operations:

  double norm ();

  int& operator () ( const int );
  Type& operator [] ( const int );

  SparseRowVector& operator = ( const SparseRowVector& );

// Vector Addition / Subtraction:

  friend SparseRowVector operator + ( const SparseRowVector& );
  friend SparseRowVector operator - ( const SparseRowVector& );

  friend SparseRowVector operator +
  ( const SparseRowVector&, const SparseRowVector& );
  friend SparseRowVector operator -
  ( const SparseRowVector&, const SparseRowVector& );

  friend DenseRowVector<Type> operator +
  ( const SparseRowVector&, const DenseRowVector<Type>& );
  friend DenseRowVector<Type> operator -
  ( const SparseRowVector&, const DenseRowVector<Type>& );

  friend DenseRowVector<Type> operator +
  ( const DenseRowVector<Type>&, const SparseRowVector& );
  friend DenseRowVector<Type> operator -
  ( const DenseRowVector<Type>&, const SparseRowVector& );

// Vector-Scaler Multiplication:

  friend SparseRowVector
  operator * ( const Type, const SparseRowVector& );
  friend SparseRowVector
  operator * ( const SparseRowVector&, const Type );
```

```
// Vector-Vector Multiplication:

  friend Type operator *
  ( const SparseRowVector&, const SparseColVector<Type>& );
  friend Type operator *
  ( const SparseRowVector&, const DenseColVector<Type>& );

// Matrix-Vector Multiplication:

  friend SparseRowVector operator *
  ( const SparseRowVector&, const SparseColMatrix<Type>& );
  friend DenseRowVector<Type> operator *
  ( const SparseRowVector&, const DenseColMatrix<Type>& );

  friend SparseRowMatrix<Type> operator *
  ( const SparseColVector<Type>&, const SparseRowVector& );
  friend SparseRowMatrix<Type> operator *
  ( const DenseColVector<Type>&, const SparseRowVector& );

// Row-Column Exchange:

  friend SparseRowVector trans ( const SparseColVector<Type>& );
  friend SparseColVector<Type> trans ( const SparseRowVector& );

// Dense-Sparse Transform:

  friend SparseRowVector
  dense2sparse ( const DenseRowVector<Type>& );
  friend DenseRowVector<Type>
  sparse2dense ( const SparseRowVector& );

};
```

2.5 Sparse-Column-Vectors

General Description:

SparseColVector is very similar to **SparseRowVector** except that it can only be used as a column vector. It has the same set of member functions as **SparseRowVector** with possibly different types of objects as their arguments.

C++ Definition:

```
template <class Type>
class SparseColVector : public Vector<Type>
{

protected:

// Internal Data:

  int *index; Type *array;

public:

// Member Functions:

// Initialization / Termination:

  SparseColVector ( const int = 1, const int = 1 );
  SparseColVector ( const SparseColVector& );

  SparseColVector
  ( const int, const int, const int*, const Type* );

 ~SparseColVector () { delete [] index, array; }

// Basic Operations:
```

```
  double norm ();

  int& operator () ( const int );
  Type& operator [] ( const int );

  SparseColVector& operator = ( const SparseColVector& );

// Vector Addition / Subtraction:

  friend SparseColVector operator + ( const SparseColVector& );
  friend SparseColVector operator - ( const SparseColVector& );

  friend SparseColVector operator +
  ( const SparseColVector&, const SparseColVector& );
  friend SparseColVector operator -
  ( const SparseColVector&, const SparseColVector& );

  friend DenseColVector<Type> operator +
  ( const SparseColVector&, const DenseColVector<Type>& );
  friend DenseColVector<Type> operator -
  ( const SparseColVector&, const DenseColVector<Type>& );

  friend DenseColVector<Type> operator +
  ( const DenseColVector<Type>&, const SparseColVector& );
  friend DenseColVector<Type> operator -
  ( const DenseColVector<Type>&, const SparseColVector& );

// Vector-Scaler Multiplication:

  friend SparseColVector
  operator * ( const Type, const SparseColVector& );
  friend SparseColVector
  operator * ( const SparseColVector&, const Type );

// Vector-Vector Multiplication:

  friend Type operator *
```

```
  ( const SparseRowVector<Type>&, const SparseColVector& );
  friend Type operator *
  ( const DenseRowVector<Type>&, const SparseColVector& );

// Matrix-Vector Multiplication:

  friend SparseColVector operator *
  ( const SparseRowMatrix<Type>&, const SparseColVector& );
  friend DenseColVector<Type> operator *
  ( const DenseRowMatrix<Type>&, const SparseColVector& );

  friend SparseRowMatrix<Type> operator *
  ( const SparseColVector&, const SparseRowVector<Type>& );
  friend SparseColMatrix<Type> operator *
  ( const SparseColVector&, const DenseRowVector<Type>& );

// Row-Column Exchange:

  friend SparseRowVector<Type> trans ( const SparseColVector& );
  friend SparseColVector trans ( const SparseRowVector<Type>& );

// Dense-Sparse Transform:

  friend SparseColVector
  dense2sparse ( const DenseColVector<Type>& );
  friend DenseColVector<Type>
  sparse2dense ( const SparseColVector& );

};
```

2.6 Matrices

General Description:

Matrix is a general matrix object with data shared by specific matrix types, sparse or dense, row or column. A matrix has two integer data, **numRows**, the number of rows, and **numCols**, the number of columns. The two integers are protected and can only be accessed through the member functions of the matrix.

There are two member functions open to public. One is **getNumRows ()** and the other **getNumCols ()**. They can be called to obtain the numbers of rows and columns of the matrix.

A **Matrix** can be of int, float, or double types. We define **Matrix** as a template with its type as a parameter.

C++ Definition:

```
template <class Type>
class Matrix {

public:

// Member Functions:

  int getNumRows ();
  int getNumCols ();

protected:

// Internal Data:

  int numRows;
  int numCols;

};
```

2.7 Dense-Row-Matrices

General Description:

DenseRowMatrix is a matrix subclass. Every matrix subclass has an array of vectors. They are the row vectors of the matrix if the matrix is a row matrix, or the column vectors if the matrix is a column matrix. If the matrix is sparse, the vectors must also be sparse. **DenseRowMatrix** is a row matrix with an array of dense row vectors.

The vectors of a matrix, row or column, can be accessed to by using operator []. Let A be a matrix. Then, $A[i]$ returns the ith row or column of A. Since $A[i]$ is a vector. The element within the vector can be accessed to as a general vector element. Therefore, $A[i][j]$ is the jth nonzero element of the ith row or column vector of A, and $A[i](j)$ is the index of $A[i][j]$ in $A[i]$.

Every matrix subclass has a number of member functions used for basic arithmetic operations upon the matrix objects. They include matrix addition, subtraction, and multiplication, matrix-vector multiplication, etc. We implement the operations by using the conventional operators $+$, $-$, and $*$. They are overloaded for different matrix subclasses.

There are also some other types of member functions, basically used for changing the storage format of the matrix, for example, changing a row matrix to a column matrix, or a sparse matrix to a dense matrix, or vice versa.

A matrix subclass can also be of int, float, or double types. We define a matrix subclass as a template with its type as a parameter.

C++ Definition:

```
template <class Type>
class DenseRowMatrix : public Matrix<Type> {

public:

// matrix operations:

// default constructuors:
```

```
  DenseRowMatrix ( const int = 1, const int = 1 );
  DenseRowMatrix ( const DenseRowMatrix& );

  DenseRowMatrix
  ( const int, const int, const DenseRowVector<Type>* );

 ~DenseRowMatrix () { delete [] array; }

// other operations:

  DenseRowMatrix& operator = ( const DenseRowMatrix& );

  DenseRowVector<Type>& operator [] ( const int );

  friend DenseRowMatrix operator + ( const DenseRowMatrix& );
  friend DenseRowMatrix operator - ( const DenseRowMatrix& );

  friend DenseRowMatrix operator +
  ( const DenseRowMatrix&, const DenseRowMatrix& );
  friend DenseRowMatrix operator -
  ( const DenseRowMatrix&, const DenseRowMatrix& );

  friend DenseRowMatrix operator +
  ( const DenseRowMatrix&, const SparseRowMatrix<Type>& );
  friend DenseRowMatrix operator - (
   const DenseRowMatrix&, const SparseRowMatrix<Type>& );

  friend DenseRowMatrix operator +
  ( const SparseRowMatrix<Type>&, const DenseRowMatrix& );
  friend DenseRowMatrix operator -
  ( const SparseRowMatrix<Type>&, const DenseRowMatrix& );

  friend DenseRowMatrix operator *
  ( const Type, const DenseRowMatrix& );
  friend DenseRowMatrix operator *
  ( const DenseRowMatrix&, const Type );
```

```
// matrix functions:

  friend DenseColVector<Type> operator *
  ( const DenseRowMatrix&, const DenseColVector<Type>& );
  friend DenseColVector<Type> operator *
  ( const DenseRowMatrix&, const SparseColVector<Type>& );

  friend DenseRowMatrix operator *
  ( const DenseRowMatrix&, const DenseColMatrix<Type>& );
  friend DenseColMatrix<Type> operator *
  ( const DenseColMatrix<Type>&, const DenseRowMatrix& );
  friend DenseRowMatrix operator *
  ( const DenseRowMatrix&, const SparseColMatrix<Type>& );
  friend DenseColMatrix<Type> operator *
  ( const SparseColMatrix<Type>&, const DenseRowMatrix& );

  friend DenseRowMatrix trans ( const DenseColMatrix<Type>& );
  friend DenseColMatrix<Type> trans ( const DenseRowMatrix& );

  friend DenseRowMatrix
  col2row ( const DenseColMatrix<Type>& );
  friend DenseColMatrix<Type>
  row2col ( const DenseRowMatrix& );

  friend DenseRowMatrix
  sparse2dense ( const SparseRowMatrix<Type>& );
  friend SparseRowMatrix<Type>
  dense2sparse ( const DenseRowMatrix& );

protected:

  DenseRowVector<Type> *array;

};
```

2.8 Dense-Column-Matrices

General Description:

DenseColMatrix has a similar structure as **DenseRowMatrix** except that it is considered as a matrix with a set of column vectors. In SMV, the two classes of matrices are different types of objects. Row matrices cannot add or subtract column matrices. A row matrix times a column matrix is implemented through vector inner-product, while the other way around through vector outer-product.

C++ Definition:

```
template <class Type>
class DenseColMatrix : public Matrix<Type> {

public:

// matrix operations:

// default constructuors:

  DenseColMatrix ( const int = 1, const int = 1 );
  DenseColMatrix ( const DenseColMatrix& );

  DenseColMatrix
  ( const int, const int, const DenseColVector<Type>* );

 ~DenseColMatrix () { delete [] array; }

// other operations:

  DenseColMatrix& operator = ( const DenseColMatrix& );

  DenseColVector<Type>& operator [] ( const int );

  friend DenseColMatrix operator + ( const DenseColMatrix& );
```

```
  friend DenseColMatrix operator - ( const DenseColMatrix& );

  friend DenseColMatrix operator +
  (const DenseColMatrix&, const DenseColMatrix& );
  friend DenseColMatrix operator -
  ( const DenseColMatrix&, const DenseColMatrix& );

  friend DenseColMatrix operator +
  ( const DenseColMatrix&, const SparseColMatrix<Type>& );
  friend DenseColMatrix operator -
  ( const DenseColMatrix&, const SparseColMatrix<Type>& );

  friend DenseColMatrix operator +
  ( const SparseColMatrix<Type>&, const DenseColMatrix& );
  friend DenseColMatrix operator -
  ( const SparseColMatrix<Type>&, const DenseColMatrix& );

  friend DenseColMatrix operator *
  ( const Type, const DenseColMatrix& );
  friend DenseColMatrix operator *
  ( const DenseColMatrix&, const Type );

// matrix functions:

  friend DenseRowVector<Type> operator *
  ( const DenseRowVector<Type>&, const DenseColMatrix& );
  friend DenseRowVector<Type> operator *
  ( const SparseRowVector<Type>&, const DenseColMatrix& );

  friend DenseColMatrix operator *
  ( const DenseColMatrix&, const DenseRowMatrix<Type>& );
  friend DenseRowMatrix<Type> operator *
  ( const DenseRowMatrix<Type>&, const DenseColMatrix& );
  friend DenseRowMatrix<Type> operator *
  ( const DenseColMatrix&, const SparseRowMatrix<Type>& );
  friend DenseRowMatrix<Type> operator *
  ( const SparseRowMatrix<Type>&, const DenseColMatrix& );
```

```
  friend DenseColMatrix operator *
  ( const DenseColVector<Type>&, const DenseRowVector<Type>& );

  friend DenseColMatrix trans ( const DenseRowMatrix<Type>& );
  friend DenseRowMatrix<Type> trans ( const DenseColMatrix& );

  friend DenseColMatrix
  row2col ( const DenseRowMatrix<Type>& );
  friend DenseRowMatrix<Type>
  col2row ( const DenseColMatrix& );

  friend DenseColMatrix
  sparse2dense ( const SparseColMatrix<Type>& );
  friend SparseColMatrix<Type>
  dense2sparse ( const DenseColMatrix& );

protected:

  DenseColVector<Type> *array;

};
```

2.9 Sparse-Row-Matrices

General Description:

SparseRowMatrix is similar to **DenseRowMatrix** except that the row vectors are sparse. In other words, it is defined in terms of **SparseRowVector**.

C++ Definition:

```
template <class Type>
class SparseRowMatrix : public Matrix<Type>
{

public:

// matrix operations:

// default constructuors:

  SparseRowMatrix ( const int = 1, const int = 1 );
  SparseRowMatrix ( const SparseRowMatrix& );

  SparseRowMatrix
  ( const int, const int, const SparseRowVector<Type>* );

 ~SparseRowMatrix () { delete [] array; }

// other operations:

  SparseRowMatrix& operator = ( const SparseRowMatrix& );

  SparseRowVector<Type>& operator [] ( const int );

  friend SparseRowMatrix operator + ( const SparseRowMatrix& );
  friend SparseRowMatrix operator - ( const SparseRowMatrix& );
```

```
  friend SparseRowMatrix operator +
  ( const SparseRowMatrix&, const SparseRowMatrix& );
  friend SparseRowMatrix operator -
  ( const SparseRowMatrix&, const SparseRowMatrix& );

  friend DenseRowMatrix<Type> operator +
  ( const SparseRowMatrix&, const DenseRowMatrix<Type>& );
  friend DenseRowMatrix<Type> operator -
  ( const SparseRowMatrix&, const DenseRowMatrix<Type>& );

  friend DenseRowMatrix<Type> operator +
  ( const DenseRowMatrix<Type>&, const SparseRowMatrix& );
  friend DenseRowMatrix<Type> operator -
  ( const DenseRowMatrix<Type>&, const SparseRowMatrix& );

  friend SparseRowMatrix operator *
  ( const Type, const SparseRowMatrix& );
  friend SparseRowMatrix operator *
  ( const SparseRowMatrix&, const Type );

// matrix functions:

  friend DenseColVector<Type> operator *
  ( const SparseRowMatrix&, const DenseColVector<Type>& );
  friend SparseColVector<Type> operator *
  ( const SparseRowMatrix&, const SparseColVector<Type>& );

  friend DenseRowMatrix<Type> operator *
  ( const SparseRowMatrix&, const DenseColMatrix<Type>& );
  friend DenseRowMatrix<Type> operator *
  ( const DenseColMatrix<Type>&, const SparseRowMatrix& );
  friend SparseRowMatrix operator *
  ( const SparseRowMatrix&, const SparseColMatrix<Type>& );
  friend SparseRowMatrix operator *
  ( const SparseColMatrix<Type>&, const SparseRowMatrix& );

  friend SparseRowMatrix operator *
```

```
  ( const SparseColVector<Type>&, const SparseRowVector<Type>& );
  friend SparseRowMatrix operator *
  ( const DenseColVector<Type>&, const SparseRowVector<Type>& );

  friend SparseRowMatrix trans ( const SparseColMatrix<Type>& );
  friend SparseColMatrix<Type> trans ( const SparseRowMatrix& );

  friend SparseRowMatrix
  col2row ( const SparseColMatrix<Type>& );
  friend SparseColMatrix<Type>
  row2col ( const SparseRowMatrix& );

  friend DenseRowMatrix<Type>
  sparse2dense ( const SparseRowMatrix& );
  friend SparseRowMatrix
  dense2sparse ( const DenseRowMatrix<Type>& );

protected:

  SparseRowVector<Type> *array;

};
```

2.10 Sparse-Column-Matrices

General Description:

SparseColMatrix is similar to **DenseColMatrix** except that it is a sparse matrix. It is defined in terms of **SparseColVector**, and is a column matrix with sparse column vectors.

C++ Definition:

```
template <class Type>
class SparseColMatrix : public Matrix<Type>
{

public:

// matrix operations:

// default constructuors:

  SparseColMatrix ( const int = 1, const int = 1 );
  SparseColMatrix ( const SparseColMatrix& );

  SparseColMatrix
  ( const int, const int, const SparseColVector<Type>* );

 ~SparseColMatrix () { delete [] array; }

// other operations:

  SparseColMatrix& operator = ( const SparseColMatrix& );
  SparseColVector<Type>& operator [] ( const int );

  friend SparseColMatrix operator + ( const SparseColMatrix& );
  friend SparseColMatrix operator - ( const SparseColMatrix& );

  friend SparseColMatrix operator +
```

```
  ( const SparseColMatrix&, const SparseColMatrix& );
  friend SparseColMatrix operator -
  ( const SparseColMatrix&, const SparseColMatrix& );

  friend DenseColMatrix<Type> operator +
  ( const SparseColMatrix&, const DenseColMatrix<Type>& );
  friend DenseColMatrix<Type> operator -
  ( const SparseColMatrix&, const DenseColMatrix<Type>& );

  friend DenseColMatrix<Type> operator +
  ( const DenseColMatrix<Type>&, const SparseColMatrix& );
  friend DenseColMatrix<Type> operator -
  ( const DenseColMatrix<Type>&, const SparseColMatrix& );

  friend SparseColMatrix operator *
  ( const Type, const SparseColMatrix& );
  friend SparseColMatrix operator *
  ( const SparseColMatrix&, const Type );

// matrix functions:

  friend DenseRowVector<Type> operator *
  ( const DenseRowVector<Type>&, const SparseColMatrix& );
  friend SparseRowVector<Type> operator *
  ( const SparseRowVector<Type>&, const SparseColMatrix& );

  friend DenseColMatrix<Type> operator *
  ( const SparseColMatrix&, const DenseRowMatrix<Type>& );
  friend DenseRowMatrix<Type> operator *
  ( const DenseRowMatrix<Type>&, const SparseColMatrix& );
  friend SparseRowMatrix<Type> operator *
  ( const SparseColMatrix&, const SparseRowMatrix<Type>& );
  friend SparseRowMatrix<Type> operator *
  ( const SparseRowMatrix<Type>&, const SparseColMatrix& );

  friend SparseColMatrix operator *
  ( const SparseColVector<Type>&, const DenseRowVector<Type>& );
```

```
  friend SparseColMatrix trans ( const SparseRowMatrix<Type>& );
  friend SparseRowMatrix<Type> trans ( const SparseColMatrix& );

  friend SparseRowMatrix<Type>
  col2row ( const SparseColMatrix& );
  friend SparseColMatrix
  row2col ( const SparseRowMatrix<Type>& );

  friend DenseColMatrix<Type>
  sparse2dense ( const SparseColMatrix& );
  friend SparseColMatrix
  dense2sparse ( const DenseColMatrix<Type>& );

protected:

  SparseColVector<Type> *array;

};
```

Chapter 3

Example Programs

3.1 Vector Inner Product

```
// A driver program for computing the inner product of two
// randomly generated vectors.

#include "smv.h"

main ()
  {
   int n = 100000;                        // vector dimension
   int nsU = 10000, nsV = 10000;          // nonzero elements

// temporary variables:
   float drdcUV, drscUV, srdcUV, srscUV;

// vector classes:

   DenseRowVector<float> drU (n);
   DenseColVector<float> dcV (n);

   SparseRowVector<float> srU (n,nsU);
   SparseColVector<float> scV (n,nsV);

// generate vectors:

   srand (100);

   for (int i = 0; i < n; i++)
       drU [i] = (float) 100 * rand () / (32 * 1024);
   for (int i = 0; i < n; i++)
       dcV [i] = (float) 100 * rand () / (32 * 1024);

   for (int i = 0; i < nsU; i++)
       srU [i] = (float) 100 * rand () / (32 * 1024);
   for (int i = 0; i < nsV; i++)
       scV [i] = (float) 100 * rand () / (32 * 1024);
```

```
   for (int i = 0; i < nsU; i++)
       srU (i) = 10 * i + 10 * rand () / (32 * 1024);
   for (int i = 0; i < nsV; i++)
       scV (i) = 10 * i + 10 * rand () / (32 * 1024);

// the inner-product of a dense row vector and a dense
// column vector:
   drdcUV = drU * dcV;

// the inner-product of a dense row vector and a sparse
// column vector:
   drscUV = drU * scV;

// the inner-product of a sparse row vector and a dense
// column vector:
   srdcUV = srU * dcV;

// the inner-product of a sparse row vector and a sparse
// column vector:
   srscUV = srU * scV;

   }
```

3.2 Saxpy

```
// A driver program for saxpy operation, u + alpha * v,
// where u and v are vectors and alpha a scaler.

#include "smv.h"

main ()
  {
   int n = 1000;                          // vector dimension
   int nsU = 40, nsV = 50, nsUV = 100;  // nonzero elements

   float alpha;            // a scaler

// vector classes:

   DenseRowVector<float> drU (n), drV (n), drUV (n);
   DenseColVector<float> dcU (n), dcV (n), dcUV (n);

   SparseRowVector<float>
   srU (n, nsU), srV (n, nsV), srUV (n, nsUV);

   SparseColVector<float>
   scU (n, nsU), scV (n, nsV), scUV (n, nsUV);

// generate vectors:

   srand (100);

   alpha = (float) 100 * rand () / (32 * 1024);

   for (int i = 0; i < n; i++)
       drU [i] = (float) 100 * rand () / (32 * 1024);
   for (int i = 0; i < n; i++)
       drV [i] = (float) 100 * rand () / (32 * 1024);
```

```
    for (int i = 0; i < n; i++)
        dcU [i] = (float) 100 * rand () / (32 * 1024);
    for (int i = 0; i < n; i++)
        dcV [i] = (float) 100 * rand () / (32 * 1024);

    for (int i = 0; i < nsU; i++)
        srU [i] = (float) 100 * rand () / (32 * 1024);
    for (int i = 0; i < nsV; i++)
        srV [i] = (float) 100 * rand () / (32 * 1024);

    for (int i = 0; i < nsU; i++)
        srU (i) = 25 * i + 25 * rand () / (32 * 1024);
    for (int i = 0; i < nsV; i++)
        srV (i) = 20 * i + 20 * rand () / (32 * 1024);

    for (int i = 0; i < nsU; i++)
        scU [i] = (float) 100 * rand () / (32 * 1024);
    for (int i = 0; i < nsV; i++)
        scV [i] = (float) 100 * rand () / (32 * 1024);

    for (int i = 0; i < nsU; i++)
        scU (i) = 25 * i + 25 * rand () / (32 * 1024);
    for (int i = 0; i < nsV; i++)
        scV (i) = 20 * i + 20 * rand () / (32 * 1024);

// saxpy of two dense row vectors:
    drUV = drU + alpha * drV;

// saxpy of two dense column vectors:
    dcUV = dcU + alpha * dcV;

// saxpy of two sparse row vectors:
    srUV = srU + alpha * srV;

// saxpy of two sparse column vectors:
    scUV = scU + alpha * scV;
```

```
// saxpy of a dense row vector and a sparse row vector:
   drUV = drU + alpha * srV;

// saxpy of a dense column vector and a sparse
// column vector:
   dcUV = dcU + alpha * scV;

// saxpy of a sparse row vector and a dense row vector:
   drUV = srU + alpha * drV;

// saxpy of a sparse column vector and a dense
// column vector:
   dcUV = scU + alpha * dcV;

   }
```

3.3 Matrix Multiplication

```
// A driver program for matrix-matrix multiplication.

#include "smv.h"

main ()
  {
   int n = 100;                  // matrix dimension

// matrix classes:

   DenseRowMatrix<float> drA (n, n), drAB (n, n);
   DenseColMatrix<float> dcB (n, n);

   SparseRowMatrix<float> srA (n, n), srAB (n, n);
   SparseColMatrix<float> scB (n, n);

// allocate memory:

   for (int i = 0; i < n; i++)
       drA [i] = * new DenseRowVector<float> (n);
   for (int i = 0; i < n; i++)
       dcB [i] = * new DenseColVector<float> (n);

   for (int i = 0; i < n; i++)
       srA [i] = * new SparseRowVector<float> (n, n/5);
   for (int i = 0; i < n; i++)
       scv [i] = * new SparseColVector<float> (n, n/5);

// generate matrices:

   srand (100);

   for (int i = 0; i < n; i++)
       for (int j = 0; j < n; j++)
```

```
            drA [i][j] = (float) 100 * rand () / (32 * 1024);

   for (int i = 0; i < n; i++)
       for (int j = 0; j < n; j++)
           dcB [i][j] = (float) 100 * rand () / (32 * 1024);

   for (int i = 0; i < n; i++) {
       for (int j = 0; j < n/5; j++)
           srA [i][j] = (float) 100 * rand () / (32 * 1024);
       for (int j = 0; j < n/5; j++)
           srA [i](j) = 5 * j + 5 * rand () / (32 * 1024);
       }

   for (int i = 0; i < n; i++) {
       for (int j = 0; j < n/5; j++)
           scB [i][j] = (float) 100 * rand () / (32 * 1024);
       for (int j = 0; j < n/5; j++)
           scB [i](j) = 5 * j + 5 * rand () / (32 * 1024);
       }

// a dense row matrix times a dense column matrix:
   drAB = drA * dcB;

// a dense row matrix times a sparse column matrix:
   drAB = drA * scB;

// a sparse row matrix times a dense column matrix:
   drAB = srA * dcB;

// a sparse row matrix times a sparse column matrix:
   srAB = srA * scB;

   }
```

3.4 Gaxpy

```
// a driver program for gaxpy operation, u + A * v,
// where u and v are two vectors and A a matrix.

#include "smv.h"

main ()
  {
   int n = 100;                // vector and matrix dimension
   int nsU = 20, nsV = 20, nsUV = 100;   // nonzero elements

// vector and matrix classes:

   DenseRowVector<float> drU (n), drV (n), drUV (n);
   DenseColVector<float> dcU (n), dcV (n), dcUV (n);

   SparseRowVector<float> srU (n, nsU), srV (n, nsV), srUV (n, nsUV);
   SparseColVector<float> scU (n, nsU), scV (n, nsV), scUV (n, nsUV);

   DenseRowMatrix<float> drA (n, n);
   DenseColMatrix<float> dcA (n, n);

// allocate memory:

   for (int i = 0; i < n; i++)
       drA [i] = * new DenseRowVector<float> (n);
   for (int i = 0; i < n; i++)
       dcA [i] = * new DenseColVector<float> (n);

   for (int i = 0; i < n; i++)
       srA [i] = * new SparseRowVector<float> (n, n/5);
   for (int i = 0; i < n; i++)
       scA [i] = * new SparseColVector<float> (n, n/5);

// generate vectors and matrices:
```

```
srand (100);

for (int i = 0; i < n; i++)
    drU [i] = (float) 10 * rand () / (32 * 1024);
for (int i = 0; i < n; i++)
    dcV [i] = (float) 10 * rand () / (32 * 1024);

for (int i = 0; i < nsU; i++)
    srU [i] = (float) 10 * rand () / (32 * 1024);
for (int i = 0; i < nsV; i++)
    scV [i] = (float) 10 * rand () / (32 * 1024);

for (int i = 0; i < nsU; i++)
    srU (i) = 5 * i + 5 * rand () / (32 * 1024);
for (int i = 0; i < nsV; i++)
    scV (i) = 5 * i + 5 * rand () / (32 * 1024);

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j ++)
        drA [i][j] = (float) 10 * rand () / (32 * 1024);

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j ++)
        dcA [i][j] = (float) 10 * rand () / (32 * 1024);

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n/5; j++)
        srA [i][j] = (float) 10 * rand () / (32 * 1024);
    for (int j = 0; j < n/5; j++)
        srA [i](j) = 5 * j + 5 * rand () / (32 * 1024);
    }

for (int i = 0; i < n; i++) {
    for (int j = 0; j < n/5; j++)
        scA [i][j] = (float) 10 * rand () / (32 * 1024);
    for (int j = 0; j < n/5; j++)
```

```
            scA [i](j) = 5 * j + 5 * rand () / (32 * 1024);
        }

// a dense row vector plus a dense row vector times
// a dense column matrix:
   drUV = drU + drV * dcA;

// a dense column vector plus a dense row matrix times
// a dense column vector:
   dcUV = dcU + drA * dcV;

// a sparse row vector plus a sparse row vector times
// a sparse column matrix:
   srUV = srU + srV * scA;

// a sparse column vector plus a sparse row matrix times
// a sparse column vector:
   scUV = scU + srA * scV;

// a dense row vector plus a sparse row vector times
// a dense column matrix:
   drUV = drU + srV * dcA;

// a dense column vector plus a dense row matrix times
// a sparse column vector:
   dcUV = dcU + drA * scV;

// a sparse row vector plus a dense row vector times
// a sparse column matrix:
   drUV = srU + drV * scA;

// a sparse column vector plus a sparse row matrix times
// a dense column vector:
   dcUV = scU + srA * dcV;

   }
```

Chapter 4

Future Development

4.1 Performance Issues

In the initial version of SMV, we have implemented all vector as well as matrix level computation with our own routines. Some of them, especially those for dense computation, may actually be called directly from some existing libraries such as LAPACK. Since the routines in these libraries have been well tuned, they usually perform more efficiently. We made SMV independent of other software packages first. In future, we may consider either improving the performance of SMV routines, or replacing them with some existing libraries.

For sparse computation, sometimes, it is better to transform sparse vectors to dense vectors and then perform regular dense computation on them. Better performance can be achieved this way since we do not need to do the extra work associated with sparse computation while being able to take advantage of dense vector computation in cache access and vector processing. However, in current implementation of SMV, we have not implemented any sparse-to-dense or dense-to-sparse transformation for our sparse calculations. We will address the issue in our future work.

Another performance issue related to current implementation of SMV is that we have overloaded a number of arithmetic operators such as $+$, $-$, and $*$. Some of these operators require a temporary storage to keep the returning object. When the object is a vector or matrix, not only is a big storage space required but also extra work to move the data. If the operators are called multiple times, the overhead can be large. From our experiments, we have not seen that current C++ compiler can help optimize the the code and reduce the cost.

4.2 Parallelization

Besides sparse computation, another issue that SMV is intended to address is automatic parallelization. By automatic parallelization we mean that users do not need to know how or where to parallelize certain part of the code, but just focus on the required vector or matrix computation. This can be done in SMV straightforwardly since calculations in SMV, sparse as well as parallel, can all be implemented inside of the vector or matrix objects, and users call member operators or functions without worrying about whether or not they require sparse or parallel computation and how they can do it.

Inside of the vector or matrix objects, related calculations can be parallelized as necessary. They can be done either manually or by using a parallel compiler. In the current version of SMV, we have not implemented any parallel code, but in future, we will consider parallelizing the dense as well as sparse calculations involved in SMV. Since sparse calculation is hard to parallelize, we will most likely rely on using a parallel compiler.

4.3 Object-Oriented Optimization

The goal of SMV is to provide matrix-vector level support for objected-oriented optimization. Therefore, the objects and related calculations in SMV are limited to this purpose. Still, some calculations are to be included such as matrix-vector functions **sum**, **max**, **min**, and **norm**, point-wise vector or matrix multiplication, matrix decomposition, etc.

As our first effort towards object-oriented optimization, we will consider using SMV to implement an object-oriented code for solving simple linear programming and linearly constrained least-squares problems using an interior point method. This, in particular, will require a sparse Cholesky factorization routine to be developed for matrix objects.