*Journal of the Nigerian Association of Mathematical Physics*
*Volume* **12 (May, 2008), 505 – 516**


# Matrix and vector data structures-foundation for scientific computing in C++

**Innocent Okoloko**
**Department of Computer Science, University of Benin**
**Benin City, Nigeria**

***Abstract***

***Matrices and vectors have a wide range of applications in science and engineering, those applications require that vectors and matrices be defined and used in systems design. However the matrix and vector data structures and their operations are not generally defined in some modern programming languages such as C++. Every C++ programmer who wishes to use them must implement them from scratch using primitive data types, even though the task he wants to perform may not be the development of data structures per se; this is a difficult task even for experienced C++ programmers. This paper presents a set of reusable template C++ matrix and vector classes capable of the basic allowable operations on vectors and matrices; which can be used by the C++ programmer who requires the task of defining and using vectors and matrices on the fly. They can therefore be used as a basis in scientific computing and computational linear algebra. They have been tested successfully on the Linux and Windows platforms.***

## 1.0 Introduction

Matrices have a very wide range of applications in science and engineering; these include computer graphics and vision, computational geometry and modeling, artificial intelligence, electrical networks, bioinformatics, industrial physics, and in many other areas. These applications require that, 1) data be represented either as vectors and/or matrices, and 2) linear algebra functions are made available for operation on the data. Some popular programming languages (such as C++) do not include the *vector* and *matrix* as a standard or extended data structures, and as such a programmer or author who needs to use them has to implement matrix and vector types and operations from scratch using C++ arrays and pointers as in [1]. However there are several difficulties with using C++ arrays directly, some of them have been identified in [2-4], they include the following:

1. Arrays cannot be dynamically created and used in C++.
2. Creating C++ arrays for an array intensive program is usually tedious and error prone. Array operations will be written at each stage of the application where they are required. This is at best a difficult task, and at worst impossible where the dimension of the required array can only be determined at run time.
3. The use of pointers may help to solve some of these problems, but the concept of C++ pointers is complex and difficult to apply to these kinds of applications, sometimes even for experienced C++ programmers.

For these reasons some authors have labelled C++ as an extremely difficult and unreliable programming language, in spite of the beauty of C++ which is not immediately obvious to non-expert programmers. This project is one of many attempts to make C++ a less difficult and more reliable programming language, and to showcase its beauty and strength, by creating standard, easily reusable vector and matrix data structures, and making same available to C++ programmers so that:

**e-mail: okolokoinno@hotmail.com, Phone: +234 08050580484**

1. The C++ programmer can learn how to avoid the problem of defining C++ arrays dynamically at run time.
2. The C++ programmer can easily define vector and matrix data structures, whose array elements can be any of the C++ numeric data types (int, float or double).
3. Researchers and students who are building systems that require matrices in C++, can forgo the onerous task of having to define arrays and array operations, and concentrate on developing the application at hand.
4. Beginning C++ programmers can learn from these classes and improve on them.
5. The C++ programmer applying the classes can reduce coding to minimum.

Many authors have previously tutored or tried to implement classes for matrix operations in their various ways, some of the literature are [2], where the design of C++ matrix classes is tutored; [3] where vector and matrix classes were used for implementing numerical formulas and functions. Also, [5] provides *The Matrix Template Library*, a set of tools for the C++ programmer. Popular complex numerical software tools such as LAPACK [6], LINPACK [7], and ScaLAPACK, also have hidden matrix capabilities. My own experience from familiarity with some of these tools listed above led me to embark on this project as a necessity. In my attempt to implement navigation algorithms in C++ on small embedded hardware, I needed to use vectors and matrices, without having to go about building them from scratch. I observed that most of the tools above are set-up of black-boxes in which the matrices are not transparent; they are ready-made software applications, so cannot be used in another program or on a small hardware. My attempt in this paper, is therefore to present a simple but comprehensive system of building blocks which can be used by the C++ programmer to create what suits him/her, with the freedom to improve on them. This is also important because, using the Java Native Interface (JNI) capability of Java; these classes can also be used in Java systems development, and so can become universal.

## 2.0 Implementation

This section concerns summary of the details of work done in creating the matrix, and vector data structures, their operations, and some of the popular standard matrix algebras that have been implemented. The linear algebras implemented here are common and have been tutored in [8-11] and in many other mathematics texts. It is impossible to include most of the equations and source code listings here, but sections that are required for use in explaining the design and functionalities are given.

### 2.1 Vector Class

This class enables the programmer to define the vector data structure. [8] defines a vector as a matrix that has only one row (row vector), or only one column (column vector), denoted by:

$$\mathbf{a} = [a_j]$$

Thus, $\mathbf{a}$ may be regarded as a 1 dimensional matrix denoted as:

$$\mathbf{a} = [a_1\ a_2 \Lambda\ a_n],$$

while its transpose

$$a^T = \begin{bmatrix} a_1 \\ a_2 \\ \mathrm{M} \\ a_n \end{bmatrix}$$

Basic allowable operations on vectors are; **transposition**, **addition**, **subtraction**, **dot (inner) product, cross product,** and **vector norm**.

In the design, the vector class forms the basis for defining the matrix class. The class is defined as a C++ header file with the name **vector_1D.h,** a standard C++ definition is given by lines 1 to 3 in the code listing below.

**//Class Definition**

```
1.      #ifndef _VECTOR_1D_H

2.      #define _VECTOR_1D_H
        …
3.      #endif
```

**//Member Functions**

```
1.      void        Allocate (int n);
2.      void        Allocate (int *n);
3.      Int         NumberElements ( ) const
4.      T&          Data (int index);
5.      const T&    Data (int index) const;
6.      void        New (int new_n);
```

In the above code listings lines 1 and 2 define an overloaded function **Allocate( )**, which performs the task of allocating values to a vector internally.

Line 3 defines an integer function **NumberElements( )**, which performs the task of returning the number of elements in a vector.

Lines 4 and 5 define an overloaded function **Data( )** which is used as the value of elements of the vector (or Matrix). A template type T is returned, meaning that the value can be any of the C++ numeric data types.

Line 6 defines a function **New( )**,which performs the task of allocating new values to a previously defined vector.

**//Overloaded Operators**

```
1.      Vector&     operator = (const Vector& v);
2.      T&          operator [] (int index);
3.      const T&    operator [] (int index) const;
4.      Vector      operator + (const Vector& v);
5.      Vector      operator - (const Vector& v);
6.      Vector      operator * (const Vector& v);
```

Operator overloading is a powerful feature in C++ that allows standard C++ arithmetic operators to be redefined by a C++ programmer.

Line 1 above redefines the C++ assignment or equality operator as a vector equality operator. Thus we can define two vectors $\mathbf{a}$ and $\mathbf{b}$ of the same dimension and use the expression $\mathbf{b} = \mathbf{a}$; all elements in $\mathbf{b}$ are replaced by those in $\mathbf{a}$.

Lines 2 and 3 redefine the C++ array subscript symbol [] as a template vector subscript symbol. Thus we can define a vector $\mathbf{a}$, and be able to access its elements by using the expressions $x = \mathbf{a}[i]$ or $\mathbf{a}[i] = x$, where $i$ is a positive integer.

Line 4 redefines the C++ addition operator as a vector addition operator. Thus we can define vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ of the same dimension, and use the expression $\mathbf{c} = \mathbf{a} + \mathbf{b}$.

Line 5 redefines the C++ subtraction operator as a vector subtraction operator. Thus we can define vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ of the same dimension, and use the expression **c=a-b**.

Line 6 redefines the C++ multiplication operator as a vector multiplication operator. Thus we can define vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$, and determine the inner product of the vectors $(\mathbf{a} \bullet \mathbf{b}) = \mathbf{a}^{\mathrm{T}}\mathbf{b}$ by using the expression **c = a*b** in C++, the first vector on the right hand side of the equation is transposed by default (though not shown).

**//Friend Functions**

```
1.      friend ostream &operator << (ostream &s, const Vector<TYPE> &v);
2.      friend istream &operator >> (istream &s, const Vector<TYPE> &v);
```

Lines 1 and 2 above are friend functions which redefines the input (>>) and output (<<) operators as overloaded vector input and output operators. These are used to write vector data to a file with the same format as the vector dimension, and to read data directly from a file into a vector.

**2.2 Matrix Class**

A matrix (2-dimensional) is a rectangular array of numbers (or functions) enclosed in brackets [8], and can be denoted by;

$$A = [a_{ij}] = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Allowable operations on matrices include; transposition, addition, subtraction, multiplication, determinant, norm, trace, scalar multiplication, diagonalization, upper triangularisation, lower triangularisation, LU decomposition, Cholesky factorization, singular value decomposition, and inverse. Some of these matrix operations have been implemented here, together with extra algebra functions suchas Gauss elimination, zeros, matrix expansion, and functions that return the dimensions of an unknown matrix. The matrix is defined as a template header file with the name **matrix_2D.h.**

**//Class Definition**

```
1.    #ifndef _MATRIX_2D_H
2.    #define _MATRIX_2D_H
      …
3.    #endif
```

Lines 1 to 3 above form a standard C++ definition of the header file **matrix_2D.h.**

**//Member Functions**

```
1.    T&          Data (int row, int col);
2.    const T&    Data (int row, int col) const;
3.    int         Rows ( ) const {return rows;}
4.    int         Cols ( ) const {return cols;}
5.    Vector<T>   GaussElimination (Vector<T>& b, Boolean& solved);
6.    Matrix      Diag (const Matrix& m);
7.    Matrix      Ut (const Matrix& m);
8.    Matrix      Lt (const Matrix& m);
9.    Matrix      Scalarmult (const Matrix& m, T val);
10.   double      Det (const Matrix& m);
11.   double      L2Norm (const Matrix& m);
12.   double      Trace (const Matrix& m);
13.   Matrix      Inv (const Matrix& m);
14.   Matrix      Zeros( );
15.   Matrix      LuDecomp (Matrix& m, Vector<int>& indx, double& d);
16.   Matrix      LuBksub (Matrix& m, Vector<int>& indx, Vector<double>& b);
17.   Matrix      InsertRows(const Matrix& m, int& d);
18.   Matrix      InsertCols(const Matrix& m, int& d);
19.   Matrix      ReplaceRows(const Matrix& m, int& d);
20.   Matrix      ReplaceCols(const Matrix& m, int& d);
21.   Matrix      DeleteRows(int& c, int& d);
22.   Matrix      DeleteCols(int& c, int& d);
```

In the above listing, lines 1 and 2 define further overloading of the template function **Data( ).** As usual, the returned value is of the template type and the array elements must be homogeneous.

Lines 3 and 4 defines two functions **Rows( )** and **Cols( )** which returns integer values of the rows and columns of the matrix respectively. So we may obtain matrix **A** of unknown dimension (probably as a dynamic output of a process), and be able to determine its dimension. This is often required for performing array operations and for dynamically creating new matrices at run time.

Line 5 defines the function **GaussElimination( )**, which implements the Gauss elimination process and returns a vector. Consider the linear system of $m$ equations in $n$ unknowns, this can be written as a vector equation $\mathbf{Ax} = \mathbf{b}$. The solutions may be obtained by Gauss elimination of the augmented matrix $\tilde{\mathbf{A}}$.

Line 6 defines the function **Diag( )**, which creates the diagonal matrix of an existing matrix, this applies only to square matrices, a matrix is returned. The diagonal matrix of a matrix $[a_{ij}]$ is denoted by $\mathbf{diag}(\mathbf{A}) = [a_{ij}]$, where $a_{ij} = 0 \quad \forall \quad i \neq j$.

Line 7 defines a function that obtains an upper matrix $\mathbf{U}$, while line 8 defines a function that returns a lower matrix $\mathbf{L}$, of a matrix $\mathbf{A}$; such that we can write matrix $\mathbf{A}$ as a sum of the two matrices $\mathbf{L} + \mathbf{U} = \mathbf{A}$. The two matrices $\mathbf{L}$ and $\mathbf{U}$ look like triangular matrices.

Line 9 defines a function that performs scalar multiplication of a matrix, a matrix is returned. The scalar multiplication of a matrix $[a_{ij}]$ by a constant value $k$ is denoted by

$$\mathbf{A} * k = [a_{ij} * k] \quad \forall \quad ij .$$

Line 10 defines a function that returns the **determinant** of the matrix, only if the matrix is non singular. The determinant of order $n$ is a scalar associated with a $n \times n$ matrix and is defined for $n \geq 2$ by

$$D = \mathbf{det}\,\mathbf{A} = \sum_{j=1}^{n}\sum_{k=1}^{n} a_{jk} C_{jk} \text{ , where } C_{jk} = (-1)^{j+k} M_{jk} \text{ [8].}$$

Line 11 defines a function that returns the **L2 norm** (norm) of a matrix, where the **L2 norm** is the length or magnitude of a vector $\overrightarrow{v} = (a_1, a_2, \Lambda\ , a_n)$ and is given as

$$\|\overrightarrow{v}\| = \sqrt{a_1^{\,2} + a_2^{\,2} + \Lambda\ + a_n^{\,2}} .$$

Line 12 defines a function that obtains the **trace** of a matrix, a value of type double is returned. The trace of an $n \times n$ matrix is the sum of its diagonal elements, and is given as

$$\mathbf{tr}(A) = \sum_{i=1}^{n} a_{ii} .$$

Line 13 defines a function that returns the **inverse** of a matrix of any dimension. The inverse $A^{-1}$ of the matrix $A$ is derived from $A$ such that $A^{-1}A = I$. The method used in implementing the function here is follows from [3].

Line 14 defines a function that reduces a matrix to the zero matrix. A zero matrix is returned.

Line 15 defines a function that performs the **LU decomposition** of a matrix, returns the two decomposed matrices **L** and **U. LU decomposition** reduces a matrix $A$ to its upper and lower triangular matrices such that $\mathbf{L} \bullet \mathbf{U} = \mathbf{A}$, where $\mathbf{L}$ is the lower triangular matrix and $\mathbf{U}$ is the upper triangular matrix.

Line 16 defines a function that performs the **LU** back substitution operation of a matrix. The substituted matrix is returned.

Lines 17 and 18 define functions that performs row and column insertion respectively, so we can use the expressions **A=A.InsertRows(B, rNum)** or **A=A.InsertCols(B, cNum)**; where the matrix B is inserted into A row wise starting from row *rNum* or column wise starting from column *cNum*. Certain applications (e.g. neural networks [1]) require operations such as row and column *insertion, replacement,* and *deletion*.

Lines 19 and 20 define functions that performs row and column replacement respectively, so we can use the expressions **A=A.ReplaceRows(B, rNum);** or **A=A.ReplaceCols(B, cNum)**. Where the matrix **B** is used to replace rows in **A** row wise starting from row rNum; or to replace column wise starting from column cNum.

Lines 21 and 22 define functions that performs row and column deleting respectively, so we can use the expressions **A=A.DeleteRows(nRows, rNum);** or **A=A.DeleteCols(nCols, cNum)**. Where nRows rows are deleted from **A** starting from row rNum, and nCols columns are deleted from **A** starting from column cNum.

**//Overloaded Operators**

1. Matrix& operator = (const Matrix& m);
2. Vector<T>& operator [] (int index);
3. Matrix operator + (const Matrix& m);
4. Matrix operator - (const Matrix& m);

5. Matrix operator * (const Matrix& m);
6. Matrix operator ^= (const Matrix& m);
7. Matrix operator = = (const Matrix& m);
8. Matrix operator += (const Matrix& m);

Rather than implement the matrix operators as functions, here we utilise the powerful C++ capability of operator overloading. In the above code listing;

Line 1 redefines the equality operator for the matrix data type. Thus we can define matrices $\mathbf{A}$ and $\mathbf{B}$, and use the expression $\mathbf{A} = \mathbf{B}$, only if $\mathbf{A}$ and $\mathbf{B}$ are of same dimension; all elements of $\mathbf{A}$ are replaced by those in $\mathbf{B}$.

Line 2 redefines the C++ array subscript operator as a matrix subscript operator. Thus we can define a matrix $\mathbf{A}$, and be able to access its elements by using the expression $x = \mathbf{A}[i][j]$; or assign a value to an element $\mathbf{A}[i][j] = x$.

Line 3 redefines the C++ addition operator as a matrix addition operator. Thus we can define matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, and use the expression $\mathbf{C} = \mathbf{A} + \mathbf{B}$, only if the three matrices are of same dimension.

Line 4 redefines the C++ subtraction operator to be used as a matrix subtraction operator. Thus we can define matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, and use the expression $\mathbf{C} = \mathbf{A} - \mathbf{B}$, only if all matrices are of same dimension.

Line 5 redefines the C++ multiplication operator to be used as a matrix multiplication operator. Thus we can define matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, and use the expression $\mathbf{C} = \mathbf{A} * \mathbf{B}$, only if

$$[c_{ik}] = [a_{ij}] * [b_{jk}]$$

is satisfied.

Line 6 defines a new compound operator (consisting of the caret and equality sign) to be used as a matrix transposition operator. Thus we can define matrices $\mathbf{A}$, $\mathbf{B}$ and use the expression $\mathbf{B}\hat{} = \mathbf{A}$, this represents $\mathbf{B} = \mathbf{A}^T$.

Line 7 defines a new compound operator (consisting of two equality signs) to be used as a matrix row insertion operator. Some applications require that new rows be dynamically added to an existing matrix thus changing its dimension and expanding the matrix at run time. With this we can define matrices $\mathbf{A}$, $\mathbf{B}$ and use the expression $\mathbf{B} = = \mathbf{A}$ in C++, only if for $[a_{ij}]$ and $[b_{kl}]$, $j = l$. All elements of $\mathbf{A}$ are stacked below the last row of $\mathbf{B}$. The new matrix $\mathbf{B} = [b_{(i+k)l}]$.

Line 8 defines a new compound operator (consisting of a plus sign followed by an equality) to be used as a matrix column insertion operator. Some applications require that new columns be dynamically added to an existing matrix thus changing its dimension and expanding the matrix. Thus we can define matrices $\mathbf{A}$, $\mathbf{B}$ and use the expression $\mathbf{B} + = \mathbf{A}$, only if for $[a_{ij}]$ and $[b_{kl}]$, $i = k$. All elements of $\mathbf{A}$ are stacked to the right after the last column of $\mathbf{B}$. The new matrix $\mathbf{B} = [b_{i(j+l)}]$.

**//Friend Functions**

1. friend ostream &operator << (ostream &s, const Matrix<TYPE> &m);
2. friend istream &operator >> (ostream &s, const Matrix<TYPE> &m);

Lines 1 and 2 above defines C++ friend functions for overloading the input operator >> and output operator <<. The functions are the same as described in Section 2.1.

### 2.3 Matrix 3D Class

In the course of using the matrix and vector classes, one inadvertently faces a situation where a process requires building an array of matrices over time, a one-dimensional array of two–dimensional matrices thus results in a three-dimensional matrix. Manipulating data in 3D is the basis for many scientific applications including computer graphics, vision, geometric modelling, fluid and rigid body navigation, and many other areas, it is therefore necessary to include a 3D matrix class. Here we simply consider it as an array of two dimensional matrices of the same dimension; this can be denoted as;

$$\mathbf{A} = [a_{ijk}] = \begin{bmatrix} a_{111} & a_{121} & \cdots & a_{1n1} \\ a_{211} & a_{221} & \cdots & a_{2n1} \\ \cdot & \cdot & \cdots & \cdot \\ a_{m11} & a_{m21} & \cdots & a_{mn1} \end{bmatrix} \begin{matrix} a_{2n2} \\ \cdot \\ a_{mn2} \end{matrix} \begin{matrix} \cdots \begin{bmatrix} a_{11l} & a_{12l} & \cdots & a_{1nl} \\ a_{112} & a_{122} & \cdots & a_{1n2} \end{bmatrix} a_{2nl} \\ \cdot \\ a_{mnl} \end{matrix}$$

Although mathematical operations are basically defined for two dimensional matrices, some of them can be applied directly to three dimensional matrices (e.g. matrix addition); while others can be applied to the 2D matrix corresponding to the first two digits of the subscript array. The file is defined as a template header file by the name **matrix_3D.h.**

**//Class Definition**

1. ifndef _MATRIX_3D_H
2. #define _MATRIX_3D_H

   **…**
3. endif

Lines 1 to 3 defines **matrix_3D.h** in C++.

**//Member Functions**

1. Matrix3D();
2. Matrix3D(int n, int m, int k);
3. inline T** operator[](const int i);
4. inline const T* const * operator[](const int i) const;
5. inline int dimX() const;
6. inline int dimY() const;
7. inline int dimZ() const;

In the code listings above, lines 1 and 2 define an overloaded Matrix3D function for defining a three dimensional matrix.

Lines 3 and 4 define the operator [] (with a double pointer) as an overloaded operator for three dimensional matrix subscripts.

Lines 5 to 7 define three inline functions that are used to return the dimensions of the matrix as constant integer values.

**//Friend Functions**

1. friend ostream &operator << (ostream &s, const Matrix3D<TYPE> &m);
2. friend istream &operator >> (istream &s, const Matrix3D<TYPE> &m);

The two lines in the code listings above are used to perform same functions as described for the matrix class. The three classes that have been explained in this section make up the complete implementation of the template matrix and vector classes.

## 3.0 Experimental results

To evaluate the efficiency of the template classes, a test program was written as a single C++ source code file to enable us automatically test most of the implemented functionalities.

### 3.1 Using the classes

To use the classes, the three header files are included to a C++ project. Remember to include them into any source file where they will be used "vector_1D.h" is already included into "matrix_2D.h" and thus may be omitted if "matrix_2D.h" is already included; The test file *TemplateFilesTest.cpp* produces the output described in 3.2.

### 3.2 Results

This section contains the experimental results for running the test program *TemplateFilesTest.cpp*.

```
Running Automated Tests...
Matrix & Vector Definitions...
Vector<int> V(3, 0);



Matrix<double> A(3,3,0);
Matrix<double> B(3,3,0);
Matrix<double> C(3,3,0);
Matrix<int> Y(3,2,8);
Matrix<float> Z(3,4,0.5);
Matrix3D<double> H(2, 3, 4);
...Done

Initialisations...
V
[    2,     4,     1]
A
[   1.012126     5.012126     2.034544
    0.013226     9.013226     4.015556
    0.341612     3.341612     6.781612
]
B
[   0.512126     2.452166     1.542126
    6.013226     3.516526     2.993226
    0.341612     7.441612     8.781612
]
... Done...

Testing Matrix Addition...
Addition: C=A+B
[   1.524252     7.464292     3.576670
    6.026452    12.529752     7.008782
    0.683224    10.783224    15.563224
]

Testing Matrix Subtraction...
Subtraction: C=A-B
[   0.500000     2.559960     0.492418
   -6.000000     5.496700     1.022330
```

```
    0.000000     -4.100000     -2.000000
]

Testing Matrix Multiplication...
Multiplication: C=A*B
[   31.352407     35.247459     34.429828
    55.577100     61.609886     62.262073
    22.585497     63.054680     70.082494
]

Testing Matrix Transpose...
Transpose: C^=A
A
[    1.012126      5.012126      2.034544
     0.013226      9.013226      4.015556
     0.341612      3.341612      6.781612
]

C
[    1.012126      0.013226      0.341612
     5.012126      9.013226      3.341612
     2.034544      4.015556      6.781612

]

Testing Matrix Determinant...
Determinant: double determinant =C.Det (C)=48.535645

Testing Matrix Trace...
Trace: double trace =C.Trace (C)=16.806964

Testing Matrix Norm...
Norm: double norm=C.L2Norm (C)=13.598589

Testing Matrix Diagonal...
Diagonal Matrix G=G.Diag (C);

[    1.012126      0.000000      0.000000
     0.000000      9.013226      0.000000
     0.000000      0.000000      6.781612
]

Testing Matrix Scalar Multiplication...
Scalar multiplication: H=H.Scalarmult(C, 2.5)
C
[    1.012126      0.013226      0.341612
     5.012126      9.013226      3.341612
     2.034544      4.015556      6.781612
```

```
]

H
[    2.530315     0.033065     0.854030
    12.530315    22.533065     8.354030
     5.086360    10.038890    16.954030
]

Testing Matrix Triangularisation...
Trianguarisation: of C; E=E.Lt(C); F=F.Ut(C);
C
[    1.012126     0.013226     0.341612
     5.012126     9.013226     3.341612
     2.034544     4.015556     6.781612
]

E=Lt(C)
[    0.000000     0.000000     0.000000
     5.012126     0.000000     0.000000
     2.034544     4.015556     0.000000
]

F=Ut(C)
[    0.000000     0.013226     0.341612
     0.000000     0.000000     3.341612
     0.000000     0.000000     0.000000
]

Testing Matrix LU Decomposition...
LU Decomposition: of C;
C
[    1.012126     0.013226     0.341612
     5.012126     9.013226     3.341612
     2.034544     4.015556     6.781612
]

C=C.LuDecomp (C, E, d)

[    1.012126     0.013226     0.341612
     4.952077     8.947730     1.649923
     2.010169     0.445808     5.359365
]

Testing Matrix Inverse...
Inverse: K=K.Inv(C)
C
[    1.012126     0.013226     0.341612
     4.952077     8.947730     1.649923
     2.010169     0.445808     5.359365
```

```
]

K
[    1.121747      0.001934     -0.072097
    -0.551705      0.112550      0.000517
    -0.374848     -0.010088      0.213588
]

Testing Dynamic Matrix Creation...
Dynamic Matrix Creation
D
41 18467 6334 26500
19169 15724 11478 29358
26962 24464 5705 28145
23281 16827 9961 491

Testing Elastic Matrices...
C
[    1.012126      0.013226      0.341612
     4.952077      8.947730      1.649923
     2.010169      0.445808      5.359365
]

E
[ 2995.000000  11942.000000
  4827.000000   5436.000000
 32391.000000  14604.000000
]



F
[ 3902.000000    153.000000     292.000000
 12382.000000   17421.000000  18716.000000
]

Row insertion of F into C: C==F
[    1.012126      0.013226       0.341612
     4.952077      8.947730       1.649923
     2.010169      0.445808       5.359365
  3902.000000    153.000000     292.000000
 12382.000000   17421.000000  18716.000000
]

Column insertion of E into C: C+=E
[    1.012126      0.013226      0.341612   2995.000000   11942.000000
     4.952077      8.947730      1.649923   4827.000000    5436.000000
     2.010169      0.445808      5.359365   32391.000000  14604.000000
]

Testing Gauss Elimination...
A
[    3.000000      2.000000      1.000000
     3.000000      5.000000      7.000000
```

```
    4.000000      9.000000      2.000000
]

b
[2.000000, 4.000000, 1.000000]

Solution Ax = b; x = b.Inv(A)
[0.712963, -0.314815, 0.490741]
Done.
```

## 4.0 Conclusion and future work

In this paper, we presented a set of template C++ matrix and vector data structures developed as building blocks for programmers and researchers developing scientific and other mathematical systems, where matrices and vector algebras are applied. The rationale for developing the classes is to help C++ programmers to surmount some difficulties arising from the use of C++ arrays at run time, to present a set of simple building blocks for the C++ developer, and to help new C++ developers learn the methods of building reusable template classes in C++. Experimental tests show for efficiency, accuracy and robustness. The classes had previously been used by the author to implement the Extended Kalman filter [12] navigation algorithm on a mobility robot platform running Linux. Since these classes are building blocks for matrix involved software, any matrix operations which are required can be added or developed by using them. The classes are available at http://www.nampjournals.org. Users can freely experiment with them, make extensions to them, make observations and send feedback. The classes can be compiled with any ANSI compliant version of C++; the recent test run was done using Microsoft Visual C++ 2005 Express Edition. A lot of future research work can continue from this foundation, for example the concept of three-dimensional matrix algebras is unpopular; having this tool can help people perform simulated experimental work on that. Some of the capabilities and the flexibility of the classes described here are not available in the Java programming language. For example, the operator overloading capability used in the classes gives me the ability to write out the Kalman filter equations (and other matrix equations) in the C++ programs, almost exactly the way they are written on paper. This is one area I found C++ more beautiful than Java, and for this reason I hope to make the classes work in Java programming.

### *References*


[1] Rogers, J., (1997), Object-Oriented Neural Networks in C++, Academic Press Inc. London.

[2] Seed, G., (2002), Object Oriented Programming in C++, With Applications to Computer Graphics, Springer Verlag, London.

[3] Press, W. H., Teukolsky, S. A., Vetterling, W. T., Flannery, B. P., (2002), Numerical Recipes in C++, Cambridge University Press, Cambridge, M.A.

[4] Stroustrup B, (1997), The C++ Programming Language, 3rd ed., Addison-Wesley, Reading, M.A.

[5] Lumsdaine, A., and Seik, J. (1998), "The Matrix Template Library", Available http://www.lsc.nd.edu/researh/mtl.

[6] Anderson, E. et al., (2000) LAPACK User's Guide, 3rd ed. SIAM, Philadelphia.

[7] Dongarra, J., Bunch, J., Moler, C. and Stewart, G. (1979), Linpack Users Guide, SIAM, Philadelphia.

[8] Kreyszig, E., (1999), Advanced Engineering Mathematics, John Wiley and Sons, N.Y.

[9] Kwak, J. H., and Hong, S. (2004), Linear Algebra, Bikhauser Boston.

[10] Roman, S., (2005), Advanced Linear Algebra, Springer Science+Business Media Inc. N.Y. 10013.

[11] Bar-Shalom, Y., Rong Li, X., Kirubarajan, T., (2001), Estimation with Applications to Tracking and Navigation, John Wiley and Sons, N.Y.