Chapter V

Building Classes

A class embodies a set of variables and a set of functions that may operate on the variables in the class. These variables and functions are referred to as members of that class. Variables in a class are usually made private and hence can be accessed only by member functions of the class. Member functions are usually made public and can be accessed from anywhere outside the class. A class, same as a structure, is a template that can be used only by declaring an instance of that class. This is similar to declaring an instance of a built-in type such as int, float, etc. An instance of a built-in type is called a variable, an instance of a class is called an "object" and hence the phrase "object-oriented programming". With the use of classes we will be able to build new variable types and define operations that can be carried-out on the new type. For example a matrix can be defined as a class and the various operations on matrices such as addition, multiplication, inverse, transpose, etc can be defined by member functions. The use of classes makes the development of large projects less prone to errors. The project can be divided into a set classes each with its own set of responsibilities. Each class can then be developed by one or more programmers and tested independently. Once each class is guaranteed to work, the larger project can then be assembled from the individual classes. This form of object oriented programming is the latest trend in the advancement of paradigms for efficient programming that gives the developers a better chance at producing almost bug free routines. Unfortunately, C++ is platform dependent and hence a program developed for one operating system may not be suitable for another operating system. This does not of course apply to the type of programs we have developed so far, since these are console type applications that can be recompiled and run on other systems. What we are referring to here are programs that can run without recompilation on other systems via the Internet and web browsers rather than the operating system. C++ is not geared for that. It is environment (operating system) specific. A new language called Java developed by Sun Microsystems which is also object oriented and shares a lot of commonalities with C++ is multi-platform. Such a language is referred to as "develop once deploy anywhere." Java's popularity is growing rapidly and may eventually be the dominant language around. Does this make our study of C++ futile? Not in the least, knowing C++ will make the transition to Java relatively easy. Also since no one can predict the future with any accuracy it is always best to be equipped with more than one development tool.

5.1 Declaring a Class

A class can be declared as follows:

```
class ClassName
      {
  private:
        private members;
        ...
  public:
```

```
    public members;
    ...
     };
```

Apart from private and public members we can also include **protected** members. Protected members are accessible to member functions of the class as well as descendent classes. Descendent classes are classes derived from a (base) class and have full access to public as well as protected members of the base class. The subject of descendent classes has to do with the inheritance properties of classes and is discussed in Chapter 12.

The following example develops a simple class for complex variables.

```
#include <iostream.h>
#include <conio.h>

class _Complex
  {
  private:
      double Real;
      double Imag;
  public:
      _Complex()      //Constructor
        {
        assign();
        }
      void assign(double realValue=0, double imagValue=0);

      double getReal()
        {
        return Real;
        }

      double getImag()
        {
        return Imag;
        }

      void add(_Complex &c1, _Complex &c2);
      void print();
      };

//Defining member function add
```

```
void _Complex::assign(double RealValue, double ImagValue)
 {
 Real = RealValue;
 Imag = ImagValue;
 }

void _Complex::add(_Complex &c1, _Complex &c2)
 {
 Real = c1.Real + c2.Real;
 Imag = c1.Imag + c2.Imag;
 }

void _Complex::print()
 {
 if(Imag >= 0)
   cout << Real << "+i" << Imag;
 else
   cout << Real << "-i" << -Imag;
 cout << endl;
 }

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

int main()
{
_Complex c1, c2, c3;  //declaring objects of _Complex

c1.assign(2,3);
c2.assign(4,5);
c3.add(c1,c2);
c1.print();
c2.print();
c3.print();
getch();

 return 1;
  }
```

In the previous program we can notice the following:

- All member variables are declared private and hence are only accessible by member functions.
- Some member functions are defined within the class others are declared within the class and defined outside the class. A member function defined inside a class is

considered an inline function. The compiler will only allow small functions to be inline and hence if the function is too large to be inline you should define it outside the class, otherwise the compiler may issue a warning message. In any case the qualifier inline is only a suggestion to the compiler and the compiler can ignore it if the function is too large to be considered inline.

- One member function that is automatically invoked upon declaring an object of the function is the constructor member function. This member function is used to initialize the variables. It has the same name as that of the class.
- The scope resolution operator :: is used to define functions outside the class. The syntax is as follows: return type ClassName::Function(<Arguments>)
- The use of the underscore prior to the name of the class is for the purpose of avoiding a clash between a built-in library class that could contain the same named class. Some compilers such as Borland's (renamed as Inprise) C++ Builder contains a class for manipulating complex numbers.
- A class is used by declaring objects of a class same as is done with built-in types.
- Public member functions are accessed using the dot operator. If an object of a class is declared as a pointer then **new** should be used to allocate memory to the class and the **->** operator to access public members.
- Same as a constructor is needed to initiate variables by maybe allocating memory and values a destructor is needed to deallocate memory once the instance of the class is no longer needed. Destructors carry the same name as the class but preceded by a tilde ~.

We will provide more details on constructors.

5.2 Constructors

A constructor may have an argument list. A special type of constructor called a copy constructor has an argument which is a reference to an object of the same class. A copy constructor allows you to create an instance of the structure using an existing instance. The syntax for a class is:

```
class ClassName
    {
  public:
      ClassName( ); //constructor
      ClassName( ClassName &c); // copy constructor
      ClassName(<parameter list>); //another constructor
            ...
      };
```

As example consider the following code segment:

```
class _Complex
  {
  private:
      double Real;
      double Imag;
  public:
      void assign(double realValue=0, double imagValue=0);
      _Complex(Complex &c);
     ...
         };
```

Constructor rules

- The name of the constructor must be identical to the name of the class.
- You must not include a return type not even a void.
- A class can have any number of constructors including none. If no constructors are included then the compiler will generate one.
- The declaration of a class instance always involves a constructor for example:

```
_Complex c1;            // invokes the 1st constructor using default values.
_Complex c2(1.1, 3.7);      // uses 1st constructor in the above segment
_Complex c3(c2);            // uses the copy constructor
```

Now we use these basics in developing a program that handles complex numbers through a class _Complex.

```
#include <iostream.h>
#include <conio.h>

class _Complex
  {
  private:
      double Real;
      double Imag;
  public:
      _Complex(double realValue=0, double imagValue=0)
         {
         Real = realValue;
         Imag = imagValue;
         }

      _Complex(_Complex &c)
         {
```

```
        Real = c.Real;
        Imag = c.Imag;
        }
    void assign(_Complex &c)
      {
      Real = c.Real;
      Imag = c.Imag;
      }

    double getReal() const
      {
      return Real;
      }

    double getImag() const
      {
      return Imag;
      }

    void add(_Complex &c1, _Complex &c2);
    void print();
    };

//Defining member function add

void _Complex::add(_Complex &c1, _Complex &c2)
  {
  Real = c1.Real + c2.Real;
  Imag = c1.Imag + c2.Imag;
  }

void _Complex::print()
  {
  if(Imag >= 0)
     cout << Real << "+i" << Imag;
  else
     cout << Real << "-i" << -Imag;
  cout << endl;
  }

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

int main()
```

```
{
_Complex a(4.0, 9.0);
_Complex b(a);
_Complex c;

c.add(a,b);
c.print();

getch();

 return 1;
  }
```

In the above program we chose to declare the member functions getReal( ) and getImag( ) with a **const** modifier. A member function that is declared with the const modifier cannot modify data members of the object, nor can it call any non-const member functions. Objects declared as const, such as **const** *_Complex a(3.0,4.0)*; can only be accessed by const member functions. Hence, it is always best to declare member functions that do not modify member variables as const. This allows users of the class to declare constant objects.

5.3 Destructors

A destructor automatically removes an instance of a class once it is out of scope. The compiler automatically generates a destructor if the programmer has not defined one. However, it is always best that a constructor be defined by the programmer especially in the case when memory has to be deallocated. The following are the set rules governing the design of destructors:

- The name of the destructor must begin with the tilde character (~). The rest of the destructor name must be identical to the name of its class.
- No return types are allowed not even a void.
- A class can have no more than one destructor. If one is not included the compiler automatically creates one.
- The destructor cannot have any parameters.
- The runtime system automatically invokes a class destructor when the instance of the class is out of scope.

The following program defines a class of type Matrix and demonstrates the use of constructor and destructor for memory allocation and deallocation, respectively.

```
#include <iostream.h>
#include <conio.h>

class Matrix
  {
  private:
```

```
      double **dataPtr; //Pointer to 2-D array
      double NR;    //# of rows
      double NC;    //# of columns
  public:
      Matrix(unsigned Nrows, unsigned Ncolumns); //constructor for
                                        //memory allocation
      Matrix(Matrix &A); //copy constructor
      ~Matrix();  //destructor

      void Store(double x, unsigned i, unsigned j)
        {
        dataPtr[i][j]=x;
        }

      void recallSize(unsigned &nr, unsigned &nc)
        {
        nr=NR;
        nc=NC;
        }

      double recall(unsigned i, unsigned j)
       {
       return dataPtr[i][j];
       }

      void copy(Matrix &A);

      };

//Defining member functions

Matrix::Matrix(unsigned Nrows, unsigned Ncolumns)
 {
 NR=Nrows;
 NC=Ncolumns;

 //Allocating memory
 dataPtr = new double *[NR];

 for(int i=0; i<NR; i++)
   dataPtr[i]=new double[NC];
   }
```

```
Matrix::Matrix(Matrix &A)
 {
 int i, j;
 NR=A.NR;
 NC=A.NC;

 dataPtr = new double *[NR];
 for(i=0; i<NR; i++)
   dataPtr[i] = new double[NC];

 for(i=0; i<NR; i++)
   for(j=0; j<NC; j++)
      dataPtr[i][j] = A.dataPtr[i][j];
      }

 void Matrix::copy(Matrix &A)
  {
  //delete the current matrix
  int i, j;

  for(i=0; i<NR; i++)
     delete [] dataPtr[i];

  //recreate array with size of matrix A
  NR=A.NR;
  NC=A.NC;

  dataPtr = new double * [NR];

  for(i=0; i<NR; i++)
   dataPtr[i] = new double[NC];

 //Copy
   for(i=0; i<NR; i++)
     for(j=0; j<NC; j++)
      dataPtr[i][j] = A.dataPtr[i][j];
      }

//Destructor
Matrix::~Matrix()
 {
 for(int i=0; i<NR; i++)
    delete [] dataPtr[i];
```

```
 delete [ ] dataPtr;
 }


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

int main()
{
double A[2][3] = { {1,2,3},
                   {4,5,6}};
Matrix B(2,3);
unsigned i,j;

for(i=0; i<2; i++)
 for(j=0; j<3; j++)
   B.Store(A[i][j],i,j);

for(i=0; i<2; i++)
  {
  cout << endl;
  for(j=0; j<3; j++)
   cout << B.recall(i,j) << " ";
   }

cout << endl;

//Recall size
B.recallSize(i,j);
cout << "Size of matrix: " << i << " * " << j;

 getch();
 return 1;
   }
```

Certain functions are usually necessary in the successful design of a class:

- Setter functions to set values to private variables.
- Getter functions to obtain the values of private variables.
- Constructors and Destructors.
- Functions for manipulation of private variables

In the above program we have the following setter function:

```
void Store(double x, unsigned i, unsigned j);  // To set the value x at (i,j)
```

and the following getter functions:

```
void recallSize(unsigned &nr, unsigned &nc); //To recall size of matrix
double recall(unsigned i, unsigned j); //To recall a value from location (i,j)
```

The constructor:

```
Matrix(unsigned Nrows, unsigned Ncolumns); //constructor for
                                           //memory allocation
```

sets the number of rows and columns and allocates memory.

Before we end this chapter we will return to the Gauss-Jordan's method but this time for matrix inversion.

5.4 Matrix Inversion Using Gauss-Jordan Method

Matrix inversion is basically equivalent to solving N sets of simultaneous equations, where N is the number of rows or columns in the matrix, having the same coefficients as the matrix except the right-hand side is different for each set as follows for a 3×3 matrix:

$$AX_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad AX_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad AX_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The simultaneous solution for the above will be as follows:

$$A\begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix} = I$$

where ***I*** is the identity matrix. It is obvious then the solution of these equations should yield the matrix inverse:

$$\begin{bmatrix} X_1 & X_2 & X_3 \end{bmatrix} = A^{-1}$$

The Gauss-Jordan method can be followed by first augmenting the matrix with the identity matrix as shown in the following numerical example:

Example. Obtain the inversion of

$$A = \begin{bmatrix} 3 & -6 & 7 \\ 9 & 0 & -5 \\ 5 & -8 & 6 \end{bmatrix}$$

Solution.
Form the augmented matrix

$$\left[\begin{array}{ccc|ccc} 3 & -6 & 7 & 1 & 0 & 0 \\ 9 & 0 & -5 & 0 & 1 & 0 \\ 5 & -8 & 6 & 0 & 0 & 1 \end{array}\right]$$

Pass #1: j=1

(i) BIG = 9, L=2.
(ii) Is BIG < 10^{-7}? No. $\therefore$ continue.
(iii) switch rows 1 and 2

$$\left[\begin{array}{ccc|ccc} 9 & 0 & -5 & 0 & 1 & 0 \\ 3 & -6 & 7 & 1 & 0 & 0 \\ 5 & -8 & 6 & 0 & 0 & 1 \end{array}\right]$$

(iv) Normalization:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -\frac{5}{9} & 0 & \frac{1}{9} & 0 \\ 3 & -6 & 7 & 1 & 0 & 0 \\ 5 & -8 & 6 & 0 & 0 & 1 \end{array}\right]$$

(v) Elimination:

$$\left[\begin{array}{ccc|ccc} 1 & 0 & \frac{-5}{9} & 0 & \frac{1}{9} & 0 \\ 0 & -6-3(0) & 7-3(\frac{-5}{9}) & 1 & 0-3(\frac{1}{9}) & 0 \\ 0 & -8-5(0) & 6-5(\frac{-5}{9}) & 0 & 0-5(\frac{1}{9}) & 1-5(0) \end{array}\right]$$

$\therefore$

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -0.556 & 0 & 0.11 & 0 \\ 0 & -6 & 8.668 & 1 & -0.333 & 0 \\ 0 & -8 & 8.780 & 0 & -0.555 & 1 \end{array}\right]$$

Pass # 2: j=2
(i) BIG=8, L=3
(ii) Is BIG < 10^{-7}? No. ∴ continue.
(iii) Switch j and L rows

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -0.556 & 0 & 0.11 & 0 \\ 0 & -8 & 8.780 & 0 & -0.555 & 1 \\ 0 & -6 & 8.668 & 1 & -0.333 & 0 \end{array}\right]$$

(iv) Normalization

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -0.556 & 0 & 0.11 & 0 \\ 0 & 1 & -1.0975 & 0 & -0.069 & -0.125 \\ 0 & -6 & 8.668 & 1 & -0.333 & 0 \end{array}\right]$$

(v) Elimination

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -0.556 & 0 & 0.11 & 0 \\ 0 & 1 & -1.0975 & 0 & -0.069 & -0.125 \\ 0 & 0 & 2.083 & 1 & 0.081 & -0.75 \end{array}\right]$$

Pass # 3: j=3
(i)BIG = 2.083
(ii) Is BIG < 10^{-7}? No. ∴ continue.
(iii) Skip row switching
(iv) Normalization

$$\left[\begin{array}{ccc|ccc} 1 & 0 & -0.556 & 0 & 0.11 & 0 \\ 0 & 1 & -1.0975 & 0 & -0.069 & -0.125 \\ 0 & 0 & 1 & 0.48 & 0.039 & -0.36 \end{array}\right]$$

(v) Elimination

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & 0.267 & 0.131 & -0.2 \\ 0 & 1 & 0 & 0.52 & 0.112 & -0.52 \\ 0 & 0 & 1 & \underbrace{0.48 & 0.039 & -0.36}_{A^{-1}} \end{array}\right]$$

Example. Develop a C++ program for solving linear simultaneous equations and for matrix inversion by including two functions in the class Matrix developed in this chapter. The two functions should have the prototypes:

```
int simq(unsigned Nequations);   // Nequations = number of equations
int inv(unsigned n);   // n = # of rows or columns
```

Solution.

The following is a code extending the class Matrix to include the two functions specified in the problem along with test examples.

```
#include <iostream.h>
#include <conio.h>
#include <math.h>

class Matrix
  {
  private:
     double **dataPtr; //Pointer to 2-D array
     double NR;    //# of rows
     double NC;    //# of columns
  public:
     Matrix(unsigned Nrows, unsigned Ncolumns); //constructor for
                                    //memory allocation
     Matrix(Matrix &A); //copy constructor
     ~Matrix();  //destructor

     void Store(double x, unsigned i, unsigned j)
       {
       dataPtr[i][j]=x;
       }

     void recallSize(unsigned &nr, unsigned &nc)
```

```
 {
 nr=NR;
 nc=NC;
 }

double recall(unsigned i, unsigned j)
 {
 return dataPtr[i][j];
 }

void copy(Matrix &A);
void PrintMatrix();

int simq(unsigned Nequations); // Solution of simultaneous equations
int inv(unsigned n); //Matrix Inversion

};

//Defining member functions

Matrix::Matrix(unsigned Nrows, unsigned Ncolumns)
 {
 NR=Nrows;
 NC=Ncolumns;

 //Allocating memory
 dataPtr = new double *[NR];

 for(int i=0; i<NR; i++)
  dataPtr[i]=new double[NC];
  }

Matrix::Matrix(Matrix &A)
 {
 int i, j;
 NR=A.NR;
 NC=A.NC;

 dataPtr = new double *[NR];
 for(i=0; i<NR; i++)
  dataPtr[i] = new double[NC];

 for(i=0; i<NR; i++)
```

```
  for(j=0; j<NC; j++)
     dataPtr[i][j] = A.dataPtr[i][j];
     }

 void Matrix::copy(Matrix &A)
 {
 //delete the current matrix
 int i, j;

 for(i=0; i<NR; i++)
    delete [] dataPtr[i];

 //recreate array with size of matrix A
 NR=A.NR;
 NC=A.NC;

 dataPtr = new double * [NR];

 for(i=0; i<NR; i++)
  dataPtr[i] = new double[NC];

 //Copy
  for(i=0; i<NR; i++)
    for(j=0; j<NC; j++)
     dataPtr[i][j] = A.dataPtr[i][j];
     }

//Print Matrix
void Matrix::PrintMatrix()
 {
 int i,j;
 for(i=0; i<NR; i++)
   {
   cout << endl;
   for(j=0; j<NC; j++)
     cout << dataPtr[i][j] << "  ";
     }
   cout << endl;
   }

//Destructor
Matrix::~Matrix()
 {
```

```
 for(int i=0; i<NR; i++)
    delete [] dataPtr[i];

 delete [ ] dataPtr;
 }

int Matrix::simq(unsigned Nequations)
{
 double big, *temp;
 unsigned i,j,k,L;

 NR-Nequations;
 NC=NR+1;

 for(j=0; j<NR; j++)
   {
//Search for big
    big=fabs(dataPtr[j][j]);
    L=j;
   for(i=j; i<NR; i++)
    {
    if(big<fabs(dataPtr[i][j]))
     {
     big=fabs(dataPtr[i][j]);
     L=i;
     }
    }
//Check big
    if(big < 1.e-7) return 0; //No unique solution

// exchange rows
   if(j!=L)
     {
     temp=dataPtr[L];
     dataPtr[L]=dataPtr[j];
     dataPtr[j]=temp;
     }

// Normalization
    for(i=j+1; i<NC; i++)
      dataPtr[j][i]/=dataPtr[j][j];
     dataPtr[j][j]=1.0;
```

```
// Elimination
    for(k=0; k<NR; k++)
     {
     if(k==j) continue;
     for(i=j+1; i<NC; i++)
       dataPtr[k][i]-=dataPtr[j][i]*dataPtr[k][j];
     dataPtr[k][j]=0.0;
     }
    }
  return 1;  //a solution was found
  }

// Matrix Inversion
int Matrix::inv(unsigned n)
   {
 double big, *temp;
 unsigned i,j,k,L;

 NR=n;
 NC=n<<1;

 for(j=0; j<NR; j++)
   {
//Search for big
    big=fabs(dataPtr[j][j]);
    L=j;
   for(i=j; i<NR; i++)
    {
    if(big<fabs(dataPtr[i][j]))
     {
     big=fabs(dataPtr[i][j]);
     L=i;
     }
    }
//Check big
    if(big < 1.e-7)
      return 0;//No unique solution

// exchange rows
   if(j!=L)
     {
     temp=dataPtr[L];
     dataPtr[L]=dataPtr[j];
```

```
     dataPtr[j]=temp;
     }

// Normalization
    for(i=j+1; i<NC; i++)
      dataPtr[j][i]/=dataPtr[j][j];
     dataPtr[j][j]=1.0;

// Elimination
    for(k=0; k<NR; k++)
     {
     if(k==j) continue;
     for(i=j+1; i<NC; i++)
       dataPtr[k][i]-=dataPtr[j][i]*dataPtr[k][j];
     dataPtr[k][j]=0.0;
     }
    }
   return 1;//a solution was found
     }

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

int main()
{
cout << "Solving linear equations: " << endl;
double A[3][4] = {{3, 4, 6, 9},
                  {7, 9, 10, 2},
                  {1, 2, 5, 7}};
Matrix B(3,4);
int i,j;

for(i=0; i<3; i++)
 for(j=0; j<4; j++)
   B.Store(A[i][j],i,j);

if(!B.simq(3))
   cout << "No unique solution. " << endl;
else
  {
 cout << endl << "Solution: " << endl;
 for(i=0; i<3; i++)
    {
    cout << endl;
```

```
 for(j=0; j<4; j++)
   cout << B.recall(i,j) << "  ";
    }
 cout << endl;

 cout << "Checking residues: " << endl;
 double sum;

 for(i=0; i<3; i++)
  {
  sum=0.0;
  for(j=0; j<3; j++)
    sum+=A[i][j]*B.recall(j,3);
  cout << (sum - A[i][3]) << endl;
  }
 }
cout << endl;

cout << "Matrix inversion " << endl;
double M[3][3] = {{3,4,6},
                  {7,9,10},
                  {1,2,5}};

Matrix MI(3,6);

for(i=0; i<3; i++)
 for(j=0; j<3; j++)
   MI.Store(M[i][j],i,j);
//Augment the identity matrix
for(i=0; i<3; i++)
 for(j=3; j<6; j++)
   {
  if((i+3)==j)
   MI.Store(1.0,i,j);
  else
   MI.Store(0.0,i,j);
   }
 cout << "Augmented Matrix: " << endl;
 MI.PrintMatrix();

 if(!MI.inv(3))
   cout << "No unique solution. " << endl;
 else
```

```
 {
 cout << "Solution " << endl;
for(i=0; i<3; i++)
 {
 cout << endl;
 for(j=3; j<6; j++)
  cout << MI.recall(i,j) << " ";
 }

cout << endl;

cout << "X=Inverted Matrix * B : " << endl;
double x[3];
cout << endl;
for(i=0; i<3; i++)
 {
 x[i]=0;
 for(j=3; j<6; j++)
   x[i]+=MI.recall(i,j)*A[j-3][3];
 cout << x[i] << " ";
 }
 cout << endl;
 }
getch();
return 1;
 }
```

A few explanations are in order.

- Both simq and inv return a zero if no solution is found.
- Row switching is carried-out by switching row pointers as follows:

```
if(j!=L)
    {
 temp=dataPtr[L];
 dataPtr[L]=dataPtr[j];
 dataPtr[j]=temp;
    }
```

This is a much more efficient method than switching element by element and is the preferred approach in C++.

- In the main program we took an example which we have worked on manually in previous chapter. After obtaining the solution we calculated the residues by the substituting the solution in the three equations and subtracting the right-hand side.
- For matrix inversion we use the coefficient matrix of the three equations and check the answer by multiplying the matrix inverse by the right-hand side of the equations.

Alternatively, one could have multiplied the matrix with its inverse and compared with the identity matrix. The subtraction of the identity matrix from the matrix multiplied by its inverse yields the residue matrix. Residues are usually due to round-off errors arising from the precision of the type used (long, double, etc.) These residues can be used to determine if further refinement on the solution is required. There are many error refinement schemes available in the literature, however we will leave that for the readers to research.

Problems

1. Develop a class for handling 3D Cartesian coordinates (x,y,z). The class should contain a member function for obtaining the distance between two points in space.

2. Solve the following system of linear equations using the Gauss-Jordan method with maximum pivoting:

$$\begin{aligned} z_1 &+ z_2 + z_3 = 0 \\ (1-i)z_1 &+ z_2 = i \\ (3-i)z_1 &+ 2z_2 + z_3 = 1+2i \end{aligned}$$

3. Develop a C++ class for handling complex matrices. The class should contain a member function for solving a set of linear equations with complex coefficients. Test the class on the numerical example given in the previous problem.

4. Reduce the following matrix to an upper triangular form by reducing the elements below the diagonal to zeros using similar steps to the Gauss-Jordan method.

$$\begin{bmatrix} 3 & 8 & -18 \\ 1 & 2 & -4 \\ 1 & 3 & -7 \end{bmatrix}$$

5. Include in the class Matrix, developed in section 5.3, member functions for transforming matrices into upper or lower triangular form. A lower triangular form has its elements above the diagonal equal to zero.

6. By hand calculation obtain the inversion of the following matrix:

$$\begin{bmatrix} 3 & -2 & -1 \\ -4 & 1 & -1 \\ 2 & 0 & 1 \end{bmatrix}$$

Determine the matrix inversion of the above example by running a C++ program that utilizes the class Matrix.

7. The exponent of a matrix can be obtained using the infinite series:

$$e^A = I + A + \frac{1}{2!}A^2 + \frac{1}{3!}A^3 + \frac{1}{4!}A^4 + \ldots + \frac{1}{k!}A^k + \ldots$$

This is a convergent series same as the series we used previously to obtain the exponent of a scalar value (a single value). Include a member function for the exponent of a matrix in class Matrix developed in this chapter. Develop the necessary code for this member and test it on a numerical example using the matrix in problem 6.

8. The solution of a set of differential equations placed in matrix form:

$$\dot{X}(t) = AX(t)$$

is given by

$$X(t) = e^{At}X(0)$$

Obtain the values of X at t=0.1,0.2,03 for the following set of equations:

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = 5 - 3.0x_1 - 2.0x_2$$

Given X(0)={1.0, 0.0}. Use the code you have developed in question 7.