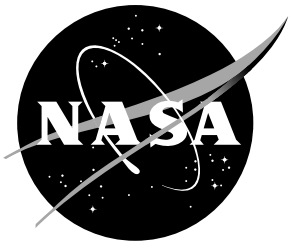
NASA Contractor Report 201592



# CLMNANAL: A C++ Program for Application of the Coleman Stability Analysis to Rotorcraft

Michael B. Lance
*Lockheed Martin Engineering & Sciences Company*
*Hampton, Virginia*

Contract NAS1-19000

August 1996

National Aeronautics and
Space Administration
Langley Research Center
Hampton, Virginia 23681-0001

# Summary

The theory of the self-excited instability of hinged rotor blades, commonly known as ground resonance, was worked out by Robert P. Coleman and Arnold M. Feingold between the years of 1942 and 1947. Two Restricted Reports and a Technical Note were released describing the theoretical analysis of ground resonance, and the methodology for the application of the theory to rotorcraft (reference 1). Various interpretive papers were written on the subject of application of the theory to rotor systems with 3 or more blades. A FORTRAN 4 program was developed based on those papers. That program was upgraded and corrected in a FORTRAN 77 version. This C++ program was based on the FORTRAN 77 version.

# Introduction

CLMNANAL is a C++ program for applying the analysis presented in reference 1 to rotorcraft configurations with 3 or more articulated blades. CLMNANAL will allow for analysis of multiple excitation modes, but has only been tested for the two mode case of pitch and roll. One possible use for this multiple-mode capability would be for the analysis of one or two different modes of motion for multiple weight configurations.

The C++ source code, including custom header files, is listed in the appendices.

# Compilation

CLMNANAL is fully ANSI compatible, and will compile and run on either a Unix or DOS based platform. The program was compiled on a HP-UX system (Unix) with the GNU C++ compiler. The C++ libraries were in a directory, /usr/gnu/lib/g++_include, which was linked to the standard include directory, /usr/include, by the link “gnu”. As a result, this link was part of the include statements for C++ specific header files. For compilation on such a system, the compiler switch -D_GNU was used to instruct the compiler to use the linked headers instead of the standard headers. Compilation on DOS based systems was performed using the Microsoft Visual C++ 1.52 compiler set for a DOS target application.

For limited debug output, the compiler switch -D_DBG can be used. This debug output will be printed to a file named “DebugOut”.

# Command Line Syntax

The command line syntax for CLMNANAL is:

CLMNANAL [*option*]

where available options are:

-h or -H - Displays a help file.

-o or -O - Displays an overview of CLMNANAL and how to run it.
-a or -A - Runs an analysis session.
-t or -T - Runs an analysis session and displays time elapsed for data entry, time elapsed for computations and total time elapsed.

# Input

CLMNANAL will run with two kinds of input: Dimensional (engineering units) and non-dimensional. The dimensional inputs, in the order they would appear in a text file, are:

Integer number of modes to analyze
Integer number of blades.
Rotor radius (in)
Hinge radial station (in)
Blade radius of gyration (in)
Blade weight (lb)
Pylon weight (lb)
Lag hinge radial spring constant (ft-lb/rad)
Lag hinge radial damping constant (ft-lb-sec/rad)
For each mode:
  Descriptive mode label string
  Pylon spring constant (lb/ft)
  Pylon damping constant (lb-sec/ft)
  Name of file in which to store results for this mode

The non-dimensional inputs, in the order they would appear in a text file, are:

Integer number of modes to analyze
Blade lead/lag natural frequency (Hz)
Blade lead/lag critical damping ratio
Ratio of lead/lag hinge radial location to the blade radius of gyration.
One-half the ratio of the sum of the rotor blade masses to the sum of blade masses and pylon mass
For each mode:
  Descriptive mode label string
  Pylon natural frequency (Hz)
  Pylon critical damping ratio
  Name of file in which to store results for this mode

# Output

The results of the analysis will be stored in the file specified by the user in the input. The output consists of:

A record of the input data.

A listing of complex roots, ordered by rotor RPM. The resolution of this listing will be according to user input unless a resonance was encountered. In the event of a resonance, the roots will be output at a resolution of $1/20^{th}$ the user specified resolution.

A damping analysis specifying the system damping product available, and the damping product required.

Resonance frequency information - range, predicted range and mid-point of range.

A comparison of the maximum real root and the predicted maximum real root.

# References

1. Coleman, Robert P.; and Feingold, Arnold M.: *Theory of Self-Excited Mechanical Oscillations of Helicopter Rotors With Hinged Blades.* NACA TR 1351, NACA TN 3844, 1958.

# Appendix I

# String Class

## Specification

```
// mystring.h
//
// Specification for custom String class.
//
// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#ifndef _MYSTRING_H
#define _MYSTRING_H

#ifdef _GNU
#include <gnu/iostream.h>
#else
#include <iostream.h>
#endif

class String
{
  public:

    String();  // The constructor.  Since the text data
               // member of a String instance is dynamically
               // allocated, the actual String will be built
               // with an “Invocation” function.

    String(const String&);  // This is the copy constructor.
                            // Note that since it is called by
                            // reference, the parameter does
                            // not call the constructor.

    ~String(); // The destructor.

    int IString(char*);  // Invocation function.  Handles the
                         // dynamic allocation tasks for all
                         // String instances.

    int IString(const char*);  // Overloaded IString function.
```

```
    int IString(unsigned char*);  // Overloaded IString.

    int IString(const String&);  // Overloaded IString.

    int GetLength() const;  // Function to return the String
                            // length.

// Function to return the location in the String of the
// first occurrence of the FindSet string:
    int FindFirst(const char* pFindSet) const;

// Function to return the last location in the String of the
// occurrence of the FindSet string:
    int FindLast(const char* pFindSet) const;

    char* GetTextPtr() const;  // Function to return a
                               // pointer to the String text.

    String LowerCase();  // Function to return the lower-case
                         // representation of a String.

    String UpperCase();  // Function to return the upper-case
                         // representation of a String.

// Function to return a String of NumChar characters
// starting at Position in the String:
    String GetChars(int Position, int NumChar) const;

    String GetChars(const char* pChars) const;  // Overloaded
                                                // GetChars.

// Overloaded = to handle assignments like s1=s2. It
// must also allow s1=s2=s3:
    String& operator= (const String&);

    String& operator= (const char*); // Overloaded = to handle
                                     // s1=" ".

// Overloaded + to handle s1+s2.  Can add a constant String
// to s1, the result is not a constant.  Note this function
// cannot return a referenced value because a return value
// must be created in the function to leave both the implicit
// and explicit values unchanged. The return value does not
// have any persistence.  Also handles s1+" ".
    String operator+ (const String&) const;
```

```
    String operator+ (const char &c) const;  // Overloaded +
                                             // to handle
                                             // s1 + ' '.

    int operator== (const String&) const;  // Overloaded == to
                                           // handle s1==s2.

    int operator!= (const String&) const;  // Overloaded !=
                                           // to handle
                                           // s1!=s2.

    int operator== (const char*) const;  // Overloaded == to
                                         // handle s1==" ".

    int operator!= (const char*) const;  // Overloaded != to
                                         // handle s1!=" ".

    int operator> (const String&) const; // Overloaded > to
                                         // handle s1>s2.

    int operator< (const String&) const; // Overloaded < to
                                         // handle s1<s2.

// Overloaded [] to return the i'th member of the
// text part of a String:
    char& operator[] (const int&) const;

// Overloaded cast operator.  Allows casting of Strings
// to a constant character pointer:
    operator const char* () const;

// Overloaded cast operator.  Allows casting of Strings
// to a character pointer:
    operator char* () const;

// Function to compare a String to two input strings:
    int CompOr(const char*, const char*) const;

// All the following functions must be friends because the
// first function argument is not a member of the String
// class. All true class member functions are called with the
// first argument data supplied implicitly by the class object
// through use of the Object.Function(Argument) notation.
// Friend functions are called like non-class-member functions
// in order to explicitly provide the data through the
```

```
// arguments.

// This function allows an instance of s1+s2, " "+s1 or
// " "+" ".  It uses the cast operator to properly convert the
// Strings:
    friend String operator+ (const char*, const char*);

// Function to check for " " == s1:
    friend int operator== (char*, const String&);

// Function to check for " " != s1:
    friend int operator!= (char*, const String&);

// Function to check for s1>s2, " ">s1, or " ">" ":
    friend int operator> (const char*, const char*);

// Function to check for s1<s2, " "<s1, or " "<" ":
    friend int operator< (const char*, const char*);

// Overloaded cin so String type variables can be read from
// stdin. It must return an istream type, and must be a friend
// because istream data are not accessible by the String
// class.  istreams cannot be converted to Strings, nor can
// Strings be converted to istreams.
    friend istream& operator>> (istream&, String&);

// Overloaded cout so String type variables can be written to
// stdout. It must return an ostream type, and must be a
// friend because ostream data are not accessible by the
// String class.  ostreams cannot be converted to Strings,
// nor can Strings be converted to ostreams.
    friend ostream& operator<< (ostream&, const String&);

// Data Members
  private:

    int lgth;
    char *txt;
};

#endif
```

Implementation

```
// mystring.cpp

// Implementation of custom String class.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#include <stdlib.h>
#include <string.h>
#include "mystring.h"

String::String() // Constructor with void argument.
{
  lgth=1;  // Initialize the length to 1.
  txt=NULL;  // Initialize the text pointer to NULL.
}

String::String(const String &s)  // Copy constructor.
{
  lgth=s.lgth;
  if((txt= new char[lgth+1]) == NULL)  // Catch memory
  {                                    // problem.
    cout << "\nString copy constructor could not allocate";
    cout << " memory for String \"" << s.txt << "\".\n";
  }
  strcpy(txt,s.txt);
}

String::~String()  // Destructor.
{
  delete [] txt;
}

int String::IString(char *ds)  // Invocation function.
                               // Can return an error flag
{                              // if needed.
  lgth=strlen(ds);  // Set length.
  if((txt= new char[lgth+1]) == NULL)  // Dynamic allocation
                                       // of txt.
    return(0);  // Return 0 if there is not enough memory.
  else
    strcpy(txt,ds);  // Else copy the string to txt.
```

```
  return(1);  // Return 1 on success.
}

int String::IString(const char *ds)  // Overloaded Invocation
{                                    // function.
  lgth=strlen(ds);  // Set length.
  if((txt= new char[lgth+1]) == NULL)  // Dynamic allocation
                                       // of txt.
    return(0);  // Return 0 if there is not enough memory.
  else
    strcpy(txt,ds);  // Else copy the string to txt.

  return(1);  // Return 1 on success.
}

int String::IString(unsigned char *ds)  // Overloaded
I                                        // Invocation
{                                        // function.
  lgth=strlen((const char*)ds);  // Set length.
  if((txt= new char[lgth+1]) == NULL)  // Dynamic allocation
                                       // of txt.
    return(0);  // Return 0 if there is not enough memory.
  else
    strcpy(txt,(char*)ds);  // Else copy the string to txt.

  return(1);  // Return 1 on success.
}

int String::IString(const String &InString)  // Overloaded
                                              // Invocation
{                                             // function.
  int rtInt;  // Declare a return value holder.
  lgth=InString.lgth;
  if((txt=new char[lgth+1]) == NULL)
    rtInt=1;  // Return 1 if there is not enough memory.
  else
  {
    strcpy(txt,InString.txt);  // Copy InString txt member to
                               // (*this).txt.
    rtInt=1;  // Return 1 if all is well.
  }
  return(rtInt);
}

int String::FindFirst(const char* pFindSet) const
{
```

```
  int Location= -1;  // Declare and assign Location.
  int SetLgth= (int)strlen(pFindSet);  // Declare and assign
                                       // SetLgth.
  if(SetLgth > lgth)
    return(Location);  // Return an error value.
  int i;  // Declare a counter.
  int j;  // Declare a counter.
  i -= i;  // Initialize i to zero.
  j -= j;  // Initialize j to zero.
  int LgthLeft= lgth;  // Declare and assign LgthLeft.
  int SetLeft= SetLgth;  // Declare and assign SetLeft.
  while((SetLeft) && (SetLeft <= LgthLeft))
  {
    if(txt[i] == pFindSet[j])
    {
      ++j;  // Increment j.
      --SetLeft;  // Decrement SetLeft.
    }
    else
    {
      if(j > 0)  // If there was a previous match, and now a
      {          // mismatch:
        --i;  // Decrement i to recompare this char with
              // FindSet[0].
        ++LgthLeft;  // Increment LgthLeft to reflect the i
                     // adjustment.
      }
      j -= j;  // Reset j to start comparing FindSet[0].
      SetLeft= SetLgth;  // Reset SetLeft to reflect the j
                         // adjustment.
    }
    --LgthLeft;  // Decrement LgthLeft.
    ++i;  // Increment i.
  }  // End WHILE((j<SetLgth)&&(SetLeft<=LgthLeft)).
  if(!SetLeft)  // If all pFindSet was found:
    Location= i-j;  // Compute Location.
  return(Location);
}

int String::FindLast(const char* pFindSet) const
{
  int Location= -1;  // Declare and assign Location.
  int SetLgth= (int)strlen(pFindSet);  // Declare and assign
                                       // SetLgth.
  if(SetLgth > lgth)
    return(Location);  // Return an error value.
```

```
  int i;  // Declare a counter.
  int j;  // Declare a counter.
  i= lgth-1;  // Initialize i.
  j= SetLgth-1;  // Initialize j.
// Note:  j+1 is number of members left in FindSet to match.
//        i+1 is number of members left in txt to compare to.
  while((j+1) && ((j+1) <= (i+1)))
  {
    if(txt[i] == pFindSet[j])  // When FindSet[0] is found,
                               // done.
      --j;  // Decrement j.
    else
    {
      if(j < (SetLgth-1))  // If there was a previous match,
                           // and now a mismatch:
        ++i;  // Increment i to recompare this char with
              // FindSet[SetLength-1].
      j= SetLgth-1;  // Reset j to start comparing
                     // FindSet[SetLength-1].
    }
    --i;  // Decrement i.
  }  // End WHILE((SetLeft)&&(SetLeft<=LgthLeft)).
// if(j == -1), j was decremented below 0, all were
// found:
  if(!++j)  // If all pFindSet was found:
    Location= ++i;  // Compute Location.
  return(Location);
}

int String::GetLength() const
{
  return(lgth);  // Return the lgth member instance.
}

char* String::GetTextPtr() const
{
  return (txt);  // Return the txt member.
}

String String::LowerCase()
{
  int itxt;  // int for integer representation of txt members.
  char* pLowTxt;  // Pointer to char to hold converted txt.
  pLowTxt= new char [lgth+1];  // Allocate heap for LowTxt.
  for(int i=0; i<lgth; ++i)
  {
```

```
    itxt= (int)*(txt+i);  // Convert txt[i] to int.
    if((itxt > 64) && (itxt < 91))  // If txt[i] is an upper-
                                    // case letter,
      *(pLowTxt+i)= (char)(itxt+32);  // load the converted
                                      // letter to LowTxt.
    else  // Not an upper-case letter.
      *(pLowTxt+i)= (char)itxt;  // Load the char to LowTxt.
  }
  *(pLowTxt+i)='\0';  // Explicitly append NULL.
  String RetString;  // Declare a String for return.
  RetString.IString(pLowTxt);  // Invoke RetString.
  delete [] pLowTxt;  // Free heap.
  return (RetString);  // Return the converted string.
}

String String::UpperCase()
{
  int itxt;  // int for integer representation of txt members.
  char* pUpTxt;  // Pointer to char to hold the converted txt.
  pUpTxt= new char [lgth+1];  // Allocate heap for UpTxt.
  for(int i=0; i<lgth; ++i)
  {
    itxt= (int)*(txt+i);  // Convert txt[i] to int.
    if((itxt > 96) && (itxt < 123))  // If txt[i] is an lower-
                                     // case letter,
      *(pUpTxt+i)= (char)(itxt-32);  // load the converted
                                     // letter to UpTxt.
    else  // Not an lower-case letter.
      *(pUpTxt+i)= (char)itxt;  // Load the char to UpTxt.
  }
  *(pUpTxt+i)='\0';  // Explicitly append NULL.
  String RetString;  // Declare a String for return.
  RetString.IString(pUpTxt);  // Invoke RetString.
  delete [] pUpTxt;  // Free heap.
  return (RetString);  // Return the converted string.
}

String String::GetChars(int Position, int NumChar) const
{
  String RetString;  // Declare a String for the return value.
  int LastIndex= lgth-1;  // Determine the last index of txt.
  if((Position > LastIndex) || (Position < 0) ||
     (NumChar <= 0))
    return (RetString);  // Return the empty String.
  if((Position+NumChar-1) > LastIndex)
    NumChar= lgth-Position;  // Adjust NumChar.
```

```
  char* pTxt= new char[NumChar + 1];
  if(pTxt == NULL)
    return(RetString);  // Return the empty String.
  for(int i= 0; i< NumChar; ++i)
    *(pTxt+i)= txt[i+Position];  // Load RetString.txt[i].
  *(pTxt+i)= '\0';  // Add terminator.
  RetString.IString(pTxt);  // Invoke RetString.
  delete [] pTxt;  // Free heap.
  return (RetString);
}

String String::GetChars(const char* pChars) const
{
  String RetString;  // Declare a String for return.
  int Position= FindFirst(pChars);  // Find the position.
  int NumChar= strlen(pChars);  // Determine NumChars.
  RetString= GetChars(Position, NumChar);  // Get the chars.
  return(RetString);
}

String& String::operator= (const String &s) // Handles s1=s2.
{
  if(this == &s)  // Takes care of s1=s1.
    return *this;
  delete [] txt; // Get rid of space allocated to s1.txt.
  lgth= strlen(s.txt);
  txt= new char[lgth+1]; // Dynamically allocate new implicit
                         // txt storage space.
  if(txt == NULL)
  {
    cout << "\nMemory exhaustion:  String::operator=,”;
    cout << “ new char.\n";
  }
  strcpy(txt,s.txt);
  return *this; // Return the implicit String, s1.
}

String& String::operator= (const char *ds)
{
  lgth= strlen(ds);
  txt= new char[lgth+1];
  if(txt == NULL)
  {
    cout << "\nMemory exhaustion:  String::operator=,”;
    cout << “ new char.\n";
  }
```

```
  strcpy(txt,ds);
  return *this;
}

String String::operator+ (const String &s) const
{
  char *bf= new char[lgth+s.lgth+1]; // Buffer variable to
                                     // hold text of both
                                     // implicit and s.
  if(bf == NULL)
  {
    cout << "\nMemory exhaustion:  String::operator+,";
    cout << " new char.\n";
  }
  strcpy(bf,txt);
  strcat(bf,s.txt);
  bf[lgth+s.lgth]= '\0';  // Terminate bf.
  String rets;  // Uses the default constructor.
  int retval;
  if(!(retval=rets.IString(bf)))
  {
    cout << "\nMemory exhausted at overloaded + for";
    cout << " Strings.\n";
  }
  delete [] bf;
  return rets;
}

String String::operator+ (const char &c) const
{
  char *bf= new char[lgth +2];  // Buffer to hold both txt
                                // and c.
  if(bf == NULL)
  {
    cout << "\nMemory exhaustion:  String::operator+,";
    cout << " new char.\n";
  }
  for(int i=0; i< lgth; ++i)
    *(bf+i)= *(txt+i);  // Assign bf[i].
  *(bf+i)= c;  // Assign bf[lgth];
  *(bf+i+1)= '\0';  // Terminate bf.
  String retString;
  int retval;
  if(!(retval= retString.IString(bf)))
  {
    cout << "\nMemory exhaustion in String::operator+";
```

```
    cout << “ (const char& c).\n";
  }
  delete [] bf;  // Free heap.
  return(retString);
}

int String::CompOr(const char* InChar1, const char* InChar2)
const
{
  return((int)((*this == InChar1) || (*this == InChar2)));
}

String operator+ (const char *pc1, const char *pc2)
{
// Determine the string lengths:
  int len1= strlen(pc1);
  int len2= strlen(pc2);
  char *bf= new char[len1+len2+1];
  if(bf == NULL)
  {
    cout << "\nMemory exhaustion:  String::operator+,”;
    cout << “ new char.\n";
  }
  strcpy(bf,pc1);
  strcat(bf,pc2);
  String rets;
  int retval;
  if(!(retval=rets.IString(bf)))
  {
    cout << "\nMemory exhausted at overloaded + for”;
    cout << “ Strings.\n";
  }
  delete [] bf;
  return rets;
}

int String::operator== (const String &s) const
{
  int retval=strcmp(txt,s.txt);  // Returns 0 if equal, 1 if
                                 // txt greater, -1 if less.
  if(retval == 0)  // If retval is 0, the strings are equal,
    retval=1;  // so return (==) = TRUE.
  else  // Else, rtval was 1 or -1,
    retval=0;  // so return (==) = FALSE.
  return(retval);
}
```

```
int String::operator!= (const String &s) const
{
  int retval=strcmp(txt,s.txt);
  if(retval != 0)  // If retval 1 or -1, not equal is true...
    retval=1;  // return (!=) = TRUE.
  else  // Else, retval was 0, so they are equal...
    retval=0;  // return (!=) = FALSE.
  return(retval);
}

int String::operator== (const char *ds) const
{
  int retval=strcmp(txt,ds);
  //cout << "\nThe comparison value was " << retval << "\n";
  if(retval == 0)
    retval=1;
  else
    retval=0;
  return(retval);
}

int String::operator!= (const char *ds) const
{
  int retval=strcmp(txt,ds);
  if(retval != 0)
    retval=1;
  else
    retval=0;
  return(retval);
}

int operator== (char *s1, const String &s2)
{
  int retval=strcmp(s1,s2.txt);
  if(retval == 0)
    retval=1;
  else
    retval=0;
  return(retval);
}

int operator!= (char *s1, const String &s2)
{
  int retval=strcmp(s1,s2.txt);
  if(retval != 0)
```

```
    retval=1;
  else
    retval=0;
  return(retval);
}

int String::operator> (const String &s) const
{
  int cmpv=strcmp(txt,s.txt);
  if(cmpv == 1)
    return (1);
  else
    return (0);
}

int String::operator< (const String &s) const
{
  int cmpv=strcmp(txt,s.txt);
  if(cmpv == -1)
    return (1);
  else
    return (0);
}

int operator> (const char *pc1, const char *pc2)
{
  int cmpv=strcmp(pc1,pc2);
  if(cmpv == 1)
    return (1);
  else
    return (0);
}

int operator< (const char *pc1, const char *pc2)
{
  int cmpv=strcmp(pc1,pc2);
  if(cmpv == -1)
    return (1);
  else
    return (0);
}

String::operator const char* () const
{
  return (txt);
}
```

```
String::operator char* () const  // Overloaded cast operator.
{
  return(txt);
}

char& String::operator[] (const int &i) const
{
  return(txt[i]);
}

istream& operator>> (istream &inob, String &s)
{
  delete [] s.txt; // Clear out the memory allocated to s.
  char buf[265]; // Create some space with a fixed length.
  inob.getline(buf,132); // Bring whatever is out at the
                         // input area (stored in cin,
                         // here, inob) into the buffer.
  s.lgth=strlen(buf); // Set lgth from the buffer length.
  s.txt=new char[s.lgth+1]; // Since txt was deleted, it must
                            // be reallocated.
  if(s.txt == NULL)
  {
    cout << "\nMemory exhaustion:  String operator>>,";
    cout << " new char.\n";
  }
  strcpy(s.txt,buf); // Load buf into the txt.
  *(s.txt+s.lgth)= '\0';  // Explicitly append NULL.
  return inob; // The return type is istream.
}

ostream& operator<< (ostream& otob, const String& s)
{
  otob<<s.txt;  // Output txt.
  return otob;
}
```

# Appendix II

# Complex Class

## Specification

```
// complex.h

// Specification file for class Complex, a complex number
// structure.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#ifndef _COMPLEX_H
#define _COMPLEX_H

#include <math.h>
#include "mystring.h"

class Complex
{
  public:
    Complex();  // Constructor;
    Complex(double&);  // Conversion constructor.
    Complex(const Complex&);  // Copy constructor.
    ~Complex();  // Destructor.
    int IComplex(double, double, const char*);  // Invocator.
    double GetReal();  // Function to return the real part.
    double GetImag();  // Function to return the imaginary
                       // part.
    double GetMag();  // Function to return the magnitude.
    double GetPhs();  // Function to return the phase angle.
    void SetReal(double InReal);  // Function to set the real
                                  // part.
    void SetImag(double InImag);  // Function to set the
                                  // imaginary part.
    void SetMag(double InMag);  // Function to set the
                                // magnitude.
    void SetPhs(double InPhs);  // Function to set the phase
                                // angle.
    Complex power(double Power);  // Function to raise to a
                                  // power.
```

```
    Complex* root(int Root);  // Function to return roots.
    void root(Complex *pCRoots, int Root);  // Overloaded
                                            // root function.
    Complex operator+ (const Complex&) const;  // Overloaded
                                               // operator +.
    Complex operator- (const Complex&) const;  // Overloaded
                                               // operator -.
    Complex operator* (const Complex&) const;  // Overloaded
                                               // *.
    Complex operator* (const double&) const;  // Overloaded
                                              // operator *.
    Complex operator/ (const Complex&) const;  // Overloaded
                                               // operator /.
    Complex operator/ (const double&) const;  // Overloaded
                                              // operator /.
    Complex& operator= (const Complex&);  // Overloaded
                                          // operator =.
    int operator== (const Complex&) const;  // Overloaded
                                            // operator ==.
    int operator!= (const Complex&) const;  // Overloaded
                                            // operator !=.
    void PrintComplex() const;  // Print function.

// Function to return the complex conjugate:
    void CompConjugate (const Complex&);

// Overloaded * for "double * Complex" case:
    friend Complex operator* (double&, Complex&);

// Overloaded operator<< for output of real and imaginary
// parts:
    friend ostream& operator<< (ostream& otop,
                                Complex& OtComp);

// Data Members
  public:
    double Real;  // Data member Real.
    double Imag;  // Data member Imag.
    double Mag;  // Data member Mag.
    double Phs;  // Data member Phs, radians.
    double PI;  // Public data member PI.

  private:
    void RItoMP();  // Function to compute the Mag and Phs
                    // from Real and Imag.
    void MPtoRI();  // Function to compute the Real and Imag
```

```
                    // from Mag and Phs.
    void AdjustPhase();  // Function to remove multiples of 2
                         // Pi from phase angles.
};

#endif
```

Implementation

```
// complex.C

// Specification of class Complex.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#include "complex.h"

Complex::Complex()
{
  Real -= Real;
  Imag -= Imag;
  Mag -= Mag;
  Phs -= Phs;
  PI= 4.0*atan(1.0);
}

Complex::Complex(double& InDouble)
{
  Real=InDouble;  // Assign Real.
  Imag=0.0;  // Assign Imag.
  RItoMP();  // Compute Mag and Phs.
}

Complex::Complex(const Complex &InComplex)
{
  Real= InComplex.Real;
  Imag= InComplex.Imag;
  Mag= InComplex.Mag;
  Phs= InComplex.Phs;
  PI= InComplex.PI;
}

Complex::~Complex()
{
}

void Complex::AdjustPhase()
{
  Phs -= (int)(Phs/(2.0*PI))*2.0*PI;
}
```

```
void Complex::MPtoRI()
{
  Real=Mag*cos(Phs);  // Compute Real.
  Imag=Mag*sin(Phs);  // Compute Imag.
}

void Complex::RItoMP()
{
  Mag=sqrt(Real*Real + Imag*Imag);  // Compute Mag.
  if(Real == (double)0.0)
    Phs= PI/2.0;
  else
    Phs=atan(Imag/Real);  // Compute Phs.
  if(Real < (double)0)
    Phs += PI;
  else if(Imag < (double)0)
    Phs += 2.0 * PI;
  AdjustPhase();  // Remove multiples of 2 Pi from Phs.
}

int Complex::IComplex(double A, double B, const char *RIMP)
{
  String *RIorMP=NULL;  // Declare and initialize a String
                        // pointer.
  RIorMP=new String;
  int rtval= RIorMP->IString(RIMP);  // Invoke RIorMP String.
  if(!rtval)  // If the String allocation failed,
    return (-1);  // return an error.
  if(RIorMP->UpperCase() == (const char*)"RI")
  {
    Real=A;  // Assign Real.
    Imag=B;  // Assign Imag.
    RItoMP();  // Call function to compute Mag and Phs.
    rtval=1;  // Assign rtint 1.
  }
  else if(RIorMP->UpperCase() == (const char*)"MP")
  {
    Mag=A;  // Assign Mag.
    Phs=B;  // Assign Phs.
    AdjustPhase();  // Remove multiples of 2 Pi from Phs.
    MPtoRI();  // Call function to compute Real and Imag.
    rtval=1;  // Assign rtint 1.
  }
  else
    cout << "\nImproper option String in IComplex call.\n";
  delete RIorMP;  // Free heap.
```

```
  return(rtval);
}

double Complex::GetReal()
{
  return(Real);
}

double Complex::GetImag()
{
  return(Imag);
}

double Complex::GetMag()
{
  return(Mag);
}

double Complex::GetPhs()
{
  return(Phs);
}

void Complex::SetReal(double InReal)
{
  Real= InReal;  // Assign Real.
  RItoMP();  // Compute new Mag and Phs.
}

void Complex::SetImag(double InImag)
{
  Imag= InImag;  // Assign Imag.
  RItoMP();  // Compute new Mag and Phs.
}

void Complex::SetMag(double InMag)
{
  Mag= InMag;  // Assign Mag.
  MPtoRI();  // Compute new Real and Imag.
}

void Complex::SetPhs(double InPhs)
{
  Phs= InPhs;  // Assign Phs.
  AdjustPhase();  // Remove multiples of 2 PI from Phs.
  MPtoRI();  // Compute new Real and Imag.
```

```
}

Complex Complex::power(double Power)
{
  Complex ReturnComplex;  // Declare a return Complex.
  double PowerMag= pow(Mag, Power);  // Declare and assign
                                     // PowerMag.
  double PowerPhs= Power*Phs;  // Declare and assign PowerPhs.
  ReturnComplex.IComplex(PowerMag, PowerPhs,
                         (const char*)"MP");
  return(ReturnComplex);
}

void Complex::root(Complex* pCRoots, int Root)
{
// Declare and assign RootMag:
  double RootMag= pow(Mag, (1/(double)Root));
  double RootPhs;  // Declare RootPhs.
  const char* MP= "MP";  // Declare and assign a descriptor
                         // for the invocator.
  int i;  // Declare a counter.
  for(i=0; i<Root; ++i)
  {
    RootPhs= (Phs + 2.0*PI*i)/Root;  // Compute RootPhs.
// Invoke the i'th array member:
    (pCRoots+i)->IComplex(RootMag, RootPhs, MP);
  }
}

Complex Complex::operator+ (const Complex &InComplex) const
{
  Complex RetComplex;  // Declare a Complex.
  RetComplex.Real= Real + InComplex.Real;  // Compute new
                                           // Real.
  RetComplex.Imag= Imag + InComplex.Imag;  // Compute new
                                           // Imag.
  RetComplex.RItoMP();  // Compute new Mag and Phs.
  return (RetComplex);
}

Complex Complex::operator- (const Complex &InComplex) const
{
  Complex RetComplex;  // Declare a Complex.
  RetComplex.Real= Real - InComplex.Real;  // Compute new
                                           // Real.
  RetComplex.Imag= Imag - InComplex.Imag;  // Compute new
```

```
                                                // Imag.
  RetComplex.RItoMP();  // Compute new Mag and Phs.
  return (RetComplex);
}

Complex Complex::operator* (const Complex &InComplex) const
{
  Complex RetComplex;  // Declare a Complex.
  RetComplex.Mag= Mag * InComplex.Mag;  // Compute new Mag.
  RetComplex.Phs= Phs + InComplex.Phs;  // Compute new Phs.
  RetComplex.AdjustPhase();
  RetComplex.MPtoRI();  // Compute new Real and Imag.
  return (RetComplex);
}

Complex Complex::operator* (const double &InDouble) const
{
  Complex RetComplex;  // Declare a Complex.
  RetComplex.Mag= Mag * InDouble;  // Compute new Mag.
  RetComplex.Phs= Phs;  // Assign Phs.
  RetComplex.MPtoRI();  // Compute new Real and Imag.
  return (RetComplex);
}

Complex Complex::operator/ (const Complex &InComplex) const
{
  Complex RetComplex;  // Declare a Complex.
  RetComplex.Mag= Mag / InComplex.Mag;  // Compute new Mag.
  RetComplex.Phs= Phs - InComplex.Phs;  // Compute new Phs.
  RetComplex.MPtoRI();  // Compute new Real and Imag.
  return (RetComplex);
}

Complex Complex::operator/ (const double &InDouble) const
{
  Complex RetComplex;  // Declare a Complex.
  RetComplex.Mag= Mag / InDouble;  // Compute new Mag.
  RetComplex.Phs= Phs;  // Assign Phs.
  RetComplex.MPtoRI();  // Compute new Real and Imag.
  return (RetComplex);
}

Complex& Complex::operator= (const Complex &InComplex)
{
  if(this == &InComplex)  // If a case of 'C = C',
    return(*this);  // return 'C'.
```

```
  Real=InComplex.Real;  // Assign Real.
  Imag=InComplex.Imag;  // Assign Imag.
  Mag=InComplex.Mag;  // Assign Mag.
  Phs=InComplex.Phs;  // Assign Phs.
  return (*this);
}

int Complex::operator== (const Complex &InComplex) const
{
  int rtint=0;  // Declare and initialize a return int.
  if(Real == InComplex.Real)  // If Real == Real,
    if(Imag == InComplex.Imag)  // check Imag.  If 1,
      if(Mag == InComplex.Mag)  // check Mag.  If 1,
        if(Phs == InComplex.Phs)  // check Phs.  If 1,
          rtint=1;  // numbers are equal.
  return (rtint);  // Return int.
}

int Complex::operator!= (const Complex &InComplex) const
{
  int rtint=0;  // Declare and initialize a return int.
  if(Real != InComplex.Real)  // If Real != Real,
    if(Imag != InComplex.Imag)  // check Imag.  If 1,
      if(Mag != InComplex.Mag)  // check Mag.  If 1,
        if(Phs != InComplex.Phs)  // check Phs.  If 1,
          rtint=1;  // numbers are equal.
  return (rtint);  // Return int.
}

void Complex::CompConjugate(const Complex &InComplex)
{
  Real= InComplex.Real;  // Assign Real.
  Imag= -InComplex.Imag;  // Assign Imag.
  RItoMP();  // Compute new Mag and Phs.
}

void Complex::PrintComplex() const
{
  cout << "\nAddress " << this << " data:";
  cout << "\nIn RI format:\n";
  cout << Real << " + " << Imag << " i\n";
  cout << "\nIn MP format:";
  cout << "\n" << Mag << " cis " << Phs << " radians";
  cout << "\n" << Mag << " cis " << Phs*180.0/PI;
  cout << " degrees\n";
}
```

```
Complex operator* (double &InDouble, Complex &InComplex)
{
  Complex RetComplex;  // Declare a Complex.
  RetComplex.Mag= InComplex.Mag * fabs(InDouble);  // Compute
                                                   // new Mag.
  RetComplex.Phs= InComplex.Phs;  // Assign Phs.
// Check for a negative multiplier:
  if((int)(InDouble/fabs(InDouble)) < 0)  // If -,
    RetComplex.Phs= InComplex.Phs + InComplex.PI;  // adjust
                                                   // Phs.
  RetComplex.AdjustPhase();  // Remove increments of 2 Pi
                             // from Phs.
  RetComplex.MPtoRI();  // Compute new Real and Imag.
  return (RetComplex);
}

ostream& operator<< (ostream& otop, Complex& OtComp)
{
  otop.setf(ios::showpos);  // Make the + print for positive
                            // numbers.
  otop << OtComp.Real << "             " << OtComp.Imag;
  otop << " i";
  return(otop);
}
```

# Appendix III

# CDatDim Class

## Specification

```
// cdatdim.h
//
// Specification file for class CDatDim.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#ifndef _CDATDIM_H
#define _CDATDIM_H

#ifdef _GNU
#include <gnu/fstream.h>
#else
#include <fstream.h>
#endif

#include <math.h>
#include "mystring.h"

#ifndef _PI
#define _PI
#define Pi (double)(4.0*atan((double)1.0))
#endif

class CDatDim
{
  public:
    CDatDim();  // Constructor.
    ~CDatDim();  // Destructor.
    CDatDim(const CDatDim& Orig);  // Copy constructor.
    int ICDatDim(int Interactive= 0);  // Invocator.
// Function to read input from file:
    void ReadDataDimensional(fstream* File);
// Function to print a file top Header:
    void PrintHeader(int Mode, fstream* pFile);
// Function to save input data to file:
    void SaveData(fstream* File);
```

```
// Overloaded assignment operator:
    CDatDim& operator= (const CDatDim& Orig);
// Overloaded output operator:
    friend ostream& operator<< (ostream& otop,
                                CDatDim& OutData);
    friend class CClmnData;  // Declare class CClmnData to be
                             // a friend.

  protected:
    void InPrompt();  // Function to prompt user for input.
// Function to make a non-shallow copy:
    int MakeCopy(const CDatDim& Orig);
    int AllocateMemory();  // Function to allocate memory.

  private:
    int NumBlades;  // Number of rotor blades.
    int NumModes;  // Number of modes of motion to analyze.
    double Radius;  // Blade radius, inches.
    double HingeRadius;  // Lag-hinge radial station, inches.
    double HingeToCG;  // Distance from hinge station to blade
                       // center of gravity.
    double BladeWeight;  // Blade weight, pounds.
    double PylonWeight;  // Support pylon (gimbal) weight,
                         // pounds.
    double *pStructK;  // Array of NumModes structural spring
                       // constants, pound/foot.
    double *pStructB;  // Array of NumModes damping
                       // coefficients, pound/foot-second.
    double BladeK;  // Lag hinge spring constant,
                    // foot-pound/radian.
    double BladeB;  // Lag damping constant,
                    // foot-pound-second/radian.
    String *pOutFileName;  // Array of NumModes file names
                           // for output.
    String *pModeLabel;  // Array of NumModes mode identifier
                         // Strings.
};

#endif
```

Implementation

```
// cdatdim.C

// Implementation of class CDatDim.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#include "cdatdim.h"

CDatDim::CDatDim()
{
  NumBlades -= NumBlades;  // Initialize to zero.
  Radius -= Radius;  // Initialize to zero.
  HingeRadius -= HingeRadius;  // Initialize to zero.
  HingeToCG -= HingeToCG;  // Initialize to zero.
  BladeWeight -= BladeWeight;  // Initialize to zero.
  PylonWeight -= PylonWeight;  // Initialize to zero.
  BladeK -= BladeK;  // Initialize to zero.
  BladeB -= BladeB;  // Initialize to zero.
  pStructB= NULL;  // Point to NULL.
  pStructK= NULL;  // Point to NULL.
  pOutFileName= NULL;  // Point to NULL.
  pModeLabel= NULL;  // Point to NULL.
}

CDatDim::~CDatDim()
{
  if(NumModes > 1)  // If more than 1 Mode, then arrays
  {                 // were allocated.
    delete [] pStructB;  // Free array of heap.
    delete [] pStructK;  // Free array of heap.
    delete [] pOutFileName;  // Free array of heap.
    delete [] pModeLabel;  // Free array of heap.
  }
  else
  {
    delete pStructB;  // Free heap.
    delete pStructK;  // Free heap.
    delete pOutFileName;  // Free heap.
    delete pModeLabel;  // Free heap.
  }
}
```

```
CDatDim::CDatDim(const CDatDim& Orig)
{
  MakeCopy(Orig);  // Call MakeCopy for a non-shallow copy.
}

void CDatDim::InPrompt()
{
  cout << "\nWhat is the number of blades ? ";
  cin >> NumBlades;
  cout << "\nWhat is the blade radius (in) ? ";
  cin >> Radius;
  cout << "\nWhat is the radial hinge location (in) ? ";
  cin >> HingeRadius;
  cout << "\nWhat is the distance from the hinge to the”;
  cout << “ blade CG (in) ? ";
  cin >> HingeToCG;
  cout << "\nWhat is the blade weight in (lb) ? ";
  cin >> BladeWeight;
  cout << "\nWhat is the support pylon weight in (lb) ? ";
  cin >> PylonWeight;
  cout << "\nWhat is the blade spring constant”;
  cout << “ (ft-lb/radian) ? ";
  cin >> BladeK;
  cout << "\nWhat is the blade damping constant”;
  cout << “ (ft-lb-sec/radian) ? ";
  cin >> BladeB;
  cin.get();  // Clear the input buffer.
  for(int i=0; i<NumModes; ++i)
  {
    cout << "\nFor mode " << i+1 << ":\n";
    cout << "\nWhat is a descriptive label for this mode ? ";
    cin >> *(pModeLabel+i);
    cout << "\nWhat is the support spring constant”;
    cout << “ (lb/ft) ? ";
    cin >> *(pStructK+i);
    cout << "\nWhat is the support damping constant”;
    cout << “ (lb-sec/ft) ? ";
    cin >> *(pStructB+i);
    cin.get();  // Clear the input buffer.
    cout << "\nWhat do you wish to name the analysis results”;
    cout << “ output file ? ";
    cin >> *(pOutFileName+i);
  }
}

void CDatDim::SaveData(fstream* File)
```

```
{
  *File << NumModes << "\n";
  *File << NumBlades << "\n";
  *File << Radius << "\n";
  *File << HingeRadius << "\n";
  *File << HingeToCG << "\n";
  *File << BladeWeight << "\n";
  *File << PylonWeight << "\n";
  *File << BladeK << "\n";
  *File << BladeB << "\n";
  for(int i=0; i<NumModes; ++i)
  {
    *File << *(pModeLabel+i) << "\n";
    *File << *(pStructK+i) << "\n";
    *File << *(pStructB+i) << "\n";
    *File << *(pOutFileName+i) << "\n";
  }
}

int CDatDim::MakeCopy(const CDatDim& Orig)
{
  NumBlades= Orig.NumBlades;  // Assign NumBlades.
  NumModes= Orig.NumModes;  // Assign NumModes.
  Radius= Orig.Radius;  // Assign Radius.
  HingeRadius= Orig.HingeRadius;  // Assign HingeRadius.
  HingeToCG= Orig.HingeToCG;  // Assign HingeToCG.
  BladeWeight= Orig.BladeWeight;  // Assign BladeWeight.
  PylonWeight= Orig.PylonWeight;  // Assign PylonWeight.
  BladeK= Orig.BladeK;  // Assign BladeK.
  BladeB= Orig.BladeB;  // Assign BladeB.
  int NoAlloc= AllocateMemory();  // Allocate required heap.
  if(NoAlloc)
    return(NoAlloc);  // Memory allocation error!
  for(int i=0; i<NumModes; ++i)
  {
    *(pStructB+i)= *(Orig.pStructB+i);  // Assign StructC[i].
    *(pStructK+i)= *(Orig.pStructK+i);  // Assign StructK[i].
    *(pOutFileName+i)= *(Orig.pOutFileName+i);  // Assign
                                          // OutFileName[i].
    *(pModeLabel+i)= *(Orig.pModeLabel+i);  // Assign
                                         // ModeLabel[i].
  }
  return(NoAlloc);
}

void CDatDim::ReadDataDimensional(fstream *File)
```

```
{
  *File >> NumBlades;
  *File >> Radius;
  *File >> HingeRadius;
  *File >> HingeToCG;
  *File >> BladeWeight;
  *File >> PylonWeight;
  *File >> BladeK;
  *File >> BladeB;
  File->get();  // Clear the input buffer.
  for(int i=0; i<NumModes; ++i)
  {
    *File >> *(pModeLabel+i);
    *File >> *(pStructK+i);
    *File >> *(pStructB+i);
    File->get();  // Clear the input buffer.
    *File >> *(pOutFileName+i);
  }
}

int CDatDim::AllocateMemory()
{
  int NoAlloc= 0;  // Declare and assign an int to return the
                   // status of the allocation.
  if(NumModes > 1)
  {
    pModeLabel= new String [NumModes];  // Allocate heap.
    pStructK= new double [NumModes];  // Allocate heap.
    pStructB= new double [NumModes];  // Allocate heap.
    pOutFileName= new String [NumModes];  // Allocate heap.
  }
  else if(NumModes == 1)
  {
    pModeLabel= new String;  // Allocate heap.
    pStructK= new double;  // Allocate heap.
    pStructB= new double;  // Allocate heap.
    pOutFileName= new String;  // Allocate heap.
  }
  else
    NoAlloc= 1;
  if((pModeLabel == NULL) || (pStructK == NULL) ||
     (pStructB == NULL) || (pOutFileName == NULL))
    NoAlloc= 1;  // There was insufficient memory!
  return(NoAlloc);
}
```

```
int CDatDim::ICDatDim(int Interactive)
{
  int ChangeInput= 1;  // Declare and initialize an input
                       // status flag.
  String EchoChoice;  // Declare a String to catch user
                      // response.
  String* pDatFileName= new String;  // Declare and allocate
                                     // a String.
  fstream* pDataFile= new fstream;  // Declare and allocate
                                    // an fstream.
  if((pDatFileName == NULL) || (pDataFile == NULL))
  {
    cout << "\nMemory allocation error !\n";
    return(1);  // Return with a memory allocation error.
  }
  int NoAlloc= 0;  // Declare and assign an int to return the
                   // status of the allocation.
  if(Interactive)
  {
    cout << "\nWhat is the number of modes to analyze ? ";
    cin >> NumModes;
    NoAlloc= AllocateMemory();  // Call the memory allocation
                                // function.
    if(NoAlloc)
      return(NoAlloc);  // Return with a memory allocation
                        // error.
    while(ChangeInput)
    {
      InPrompt();  // Call the input prompt routine.
      cout << *this;  // Echo the input.
      cout << "\n\nDo you wish to change any input (\"Y\",”;
      cout << “ ALL inputs must be";
      cout << " reentered, or \"N\") ? ";
      cin >> EchoChoice;
      EchoChoice= EchoChoice.UpperCase();
      if(EchoChoice == (const char*)"N")
        ChangeInput -= ChangeInput;  // Set the flag to
                                     // terminate the while.
      else if(EchoChoice != (const char*)"Y")
      {
        cout << "\n\"" << EchoChoice << "\" improper”;
        cout << “ choice!\n";
      }
    }  // End WHILE(ChangeInput).
    cout << "\nWhat do you wish to name the data storage”;
    cout << “ file ? ";
```

```
    cin >> *pDatFileName;
    pDataFile->open((const char*)*pDatFileName, ios::out);
    if(!pDataFile)
    {
      cout << "\nFile " << *pDatFileName << ":\n";
      return(1);  // Return with a memory allocation error.
    }

  }
  else
  {
    cout << "\nWhat is the name of the input data file ? ";
    cin >> *pDatFileName;
// Try to open the file:
    pDataFile->open((const char*)*pDatFileName, ios::in|
                    ios::nocreate);
    if(!*pDataFile)
    {
      cout << "\nFile " << *pDatFileName << ":\n";
      return(2);  // Return with a memory allocation error.
    }
    *pDataFile >> NumModes;
    NoAlloc= AllocateMemory();  // Call the memory allocation
                                // function.
    if(NoAlloc)
      return(NoAlloc);  // Return with a memory allocation
                        // error.
    ReadDataDimensional(pDataFile);  // Read the data from
                                     // file.
    if(pDataFile->bad())
    {
      cout << "\nData file bad ! \n";
      return (2);
    }
    else
      pDataFile->close();  // Close the file.
    while(ChangeInput)
    {
      cout << *this;  // Echo the input.
      cout << "\n\nDo you wish to change any input (\"Y\",”;
      cout << “ ALL inputs must be";
      cout << " reentered, or \"N\") ? ";
      cin >> EchoChoice;
      EchoChoice= EchoChoice.UpperCase();
      if(EchoChoice == (const char*)"N")
        ChangeInput -= ChangeInput;  // Set the flag to
```

```
                                        // terminate the while.
      else if(EchoChoice == (const char*)"Y")
        InPrompt();  // Prompt for new input.
      else
      {
        cout << "\n\"" << EchoChoice << "\" improper”;
        cout << “ choice!\n";
      }
    }
    pDataFile->close();  // Close the DataFile.
    pDataFile->open((const char*)*pDatFileName, ios::out|
                    ios::nocreate);
    pDataFile->clear();  // Clear the state of DataFile.
  }
  if(pDataFile->bad())
  {
    cout << "\nData file is corrupted or did not open
properly !\n";
    NoAlloc= 2;
  }
  else
  {
    SaveData(pDataFile);
    pDataFile->close();
  }
  delete pDatFileName;  // Free heap.
  delete pDataFile;  // Free heap.
  return(NoAlloc);  // Return the memory allocation results.
}

CDatDim& CDatDim::operator= (const CDatDim& Orig)
{
  if(this == &Orig)
    return(*this);
  else
    MakeCopy(Orig);  // Call MakeCopy for a non-shallow copy.
  return(*this);
}

void CDatDim::PrintHeader(int Mode, fstream *pFile)
{
  *pFile << "\nNumber of blades: " <<  NumBlades;
  *pFile << "\nBlade radius: " <<  Radius << "inches.\n";
  *pFile << "Radial hinge location: " << HingeRadius;
  *pFile << " inches.\nDistance from the hinge to the blade";
  *pFile << " CG: " << HingeToCG << " inches.\n";
```

```
  *pFile << "Blade weight: " << BladeWeight << " lbs.";
  *pFile << "\nSupport pylon weight: " << PylonWeight << ";
  *pFile << " lbs." << "\nBlade spring constant: " << BladeK;
  *pFile << " ft-lb/radian.\nBlade damping constant: ";
  *pFile << BladeB << " ft-lb-sec/radian.\n";
  *pFile << "\nMode: " << *(pModeLabel+Mode);
  *pFile << "\nSupport spring constant: " << *(pStructK+Mode);
  *pFile << " lb/ft.\nSupport damping constant: ";
  *pFile << *(pStructB+Mode) << " lb-sec/ft.\n\n";
  *pFile << "                                Complex Frequency";
  *pFile << " (Hz)\n     RPM                    Real";
  *pFile << "                    Imaginary\n";
}

ostream& operator<< (ostream& otop, CDatDim& OutData)
{
  otop << "\nNumber of blades: " <<  OutData.NumBlades;
  otop << "\nBlade radius: " <<  OutData.Radius << " inches.";
  otop << "\nRadial hinge location: " << OutData.HingeRadius;
  otop << " inches.\nDistance from the hinge to the ";
  otop << "blade CG: " << OutData.HingeToCG << " inches.";
  otop << "\nBlade weight: " << OutData.BladeWeight;
  otop << " lbs.\nSupport pylon weight: ";
  otop << OutData.PylonWeight << " lbs.";
  otop << "\nBlade spring constant: " << OutData.BladeK;
  otop << " ft-lb/radian.\nBlade damping constant: ";
  otop << OutData.BladeB << " ft-lb-sec/radian.";
  for(int i=0; i<OutData.NumModes; ++i)
  {
    otop << "\n\nFor " << *(OutData.pModeLabel+i);
    otop << " mode:\n\nSupport spring constant: ";
    otop << *(OutData.pStructK+i) << " lb/ft.";
    otop << "\nSupport damping constant: ";
    otop << *(OutData.pStructB+i) << " lb-sec/ft.";
    otop << "\nOutput data for this mode will be stored in ";
    otop << "the file \"" << *(OutData.pOutFileName+i);
    otop << "\".";
  }
  otop << "\n";
  return otop;
}
```

# Appendix IV

# CClmnData Class

## Specification

```
// cclmndat.h

// Specification file for class CClmnData.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#ifndef _CCLMNDAT_H
#define _CCLMNDAT_H

#ifdef _GNU
#include <gnu/fstream.h>
#else
#include <fstream.h>
#endif

#include <math.h>
#include "mystring.h"
#include "cdatdim.h"

#ifndef _PI
#define _PI
#define Pi (double)(4.0*atan((double)1.0))
#endif

class CClmnData
{
  public:
    CClmnData();  // Constructor.
    ~CClmnData();  // Destructor.
    CClmnData(const CClmnData& Orig);  // Copy constructor.
    int ICClmnData(int CreateMode= 0);  // Invocator.
    int ICClmnData(CDatDim* pCDatDim);  // Overloaded
                                        // invocator.
// Function to provide access to NumModes:
    void SetNumModes(int InNum);
// Function to provide access to non-array members:
```

```
    int SetNonArray(int NMod, double BDmp, double BFrq,
                    double ARat, double MRat);
// Function to provide access to array members:
    void SetArray(int Ndx, double SFrq, double SDmp, String
                  *pOFN, String *pML,double DC);
// Function to retrieve a pointer to a specific OutFileName:
    String* GetFileName(int Mode);
// Function to read input data from file:
    void ReadData(fstream* File);
// Function to return the number of modes for analysis:
    int GetNumModes();
// Function to compute the coefficients for the
// characteristic equation of motion:
     void ComputeCoeffs(double* pCoefs, double *pRpm,
                        int *pMode);
// Function to compute the damping product available:
    double ComputeDampingData(int *pMode, double *pDampAva,
                              double *pOmega2);
// Function to compute the RPM-related output:
    double ComputeRpmData(int *pMode, double *pMaxRootRpm);
// Function to coupute the predicted maximum real root:
    double ComputeMaxRoot(int *pMode, double *pMaxRootRpm);
// Overloaded assignment operator:
    CClmnData& operator= (const CClmnData& Orig);

// Overloaded output operator:
    friend ostream& operator<< (ostream& otop,
                                CClmnData &OutData);
  protected:
// Function to prompt user for input of nondimensional data:
    void InPrompt();
// Function to make a non-shallow copy:
    int MakeCopy(const CClmnData& Orig);
// Function to print a Header using nondimensional data:
    void PrintHeader(int Mode, fstream* pFile);
// Function to save input nondimensional data to file:
    void SaveData(fstream *File);
// Function to perform memory allocations:
    int AllocateMemory();

  private:
    int NumModes;  // Number of modes of motion to analyze.
    double ArmRatio;  // Ratio of lag-hinge offset to distance
                      // from lag-hinge to blade center of
                      // gravity.
    double BladeFreq;  // Blade natural frequency in lag.
```

```
    double BladeDamp;  // Blade lag critical lag damping
                       // ratio.
    double MassRatio;  // Blade mass ratio.
    double *pStructFreq;  // Array of NumModes structural
                          // natural freqencies.
    double *pStructDampRatio;  // Array of NumModes critical
                               // damping ratios.
    double *pDampConversion;  // Array of double factors for
                              // converting from non-
                              // dimensional to dimensional
                              // representation.
    String *pOutFileName;  // Array of NumModes file names
                           // for output.
    String *pModeLabel;  // Array of NumModes mode identifier
                         // Strings.
};

#endif
```

Implementation

```
// cclmndat.C

// Implementation for class CClmnData.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#include "cclmndat.h"

CClmnData::CClmnData()
{
  NumModes -= NumModes;  // Initialize to zero.
  ArmRatio -= ArmRatio;  // Initialize to zero.
  BladeFreq -= BladeFreq;  // Initialize to zero.
  BladeDamp -= BladeDamp;  // Initialize to zero.
  MassRatio -= MassRatio;  // Initialize to zero.
  pDampConversion= NULL;  // Point to NULL.
  pStructFreq= NULL;  // Point to NULL.
  pStructDampRatio= NULL;  // Point to NULL.
  pOutFileName= NULL;  // Point to NULL;
  pModeLabel= NULL;  // Point to NULL;
}

CClmnData::~CClmnData()
{
  if(NumModes > 1)  // If more than 1 Mode, then arrays
  {                 // were allocated.
    delete [] pStructFreq;  // Free array of heap.
    delete [] pStructDampRatio;  // Free array of heap.
    delete [] pOutFileName;  // Free array of heap.
    delete [] pModeLabel;  // Free array of heap.
    delete [] pDampConversion;  // Free array of heap.
  }
  else
  {
    delete pStructFreq;  // Free heap.
    delete pStructDampRatio;  // Free heap.
    delete pOutFileName;  // Free heap.
    delete pModeLabel;  // Free heap.
    delete pDampConversion;  // Free of heap.
  }
}
```

```
CClmnData::CClmnData(const CClmnData& Orig)
{
  MakeCopy(Orig);  // Call MakeCopy to make a non-shallow
                   // copy.
}

void CClmnData::InPrompt()
{
  cout << "\nWhat is the blade natural frequency in ";
  cout << "lag (Hz) ? ";
  cin >> BladeFreq;
  cout << "\nWhat is the blade lag critical damping ratio ? ";
  cin >> BladeDamp;
  cout << "\nWhat is the ratio of lag-hinge-offset to hinge-";
  cout << "to-blade CG distance ? ";
  cin >> ArmRatio;
  cout << "\nWhat is the blade mass ratio ? ";
  cin >> MassRatio;
  cin.get();  // Clear the input buffer.
  for(int i=0; i<NumModes; ++i)
  {
    cout << "\nFor mode " << i+1 << ":\n";
    cout << "\nWhat is a descriptive label for this mode ? ";
    cin >> *(pModeLabel+i);
    cout << "\nWhat is the support structural natural";
    cout << " frequency for this mode (Hz) ? ";
    cin >> *(pStructFreq+i);
    cout << "\nWhat is the support structural critical";
    cout << " damping ratio ? ";
    cin >> *(pStructDampRatio+i);
    cin.get();  // Clear the input buffer.
    cout << "\nWhat do you wish to name the analysis results";
    cout << " output file ? ";
    cin >> *(pOutFileName+i);
    *(pDampConversion+i)= 1.0;  // Assign DampConversion.
  }
  cout << "\n";
}

void CClmnData::SaveData(fstream *File)
{
  *File << NumModes << "\n";
  *File << BladeFreq << "\n";
  *File << BladeDamp << "\n";
  *File << ArmRatio << "\n";
  *File << MassRatio << "\n";
```

```
  for(int i=0; i<NumModes; ++i)
  {
    *File << *(pModeLabel+i) << "\n";
    *File << *(pStructFreq+i) << "\n";
    *File << *(pStructDampRatio+i) << "\n";
    *File << *(pOutFileName+i) << "\n";
  }
}

int CClmnData::AllocateMemory()
{
  int NoAlloc= 0;  // Declare and assign an int to return the
                   // status of the allocation.
  if(NumModes > 1)
  {
    pModeLabel= new String [NumModes];  // Allocate heap.
    pStructFreq= new double [NumModes];  // Allocate heap.
    pStructDampRatio= new double [NumModes];  // Allocate
                                              // heap.
    pOutFileName= new String [NumModes];  // Allocate heap.
    pDampConversion= new double [NumModes];  // Allocate heap.
  }
  else if(NumModes == 1)
  {
    pModeLabel= new String;  // Allocate heap.
    pStructFreq= new double;  // Allocate heap.
    pStructDampRatio= new double;  // Allocate heap.
    pOutFileName= new String;  // Allocate heap.
    pDampConversion= new double;  // Allocate heap.
  }
  else
    NoAlloc= 1;
  if((pModeLabel == NULL) || (pStructFreq == NULL) ||
     (pStructDampRatio == NULL) || (pOutFileName == NULL) ||
     (pDampConversion == NULL))
    NoAlloc= 1;
  return(NoAlloc);
}

int CClmnData::MakeCopy(const CClmnData& Orig)
{
  NumModes= Orig.NumModes;  // Assign NumModes.
  ArmRatio= Orig.ArmRatio;  // Assign ArmRatio.
  BladeFreq= Orig.BladeFreq;  // Assign BladeFreq.
  BladeDamp= Orig.BladeDamp;  // Assign BladeDamp.
  MassRatio= Orig.MassRatio;  // Assign MassRatio.
```

```
  int NoAlloc= AllocateMemory();  // Allocate required heap.
  if(NoAlloc)
    return(NoAlloc);  // Memory allocation error.
  for(int i=0; i<NumModes; ++i)  // Assign array members:
  {
    *(pStructFreq+i)= *(Orig.pStructFreq+i);
    *(pStructDampRatio+i)= *(Orig.pStructDampRatio+i);
    *(pOutFileName+i)= *(Orig.pOutFileName+i);
    *(pModeLabel+i)= *(Orig.pModeLabel+i);
// Assign DampConversion:
    *(pDampConversion+i)= *(Orig.pDampConversion+i);
  }
  return(NoAlloc);
}

int CClmnData::ICClmnData(int CreateMode)
{
  int ChangeInput= 1;  // Declare and initialize an input
                       // status flag.
  String EchoChoice;  // Declare a String to catch user
                      // response.
  String* pDatFileName= new String;  // Declare and allocate
                                     // a String.
  fstream* pDataFile= new fstream;  // Declare and allocate
                                    // an fstream.
  if((pDatFileName == NULL) || (pDataFile == NULL))
  {
    cout << "\nMemory allocation error !\n";
    return(1);  // Return with a memory allocation error.
  }
  int NoAlloc= 0;  // Declare and initialize an int to track
                   // the tatus of memory allocation.
  if(CreateMode)  // If CreateMode is 1, interactive creation:
  {
    cout << "\nWhat is the number of modes to analyze ? ";
    cin >> NumModes;
    NoAlloc= AllocateMemory();  // Call the memory allocation
                                // function.
    if(NoAlloc)  // Check the allocation status.
      return(NoAlloc);  // Memory allocation error!
    while(ChangeInput)
    {
      InPrompt();  // Call the input prompt routine.
      cout << *this;  // Echo the input.
      cout << "\n\nDo you wish to change any input (\"Y\",";
      cout << " ALL inputs must be";
```

```
      cout << " reentered, or \"N\") ? ";
      cin >> EchoChoice;
      EchoChoice= EchoChoice.UpperCase();
      if(EchoChoice == (const char*)"N")
        ChangeInput -= ChangeInput;  // Set the flag to
                                     // terminate the while.
      else if(EchoChoice != (const char*)"Y")
      {
        cout << "\n\"" << EchoChoice << "\" improper";
        cout << " choice!\n";
      }
    }
    cout << "\nWhat do you wish to name the data storage";
    cout << " file ? ";
    cin >> *pDatFileName;
    pDataFile->open((const char*)*pDatFileName, ios::out);
  }
  else if(!CreateMode)  // If CreateMode is 0, file input:
  {
    cout << "\nWhat is the name of the input data file ? ";
    cin >> *pDatFileName;
    pDataFile->open((const char*)*pDatFileName,
                    ios::in|ios::nocreate);
// Allocate heap for the fstream and open it.
    if(!*pDataFile)
    {
      cout << "\nFile " << *pDatFileName << ":\n";
      return(2);  // Return with a memory allocation error.
    }
    *pDataFile >> NumModes;
    NoAlloc= AllocateMemory();  // Call the memory allocation
                                // function.
    if(NoAlloc)  // Check the allocation status.
      return(NoAlloc);  // Memory allocation error!
    ReadData(pDataFile);  // Read the data from file.
    if(pDataFile->bad())
    {
      cout << "\nData file bad ! \n";
      pDataFile->close();  // Close the data file.
      return (2);
    }
    else
      pDataFile->close();  // Close the data file.
    while(ChangeInput)
    {
      cout << *this;  // Echo the input.
```

```
      cout << "\n\nDo you wish to change any input (\"Y\",";
      cout << " ALL inputs must be";
      cout << " reentered, or \"N\") ? ";
      cin >> EchoChoice;
      EchoChoice= EchoChoice.UpperCase();
      if(EchoChoice == (const char*)"N")
        ChangeInput -= ChangeInput;  // Set the flag to
                                     // terminate the while.
      else if(EchoChoice == (const char*)"Y")
        InPrompt();  // Prompt for new input.
      else
      {
        cout << "\n\"" << EchoChoice << "\" improper";
        cout << " choice!\n";
      }
    }
    pDataFile->open((const char*)*pDatFileName
                    ios::out|ios::nocreate);
    pDataFile->clear();  // Clear the state of DataFile.
  }
  if(pDataFile->bad())
  {
    cout << "\nData file is corrupted or did not open";
    cout << " properly !\n";
    delete pDatFileName;  // Free heap.
    delete pDataFile;  // Free heap.
    return(2);
  }
  else
  {
    SaveData(pDataFile);
    pDataFile->close();
  }
  delete pDatFileName;  // Free heap.
  delete pDataFile;  // Free heap.
  fstream* pFile;  // Declare a pointer to fstream.
  pFile= new fstream;  // Allocate heap.
  if(pFile == NULL)
    return(1);  // Return memory allocation error.
  for(int i=0; i< NumModes; ++i)
  {
// Assign DampConversion[i]:
    *(pDampConversion+i)= (double)1.0;
    pFile->open((const char*)*(pOutFileName+i), ios::out);
    if(!*pFile)
    {
```

```
      cout << "File " << *(pOutFileName+i) << ":\n";
      return(2);  // Return file error.
    }
    PrintHeader(i, pFile);  // Print the header.
    pFile->close();  // Close the file.
  }
  delete pFile;  // Free heap.
  return(NoAlloc);  // Return the memory allocation status.
}

int CClmnData::ICClmnData(CDatDim* pCDatDim)
{
  double Gravity= 32.174;  // Declare and assign Gravity.
// Declare and compute BladeMass:
  double BladeMass= pCDatDim->BladeWeight/Gravity;
// Declare and compute BladeInertia:
  double BladeInertia= BladeMass*pow(pCDatDim->HingeToCG,
                                     2.0)/144.0;
// Compute BladeFreq:
  BladeFreq= sqrt(pCDatDim->BladeK/BladeInertia);
// Compute BladeDamp:
  BladeDamp= pCDatDim->BladeB/(2.0*BladeFreq*BladeInertia);
// Compute ArmRatio:
  ArmRatio= pCDatDim->HingeRadius/pCDatDim->HingeToCG;
// Declare and compute PylonMass:
  double PylonMass= pCDatDim->PylonWeight/Gravity;
// Declare and compute RotorMass:
  double RotorMass= pCDatDim->NumBlades*BladeMass;
// Declare and compute TotalMass:
  double TotalMass= RotorMass+PylonMass;
  MassRatio= 0.5*RotorMass/TotalMass;  // Compute MassRatio.
  NumModes= pCDatDim->NumModes;  // Assign NumModes.
// Allocate heap for array members:
  int NoAlloc= AllocateMemory();
  if(NoAlloc)
    return(NoAlloc);  // Return error.
  double SFreq;  // Declare a holder for StructFreq.
   fstream* pFile;  // Declare a pointer to fstream.
  pFile= new fstream;  // Allocate heap.
  if(pFile == NULL)
    return(1);  // Return error.
  for(int i=0; i<NumModes; ++i)
  {
// Compute SFreq:
    SFreq= sqrt(*(pCDatDim->pStructK+i)/PylonMass);
    *(pStructFreq+i)= SFreq;  // Assign StructFreq.
```

```
// Compute StructDampRatio:
    *(pStructDampRatio+i)= *(pCDatDim->pStructB+i)/
(2.0*SFreq*TotalMass);
// Compute DampConversion:
    *(pDampConversion+i)= 4.0*SFreq*BladeFreq;
// Assign OutFileName[i]:.
    *(pOutFileName+i)= *(pCDatDim->pOutFileName+i);
// Assign ModeLabel[i]:
    *(pModeLabel+i)= *(pCDatDim->pModeLabel+i);
    pFile->open((const char*)*(pOutFileName+i), ios::out);
    if(!*pFile)
    {
      cout << "File " << *(pOutFileName+i) << ":\n";
      return(2);  // Return file error.
    }
    pCDatDim->PrintHeader(i, pFile);  // Print the header.
    pFile->close();  // Close the file.
  }
  delete pFile;  // Free heap.
  return NoAlloc;  // Return NoAlloc.
}

int CClmnData::SetNonArray(int NMod, double BDmp, double BFrq,
                           double ARat, double MRat)
{
  NumModes= NMod;  // Assign NumModes.
  BladeDamp= BDmp;  // Assign BladeDamp.
  BladeFreq= BFrq;  // Assign BladeFreq.
  ArmRatio= ARat;  // Assign ArmRatio;
  MassRatio= MRat;  // Assign MassRatio.
  int NoAlloc= AllocateMemory();  // Allocate memory.
  return(NoAlloc);
}

void CClmnData::SetArray(int Ndx, double SFrq, double SDmp,
                         String *pOFN, String *pML,double DC)
{
  *(pStructFreq+Ndx)= SFrq;  // Assign StructFreq.
  *(pStructDampRatio+Ndx)= SDmp;  // Assign StructDampRatio.
  *(pOutFileName+Ndx)= *(pOFN+Ndx);  // Assign OutFileName.
  *(pModeLabel+Ndx)= *(pML+Ndx);  // Assign ModeLabel.
  *(pDampConversion+Ndx)= DC;  // Assign DampConversion.
}

String* CClmnData::GetFileName(int Mode)
{
```

```
  return(pOutFileName+Mode);  // Return OutFileName[Mode].
}

void CClmnData::ReadData(fstream *File)
{
  *File >> BladeFreq;
  *File >> BladeDamp;
  *File >> ArmRatio;
  *File >> MassRatio;
  File->get();  // Clear the input buffer.
  for(int i=0; i<NumModes; ++i)
  {
    *File >> *(pModeLabel+i);
    *File >> *(pStructFreq+i);
    *File >> *(pStructDampRatio+i);
    File->get();  // Clear the input buffer.
    *File >> *(pOutFileName+i);
  }
}

int CClmnData::GetNumModes()
{
  return (NumModes);
}

void CClmnData::ComputeCoeffs(double* pCoefs, double *pRpm,
                              int *pMode)
{
  double Omega= *pRpm/60.0;  // Convert RPM to Hz.
  double Gamma= sqrt(ArmRatio*Omega*Omega +
                     BladeFreq*BladeFreq);
  double Whirl= Omega - Gamma;  // Compute whirl frequency.
  *(pCoefs+2)= 1.0 + MassRatio*Whirl/(2.0*Gamma);
  *(pCoefs+1)= Whirl*Whirl + *(pStructFreq+ *pMode) *
               *(pStructFreq+ *pMode);
  *(pCoefs)= Whirl*Whirl * *(pStructFreq+ *pMode) *
             *(pStructFreq+ *pMode);
}

double CClmnData::ComputeDampingData(int *pMode,
                                     double *pDampAva,
                                     double *pOmega2)
{
  *pDampAva= *(pDampConversion+ *pMode)* *(pStructDampRatio+
             *pMode)*BladeDamp;
  double DampReq= MassRatio* *(pStructFreq+ *pMode)*
```

```
                  *(pStructFreq+ *pMode);
  DampReq /= 8.0*BladeFreq*(*pOmega2/60.0 -
             *(pStructFreq+ *pMode));
// Return the converted damping required:
  return(*(pDampConversion+ *pMode)*DampReq);

}

double CClmnData::ComputeRpmData(int *pMode,
                                 double *pMaxRootRpm)
{
  double PredRpmRng= sqrt(2.0*MassRatio*(*pMaxRootRpm/60.0 -
                          *(pStructFreq+ *pMode)));
  PredRpmRng *= *(pStructFreq+ *pMode)*sqrt(*(pStructFreq+
                                              *pMode));
  PredRpmRng /= *pMaxRootRpm*(1.0-ArmRatio)/60.0 -
                *(pStructFreq+ *pMode);
  return (PredRpmRng*60.0);  // Return in RPM instead of Hz.
}

double CClmnData::ComputeMaxRoot(int *pMode,
                                 double *pMaxRootRpm)
{
  double PredMaxRoot= *(pStructFreq+ *pMode)*MassRatio;
  PredMaxRoot /= 8.0*(*pMaxRootRpm/60.0 - *(pStructFreq+
                                            *pMode));
  PredMaxRoot= *(pStructFreq+ *pMode)*sqrt(PredMaxRoot);
  return (PredMaxRoot);
}

CClmnData& CClmnData::operator= (const CClmnData& Orig)
{
  if(this == &Orig)
    return(*this);
  else
    MakeCopy(Orig);  // Use the copy maker.
  return(*this);
}

void CClmnData::PrintHeader(int Mode, fstream* pFile)
{
  *pFile << "\nBlade natural frequency: " <<  BladeFreq;
  *pFile << " Hertz.\nBlade lag critical damping ratio: ";
  *pFile <<  BladeDamp << " .\nRatio of lag-hinge-offset";
  *pFile << " to hinge-to-blade CG distance: " << ArmRatio;
  *pFile << ".\nBlade mass ratio: " << MassRatio << ".\n";
```

```
  *pFile << "\nMode:  " << *(pModeLabel+Mode);
  *pFile << "\nSupport structural natural frequency: ";
  *pFile << *(pStructFreq+Mode);
  *pFile << " Hertz." << "\nSupport critical damping ratio: ";
  *pFile << *(pStructDampRatio+Mode) << ".\n\n";
  *pFile << "                                Complex Frequency";
  *pFile << " (Hz)\n      RPM          ";
  *pFile << "         Real                  Imaginary\n";
}

ostream& operator<< (ostream& otop, CClmnData& OutData)
{
  otop << "\nBlade natural frequency: " <<  OutData.BladeFreq;
  otop << " Hertz.\nBlade lag critical damping ratio: ";
  otop <<  OutData.BladeDamp << " .\nRatio of lag-hinge-";
  otop << "offset to hinge-to-blade CG distance: ";
  otop << OutData.ArmRatio << ".\nBlade mass ratio: ";
  otop << OutData.MassRatio << ".";
  for(int i=0; i<OutData.NumModes; ++i)
  {
    otop << "\n\nFor " << *(OutData.pModeLabel+i);
    otop << " mode:\n\nSupport structural natural";
    otop << " frequency: " << *(OutData.pStructFreq+i);
    otop << " Hertz.\nSupport critical damping ratio: ";
    otop << *(OutData.pStructDampRatio+i) << " .";
    otop << "\nOutput data for this mode will be stored in";
    otop << " the file \"" << *(OutData.pOutFileName+i);
    otop << "\".";
  }
  otop << "\n";
  return otop;
}
```

# Appendix V

# CMainData Class

## Specification

```
// clmndata.h
//
// Header and function specifications for clmncpp.C.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#ifndef _CMAINDATA_H
#define _CMAINDATA_H

#include "cclmndat.h"
#include "complex.h"

class CMainData
{
  public:
    CMainData();  // Constructor.
    CMainData(const CMainData& Orig);  // Copy constructor.
    ~CMainData();  // Destructor.
    int ICMainData();  // Invocator.
// Function to find thecomplex roots of the characteristic
// equation.
    int SolveForRoots(double* pCoefs, Complex* pRoots);
// Function to handle error output.
// Error codes:
//  0 - Exit with no error.
//  1 - Memory allocation error (default)
//  2 - File open error
    void ErrMsg(int ErrCode= 0);
    void ErrMsg(int ErrCode , const char* pFileName);
// Function to print the complex root information:
    void PrintResult(fstream *pOutFile, double Rpm,
                     Complex *pRoot);
// Functions to print the RPM range statistics and damping
// information:
    void PrintDampAnal(fstream *pOutFile, double DampProdA,
                       double DampProdR);
```

```
    void PrintRpmStats(fstream *pOutFile, double InstRng,
                       double PInstRng,double MxRRpm,
                       double MxR, double PMxR);
// Function to echo a message in the file:
    void PrintMsg(fstream* pOutFile, const char* pOutMsg);

  public:
   long NumRpm;  // Number of discreet RPM values.
   double RpmStart;  // Starting RPM for analysis.
   double RpmEnd;  // Ending RPM for analysis.
   double RpmSpacing;  // Resolution of RPM spacing.
   String* pDampUnits;  // Units of damping product.
   CClmnData *pCClmnData;  // Pointer to CClmnData.
};

#endif
```

Implementation

```
// cmndata.C

// Implementation of functions for Coleman Analysis code.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#include <stdlib.h>
#include "cmndata.h"
#include "cdatdim.h"

fstream *pDbgFile;  // Declare a pointer to fstream.

CMainData::CMainData()
{
  NumRpm -= NumRpm;  // Initialize to zero.
  RpmStart -= RpmStart;  // Initialize to zero.
  RpmEnd -= RpmEnd;  // Initialize to zero.
  RpmSpacing -= RpmSpacing;  // Initialize to zero.
  pDampUnits= NULL;  // Point to NULL.
  pCClmnData= NULL;  // Point to NULL.
}

CMainData::CMainData(const CMainData& Orig)
{
  NumRpm= Orig.NumRpm;  // Assign Numrpm.
  RpmStart= Orig.RpmStart;  // Assign RpmStart.
  RpmEnd= Orig.RpmEnd;  // Assign RpmEnd.
  RpmSpacing= Orig.RpmSpacing;  // Assign RpmSpacing.
  pDampUnits= new String;  // Allocate heap for DampUnits.
  *pDampUnits= *Orig.pDampUnits;  // Assign DampUnits.
  pCClmnData= new CClmnData;  // Allocate heap for CClmnDat.
  *pCClmnData= *Orig.pCClmnData;  // Assign CClmnDat.
}

CMainData::~CMainData()
{
  delete pDampUnits;  // Free heap.
  delete pCClmnData;  // Free heap.
}

int CMainData::ICMainData()
{
  cout << "\nColeman Instability Analysis - C++ Style !\n";
```

```
String DimFmt;  // Declare a String to hold the dimensional
                // format.
String DataFmt;  // Declare a String to hold the data
                 // format.
int ProperFmt;  // Declare an int for control of input loop.
ProperFmt -= ProperFmt;  // Set to zero.
int IMode;  // Declare an int to hold the mode argument for
            // data type invocation.
IMode -= IMode;  // Set to zero.
int GoodExitState;  // Declare an int to hold the exit
                    // status.
GoodExitState= 1;  // Set GoodExitState true.
while(!ProperFmt)
{
  cout << "\nDo you wish to enter your data";
  cout << " \"I\"nteractively,";
  cout << " from a \"F\"ile\nor \"E\"xit the program ? ";
  cin >> DataFmt;
  DataFmt= DataFmt.UpperCase();
  if(DataFmt == (const char*)"E")
  {
    GoodExitState -= GoodExitState;  // Set GoodExitState
                                     // to false.
    ProperFmt= 1;  // Set ProperFmt true.
  }
  else if(DataFmt == (const char*)"I")
  {
    IMode= 1;  // Set IMode for interactive input.
    ProperFmt= 1;  // Set ProperFmt true.
  }
  else if(DataFmt == (const char*)"F")
    ProperFmt= 1;  // Set ProperFmt true.
  else
  {
    cout << "\n\"" << DataFmt << "\" is not a proper";
    cout << " response !\n";
    ProperFmt -= ProperFmt;  // Set ProperFmt false.
  }
}
if(!GoodExitState)
  return(GoodExitState);  // Return error condition.
ProperFmt -= ProperFmt;  // Set ProperFmt false.
pDampUnits= new String;  // Allocate heap.
if(pDampUnits == NULL)
{
  cout << "\nMemory allocation error !\n";
```

```
    return (-1);  // Return memory error.
  }
  pCClmnData= new CClmnData;  // Allocate heap for CClmnDat.
  if(pCClmnData == NULL)
    return (-1);  // Return memory error.
  int InvRetBad;  // Declare an int to catch the returns from
                  // invocations.
  InvRetBad -= InvRetBad;  // Set to zero.
  while(!ProperFmt)
  {
    cout << "\nAre your data in \"D\"imensional or \"N\"on-";
    cout << "dimensional format,\nor do you wish to ";
    cout << "\"E\"xit the program ? ";
    cin >> DimFmt;
    DimFmt= DimFmt.UpperCase();
    if(DimFmt == (const char*)"E")
    {
      GoodExitState -= GoodExitState;  // Set GoodExitState
                                       // to false.
      ProperFmt= 1;  // Set ProperFmt true.
    }
    else if(DimFmt == (const char*)"D")
    {
      CDatDim* pCDatDim;  // Declare a pointer to CDatDim.
      pCDatDim= new CDatDim;  // Alocate heap.
      if(pCDatDim == NULL)
      {
        GoodExitState= -1;  // Set GoodExitState to memory
                            // error state.
        cout << "\nInsufficient memory !\n";
        break;  // Break the while loop.
      }
// Invoke CDatDim:
      InvRetBad= pCDatDim->ICDatDim(IMode);
      if(InvRetBad)
      {
        GoodExitState= -InvRetBad;  // Set the GoodExitState.
        delete pCDatDim;  // Free heap.
        break;  // Break the while loop.
      }
// Invoke for a dimensional product:
      InvRetBad= !pDampUnits->IString((const char*)"(1/sec)");
// Invoke CClmnDat:
      InvRetBad |= pCClmnData->ICClmnData(pCDatDim);
      if(InvRetBad)
      {
```

```
        GoodExitState= -InvRetBad;  // Set the GoodExitState.
        delete pCDatDim;  // Free heap.
        break;  // Break the while loop.
      }
      delete pCDatDim;  // Free heap.
      *pDampUnits= *pDampUnits + (char)(0xB2);
      ProperFmt= 1;  // Set ProperFmt true.
    }
    else if(DimFmt == (const char*)"N")
    {
// Invoke for a nondimensional product:
      InvRetBad= !pDampUnits->IString((const char*)
                                      "        ");
// Invoke CClmnDat:
      InvRetBad |= pCClmnData->ICClmnData(IMode);
      if(InvRetBad)
        GoodExitState= -InvRetBad;  // Set the GoodExitState.
      ProperFmt= 1;  // Set ProperFmt true.
    }
    else
    {
      cout << "\n\"" << DimFmt << "\" is not a proper";
      cout << " response !\n";
      ProperFmt -= ProperFmt;  // Set ProperFmt false.
    }
  }  // End of WHILE(!ProperFmt).
  if(GoodExitState <= 0)
    return(GoodExitState);  // Return false.
  ProperFmt -= ProperFmt;  // Set ProperFmt false.
  while(!ProperFmt)
  {
    cout << "\nWhat is the starting RPM for the analysis ? ";
    cin >> RpmStart;
    cout << "\nWhat is the ending RPM for the analysis ? ";
    cin >> RpmEnd;
    cout << "\nWhat RPM resolution for the analysis ? ";
    cin >> RpmSpacing;
    if((RpmEnd <= RpmStart) || (RpmSpacing <= 0))
    {
      String YorN;  // Declare a String to catch a yes or no.
      cout << "\nNegative or zero RPM range or RPM";
      cout << " resolution !\n";
      cout << "\nDo you wish to reenter the RPM data";
      cout << " (\"Y\"es or \"N\"o, No";
      cout << " will exit the analysis !) ? ";
      int GetRet= cin.get();  // Clear the input buffer.
```

```
      cin >> YorN;
      YorN= YorN.UpperCase();
      if(YorN != (const char*)"Y")
      {
        GoodExitState -= GoodExitState;  // Set GoodExitState
                                         // to exit state.
        ProperFmt= 1;  // Set ProperFmt true.
        cout << "\nExiting analysis session.\n";
      }
    }
    else
      ProperFmt= 1;  // Set ProperFmt true.
  }
  if(GoodExitState <= 0)
    return(GoodExitState);  // Return the error condition.
// Compute NumRpm:
  NumRpm= (long)(1.0 + (RpmEnd - RpmStart)/RpmSpacing);
  return(GoodExitState);  // If all is well, return true :)
}

int CMainData::SolveForRoots(double* pCoefs, Complex* pRoots)
{
  double R= -0.5* *(pCoefs+1)/ *(pCoefs+2);  // -b/2a
  double I= *(pCoefs+1) * *(pCoefs+1) - 4.0* *(pCoefs+2)*
            *pCoefs;  // bb-4ac
  if(I < (double)0.0)  // Imaginary root:
    I= sqrt(-I)/(2.0* *(pCoefs+2));  // sqrt(4ac-bb)/2a
  else  // All real:
  {
    R= R + sqrt(I)/(2.0* *(pCoefs+2));  // -b/2a +
                                        // sqrt(bb-4ac)/2a
    I= 0.0;
  }
#ifdef _DBG
  *pDbgFile << "\nA= " << *(pCoefs+2) << "\nB= "
  *pDbgFile << *(pCoefs+1);
  *pDbgFile << "\nC= " << *(pCoefs) << "\nR= " << R;
  *pDbgFile << "\nI= " << I << "\n";
#endif
  Complex Root;  // Declare a temporary Complex.
  Root.IComplex(R,I,(const char*)"RI");  // Invoke Root.
  Root.root(pRoots, 2);  // Find the complex square roots.
// Get the absolute value of the real part of the first root:
  double AbReal= fabs(pRoots->GetReal());
  int InstabilityFound;  // Declare a flag to set true for
                         // instability.
```

```
  if(AbReal > 0.00999)
    InstabilityFound= 1;  // There is an instability.
  else
  {
    InstabilityFound -= InstabilityFound;  // Set to zero.
    pRoots->SetReal((double)0.0);  // Clear residual value.
    (pRoots+1)->SetReal((double)0.0);  // Clear residual
                                       // value.
  }
  return (InstabilityFound);
}

void CMainData::ErrMsg(int ErrCode /*= 0 */)
{
  switch(ErrCode)
  {
    case (0):
      break;
    case (1):
    {
      cout << "\nMemory allocation error !\n";
      break;
    }
    case (2):
    {
      cout << "\nError opening file !\n";
      break;
    }
    default:
      cout << "\nUndefined error code !\n";
  }
}

void CMainData::ErrMsg(int ErrCode , const char* pFileName)
{
  switch(ErrCode)
  {
    case (2):
    {
      cout << "\nError opening file \"" << pFileName;
      cout << "\" !\n";
      break;
    }
    default:
      cout << "\nUndefined error code !\n";
  }
```

```
}

void CMainData::PrintResult(fstream *pOutFile, double Rpm,
                            Complex *pRoot)
{
  *pOutFile << "  " << Rpm << "          " << *pRoot << "\n";
}

void CMainData::PrintDampAnal(fstream *pOutFile,
                              double DampProdA,
                              double DampProdR)
{
  *pOutFile << "\n     Damping product                 ";
  *pOutFile << "        Minimum damping\n      ";
  *pOutFile << " available                             ";
  *pOutFile << " product required\n     " << DampProdA;
  *pOutFile << " " << *pDampUnits << "                     ";
  *pOutFile << DampProdR << " " << *pDampUnits << "\n\n";
}

void CMainData::PrintRpmStats(fstream *pOutFile,
                              double InstRng, double PInstRng,
                              double MxRRpm, double MxR,
                              double PMxR)
{
  *pOutFile << "\n                        Predicted\n   ";
  *pOutFile << "Resonance           Resonance             ";
  *pOutFile << "Second\n   Frequency           Frequency  ";
  *pOutFile << "         Crossing\n     Range                ";
  *pOutFile << "Range              Point\n " << InstRng ;
  *pOutFile << " RPM    " << PInstRng << " RPM";
  *pOutFile << "       " << MxRRpm << " RPM\n";
  *pOutFile << "\n                        Predicted";
  *pOutFile << "\n     Maximum           Maximum";
  *pOutFile << "\n      Real              Real";
  *pOutFile << "\n      Root              Root";
  *pOutFile << "\n  " << MxR << " Hz" << "   " << PMxR;
  *pOutFile << " Hz\n";
}

void CMainData::PrintMsg(fstream* pOutFile,
                         const char* pOutMsg)
{
  *pOutFile << pOutMsg << "\n";
}
```

# Appendix VII

# CLMNANAL Application

```
// clmncpp.C

// Coleman Analysis code in C++.

// Michael B. Lance
// Lockheed Engineering and Sciences Company
// Hampton, Virginia

#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "cmndata.h"
#include "complex.h"

extern fstream *pDbgFile;  // Memory for the pointer is
                           // allocated in an include file.

// Function to extract the name of the executable file:
void GetExeName(const char* pArg0, String *pExeName);
// Function to echo help and usage:
void DisplayHelp(String* pExeName);
// Function to give an overview of the code:
void DisplayOverview(String* pExeName);

int main(int argc, char** argv)
{

  int ShowTime;  // Declare a flag to signal time data output.
  ShowTime -= ShowTime;  // Initialize to zero.
  String *pExeName;  // Declare a String on the stack.
  pExeName= new String;  // Allocate heap.
  GetExeName(*argv, pExeName);  // Get the name of the
                                // executable file.
  String *pArg1;  // Declare a String to hold argv[1].
  pArg1= new String;  // Allocate heap.
  pArg1->IString(argv[1]);  // Invoke Arg1.
  *pArg1= pArg1->LowerCase();  // Convert arg[1] to lower
                               // case.
  if(argc == 2)
  {
```

```
    if(*pArg1 == (const char*)"-h")
    {
      DisplayHelp(pExeName);  // Echo the help information.
      return(1);  // Exit with no error.
    }
    else if(*pArg1 == (const char*)"-t")
      ShowTime= 1;  // Set the ShowTime flag.
    else if(*pArg1 == (const char*)"-o")
    {
      DisplayOverview(pExeName);  // Echo the help
                                  // information.
      return(1);  // Exit with no error.
    }
    else if(*pArg1 == (const char*)"-a")
      ;  // Analysis option - run the code.
    else
    {
      cout << "\nInvalid argument !\n";
      DisplayHelp(pExeName);  // Echo the help information.
      return(0);  // Exit with error.
    }
  }
  else if(argc >= 1)
  {
    cout << "\nNo arguments or too many arguments !\n";
    DisplayHelp(pExeName);  // Echo the help information.
    return(0);  // Exit with error.
  }
  delete pExeName;  // Free heap.
  delete pArg1;  // Free heap.
  time_t Time0= time(NULL);  // Declare and initialize a
                             // time_t.
#ifdef _DBG
  if((pDbgFile= new fstream) == NULL)  // Allocate heap and
                                       // verify it.
  {
    pCMainData->ErrMsg(1);  // Call ErrMsg with ErrCode of 1.
    return(0);  // Return with error.
  }
// Open DebugOut:
  pDbgFile->open((const char*)"DebugOut", ios::out);
  if(!*pDbgFile)  // If the file failed to open properly:
  {
// Call ErrMsg with ErrCode of 2:
    pCMainData->ErrMsg(2, (const char*)"DebugOut");
    return(0);  // Return with error.
```

```
  }
  else
    cout << "\nFile \"DebugOut\" open !\n";
#endif
  CMainData *pCMainData;  // Declare a pointer to CMainData.
  pCMainData= new CMainData;  // Allocate heap.
  if(pCMainData == NULL)
  {
    cout << "\nFailed to allocate heap for CMainData !\n";
#ifdef _DBG
    *pDbgFile << "\nFailed to allocate heap for ";
    *pDbgFile << "CMainData !\n";
    pDbgFile->close();
#endif
    return(0);  // Return with error.
  }
  cout.setf(ios::showpoint);  // Set the output format to
                              // show trailing 0's.
  int SetUp= pCMainData->ICMainData();  // Invoke CMainData.
  if(SetUp < 0)
  {
    pCMainData->ErrMsg(-SetUp);  // Call ErrMsg with proper
                                 // ErrCode.
    delete pCMainData;  // Free heap.
#ifdef _DBG
    *pDbgFile << "\nFailed to set up SMainData !\n";
    pDbgFile->close();
#endif
    return(0);
  }
  else if(SetUp == 0)
  {
    pCMainData->ErrMsg();  // Call ErrMsg.
    delete pCMainData;  // Free heap.
    cout << "Good bye !\n";
#ifdef _DBG
    *pDbgFile << "\nUser termination !\n";
    pDbgFile->close();
#endif
    return (1);
  }
  int InstabilityFound;  // Flag to signal a real root.
  int RangeStarted;  // Flag to signal that the unstable RPM
                     // range has been entered.
  double Rpm= (double)0.0;  // RPM tracker.
// Assign an RPM increment:
```

```
  double DelRpm= pCMainData->RpmSpacing/20.0;
  double StartInstability= (double)0.0;  // Start of unstable
                                          // RPM range.
  double EndInstability= (double)0.0;  // End of unstable RPM
                                       // range.
  double PredInstRng= (double)0.0;  // Predicted unstable RPM
                                    // range.
  double MaxRealRoot= (double)0.0;  // Maximum real root.
  double PredMaxRealRoot= (double)0.0;  // Predicted maxium
                                        // real root.
  double MaxRootRpm= (double)0.0;  // RPM at which the maxium
                                   // real root occurs.
  double RootHolder= (double)0.0;  // Temporary holder of
                                   // Real part of root.
  double *pCoeffArray= NULL;  // Pointer to array of
                              // coefficients.
  double DampAva= (double)0.0;  // Damping product available.
  double DampReq= (double)0.0;  // Damping product required.
  fstream *pOutFile;  // Pointer to the output file.
  Complex *pRoots= NULL;  // Pointer to array of Complex
                          // roots.
  if(((pCoeffArray= new double[3]) == NULL) ||
     ((pRoots= new Complex[2]) == NULL))
  {
    pCMainData->ErrMsg(1);
#ifdef _DBG
    *pDbgFile << "\nFailed to allocate heap for pCoeffArray";
    *pDbgFile << " or pRoots !\n";
    pDbgFile->close();
#endif
    delete pCMainData;  // Free heap.
    return(0);
  }
  pOutFile= new fstream;  // Allocate heap.
  if(pOutFile == NULL)
  {
    pCMainData->ErrMsg(1);  // Call ErrMsg with ErrCode of 1.
    delete pCMainData;  // Free heap.
#ifdef _DBG
    *pDbgFile << "\nFailed to allocate heap for pOutFile !\n";
    pDbgFile->close();
#endif
    return(0);  // Exit with false.
  }
// Output in scientific notation:
  pOutFile->setf(ios::scientific, ios::floatfield);
```

```
// Make +'s show in front of positive numbers:
  pOutFile->setf(ios::showpos);

  pOutFile->precision(4);  // Set the number of digits after
                           // radix to 4.
  String FileName;  // Declare a String to hold OutFileName.
  time_t Time1= time(NULL);  // Declare and initialize a
                             // second time_t.
  for(int i=0; i< pCMainData->pCClmnData->GetNumModes(); ++i)
  {
    FileName= *pCMainData->pCClmnData->GetFileName(i);
    pOutFile->open((const char*)FileName,
                   ios::app|ios::nocreate);
    if(!*pOutFile)
    {
      pCMainData->ErrMsg(2, FileName);  // Call ErrMsg.
#ifdef _DBG
    *pDbgFile << "\nFailed to properly open \"";
    *pDbgFile << FileName << "\" !\n";
    pDbgFile->close();
#endif
      break;
    }
    Rpm= pCMainData->RpmStart;  // Initialize RPM.
    StartInstability -= StartInstability;  // Initialize to
                                           // zero.
    RangeStarted -= RangeStarted;  // Unset RangeStarted.
// While the Rpm is in range:
    while(Rpm < pCMainData->RpmEnd+DelRpm)
    {
      pCMainData->pCClmnData->ComputeCoeffs(pCoeffArray,
                                            &Rpm, &i);
#ifdef _DBG
    *pDbgFile << "\nRpm= " << Rpm;
#endif
      InstabilityFound= pCMainData-SolveForRoots(pCoeffArray,
                                                 pRoots);
      if(InstabilityFound)
      {
        EndInstability= Rpm;  // Assign EndInstability.
        if(!RangeStarted)
        {
          StartInstability= Rpm;  // Assign StartInstability.
// Assign MaxRealRoot:
          MaxRealRoot= pRoots->GetReal();
          MaxRootRpm= Rpm;  // Assign MaxRootRpm;
```

```
          RangeStarted= 1;  // Set RangeStarted.
        }
        else
        {
          RootHolder= pRoots->GetReal();  // Store real part
                                          // of root.
          if(RootHolder > MaxRealRoot)
          {
             MaxRealRoot= RootHolder;  // Re-assign
                                       // MaxRealRoot.
             MaxRootRpm= Rpm;  // Re-assign MaxRootRpm.
          }
        }
// Print the results:
        pCMainData->PrintResult(pOutFile, Rpm, pRoots);
      }
      else if(((int)(Rpm/pCMainData->RpmSpacing)*pCMainData-
                RpmSpacing) == Rpm)
// Print the results:
        pCMainData->PrintResult(pOutFile, Rpm, pRoots);
      else if((Rpm == pCMainData->RpmEnd) ||
              (Rpm == pCMainData->RpmStart))
// Print the results:
        pCMainData->PrintResult(pOutFile, Rpm, pRoots);
      Rpm += DelRpm;  // Increment RPM.
    }  // End WHILE(Rpm < pCMainData->RpmEnd+DelRpm).
    if(RangeStarted)  // If an unstable range was encountered:
    {
      DampReq= pCMainData->pCClmnData-
               ComputeDampingData(&i,&DampAva,&MaxRootRpm);
      pCMainData->PrintDampAnal(pOutFile, DampAva, DampReq);
// Print damping information:
      PredInstRng= pCMainData->pCClmnData-
                   ComputeRpmData(&i,&MaxRootRpm);
      PredMaxRealRoot= pCMainData->pCClmnData-
                       ComputeMaxRoot(&i,&MaxRootRpm);
      pCMainData->PrintRpmStats(pOutFile, (EndInstability-
                                StartInstability),
                                PredInstRng,MaxRootRpm,
                                MaxRealRoot, PredMaxRealRoot);
    }
    else
      pCMainData->PrintMsg(pOutFile, (const char*)
                           "\nNo instability found !");
    pOutFile->close();  // Close OutFile[i].
  }  // End FOR(i< pCMainData->pDatNonDim->GetNumModes()).
```

```
  delete pOutFile;  // Free heap.
  delete [] pCoeffArray;  // Free heap.
  delete [] pRoots;  // Free heap.
  delete pCMainData;  // Free heap.
  cout << "\nColeman Analysis complete !\n";
#ifdef _DBG
    *pDbgFile << "\nReached end of .C file !\n";
    pDbgFile->close();
#endif
  time_t Time2= time(NULL);  // Declare and initialize a
                             // time_t.
  if(ShowTime)
  {
    cout << "\nData entry time: " << difftime(Time1, Time0);
    cout << " seconds.\n";
    cout << "Computation time: " << difftime(Time2, Time1);
    cout << " seconds.\n";
    cout << "Total elapsed time: ";
    cout << difftime(time(NULL), Time0) << " seconds.\n";
  }
  return(1);
}

void GetExeName(const char* pArg0, String *pExeName)
{
  int LenArg0= strlen(pArg0);  // Determine the lenght of the
                               // string.
  String SrchString;  // Declare a String on the stack.
  SrchString.IString(pArg0);  // Invoke SrchString.
// Find the last "\":
  int SlashIndex= SrchString.FindLast((const char*)"\\");
  if(SlashIndex < 0)  // If did not find \, look for /.
    SlashIndex= SrchString.FindLast((const char*)"/");
// Find the last ".":
  int DotIndex= SrchString.FindLast((const char*)".");
  if(DotIndex < 0)  // Command does not have a . in the name.
    DotIndex= LenArg0;  // The . would be after the end of
                        // arg0.
  int StartIndex= SlashIndex+1;  // Compute the start index
                                 // of the command.
  int NumChar= DotIndex-StartIndex;  // Determine length of
                                     // executable name.
// Get the executable name text:
  *pExeName= SrchString.GetChars(StartIndex, NumChar);
}
```

```
void DisplayHelp(String* pExeName)
{
  cout << "\nWelcome to the Coleman Analysis program !\n";
  cout << "\nCopyright 1996";
  cout << "\nMichael B. Lance\nLockheed Engineering &";
  cout << " Sciences Company\n";
  cout << "Hampton, Virginia\n";
  cout << "\nThe usage of this program is:\n\n    ";
  cout << *pExeName << " [-h/H] [-o/O] [-a/A] [-t/T]\n\n";
  cout << "    -h:  Display this help file.\n";
  cout << "\n    -a:  Run the analysis.\n";
  cout << "\n    -t:  Run the analysis and display time";
  cout << " statistics.";
  cout << "\n         These statistics are: ";
  cout << "\n           Time elapsed for data entry.";
  cout << "\n           Time elapsed for computations.";
  cout << "\n           Total time elapsed.\n";
  cout << "\n    -o:  Display an overview of the";
  cout << " program.\n\n";
}

void DisplayOverview(String* pExeName)
{
  cout << "\nWelcome to the Coleman Analysis program !\n";
  cout << "\nCopyright 1996";
  cout << "\nMichael B. Lance\nLockheed Engineering &";
  cout << " Sciences Company\n";
  cout << "Hampton, Virginia\n";
  cout << "\nThis is a brief overview of " << *pExeName;
  cout << " and how to run it.\n"(Enter to continue)";
  int GetRet;  // Declare an int to catch input chars.
  while((GetRet= cin.get()) != 10);  // Wait for \n.
  cout << "\nHistory:\n";
  cout << "This program is an adaptation of the theory of";
  cout << " Robert P. Coleman and\nArnold M. Feingold as";
  cout << " presented in NACA Report 1351, 1958.  This\n";
  cout << "theory provided a method for the analysis of";
  cout << " multiple-bladed rotor\nsystems to determine";
  cout << " the system susceptibility to ground resonance.";
  cout << "  Their\ntreatment also provided a simple means";
  cout << " for determining the required\nproduct of rotor";
  cout << " and chassis damping factors to suppress the";
  cout << " resonance.\nThis C++ program is based on a";
  cout << " FORTRAN 77 version of a similar code.\n";
  cout << "The FORTRAN 77 code was converted by the author";
  cout << " from a FORTRAN 4 code written by Carl Freeman.\n";
```

```
  cout << "(Enter to continue)";
  while((GetRet= cin.get()) != 10);  // Wait for \n.
  cout << "\nHow to run " << *pExeName << ":\n";
  cout << *pExeName << " can be run with interactive input";
  cout << " or with data from a file.\nThese data can be";
  cout << " in a dimensional format (inches, pounds, etc.),";
  cout << " or the\nappropriate non-dimensional format";
  cout << " (length ratios, mass ratios, etc.).\n";
  cout << "  Dimensional data required/file format:\n";
  cout << "    Number of modes to analyze\n    Number of";
  cout << " blades.\n    Rotor radius (in)\n    Hinge";
  cout << " radial station (in)\n    Blade radius of ";
  cout << "gyration (in)\n    Blade weight (lb)\n    Pylon";
  cout << " weight (lb)\n    Lag hinge radial spring";
  cout << " constant (ft-lb/rad)\n    Lag hinge radial";
  cout << " damping constant (ft-lb-sec/rad)\n";
  cout << "    For each mode:\n      Descriptive mode label";
  cout << " string\n      Pylon spring constant (lb/ft)\n;"
  cout << "      Pylon damping constant (lb-sec/ft)\n";
  cout << "      Name of file to store results for this";
  cout << " mode\n(Enter to continue)";
  while((GetRet= cin.get()) != 10);  // Wait for \n.
  cout << "  Non-dimensional data required/file format:\n";
  cout << "    Number of modes to analyze\n    Blade ";
  cout << "lead/lag natural frequency (Hz)\n    Blade";
  cout << " lead/lag critical damping ratio\n    Ratio of";
  cout << " lead/lag hinge radial location to the blade";
  cout << " radius of gyration.\n    One-half the ratio ";
  cout << "of the sum of the rotor blade masses to the sum";
  cout << " of blade\n    masses and pylon mass\n    For";
  cout << " each mode:\n      Descriptive mode label";
  cout << " string\n      Pylon natural frequency (Hz)\n";
  cout << "      Pylon critical damping ratio\n      Name";
  cour << " of file to store results for this mode\n\n";
  cout << *pExeName << " will prompt for the rpm at which";
  cout << " to start the analysis, the rpm\nat which to";
  cout << " end the analysis, and the rpm increment to";
  cout << " use when displaying \nthe results.\nNOTE:  ";
  cout << "If an instability is encountered, results ";
  cout << "will be\ndisplayed at 20 times this frequency!\n";
}
```