ENGSCI331 Computational Techniques: Eigenvectors module coding help

This is a document designed to be read with the template files you have been given. There is a source file called myEigenMain.cpp containing a basic main function, a header file called myEigenFunctions.h containing some function declarations, and the corresponding function bodies in the source file called myEigenFunctions.cpp. These files are to get you started on this assignment – you will need to edit them to make them do what you need them to but should compile, build and run straight away. The code snippets below have been taken from the template code provided, so have a look in the files first and refer to this document for what you don't understand.

C++ std namespace and stream operations

Like its name suggests, C++ is a language that is more advanced than the pure C that you were taught in ENGGEN131. Its greatest strengths come from the fact that it is an *object-oriented* programming language. While we will not go into the detail of everything that means in this course, we can take advantage of some of the handy shortcuts and functions that C++ offers.

Standard screen output and input

C++ syntax

```
// In the global declarations
#include <iostream>

using namespace std;

// In the main function
cout << "Enter n:";
cin >> n;
cout << endl;
```

C equivalent

```
// In the global declarations
#include <stdio.h>

// In the main function
printf(" Enter n:");
scanf("%d",&n);
printf("\n");
```

Matlab equivalent

```
tol = input('Enter n: ');
```

You'll see that the C++ syntax makes use of the >> and << operators to get and put information to the standard output stream, the screen (cin >> for input and cout << for output). These same operators can be used with file streams to get and write information, as described below.

File output and input

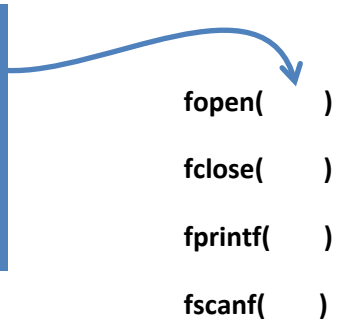
When you learnt about file opening, reading and writing in ENGGEN131 you would have used the FILE macro, defined a handle variable (probably called fid), and used the fopen, fclose, fprintf and fscanf functions to read and write. In C++ the ideas are similar – we still have to identify the file to open, close, write and read – but instead of having to specify the file each time we do, the object-orientated nature of C++ recognises that every file will need these functions, so maybe it's better to group the functions around the file instead using the file as input to the functions.

I am a **C++ file stream** object. In my suitcase of useful functions I have:

- open
- close
- << to write to me
- >> to read from me

These functions belong to me and are not used by any other streams. Each file stream has its own associated functions.

I am a **C file handle**. I must be used as an input argument to independent functions. These functions do not belong to me and are only associated with me through the argument list.



So what does this mean for you? Have a look at the snippet below.

C++ syntax

```
// In the main function

ifstream infile;
ofstream outfile;
string filename;

cout << "Please enter the filename to read:";
cin >> filename;
cout << endl;

infile.open(filename);

if(infile.is_open()) {
   infile >> n;
   for(int j = 0; j < n; j++) {
      for(int i = 0; i < n; i++) {
         infile >> A[i][j];
      }
   }
}
else
   cout << "ERROR: Unable to open file called "<< filename << endl;
```

You'll see how we define the input file stream "infile" to be of type "ifstream" (short for input file stream). Then, we can perform some useful operations using the file stream's own functions (notice that the use of the full-stop after the object name (here "infile") is how you access that object's associated functions):

- Opening the file using the infile.open() command. The only argument for this function is the filename.
- Checking that the file opened correctly using the is_open() function associated with the infile object
- Reading information from the file using the infile >> operator.

You should also see how the file input and output mirrors what we've done earlier with the screen input and output using the >> and << operators to push and get information from the streams.

Dynamic memory allocation

You may not have encountered the "`new`" and "`delete`" operators before, so here's how they work. Declaring a 1D array is the same as declaring a pointer to the beginning of the array:

```
double *b = NULL;            // b is a 1D array of doubles, its
                             // address is the null pointer, 0x000000
```

Allocating space for the array using the "`new`" operator:

```
b = new double [n];          // b now has space for n elements
```

Read the last statement like:

"I want to make `b` to be a `new` array of `double` items that is `[n]` long."

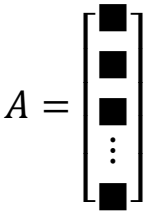
There are also allocations of 2D and 3D arrays, which you can think of as follows. Declaring "A" as a 2D array of doubles:

```
double **A = NULL;           // A is a matrix of doubles, its address
                             // is currently the null pointer 0x000000
```

Next we allocate the first dimension of the A array to be a list of 1D double arrays:

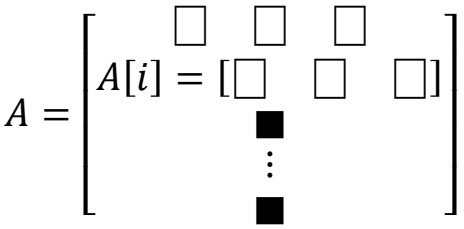
```
A = new double* [n];
```

This means that each of the filled squares on the right has a type of `double*` that is, a 1D double array (that is not yet allocated).



Now we need to loop through the list of arrays stored inside A (the rows we created above) and allocate space for each of them:

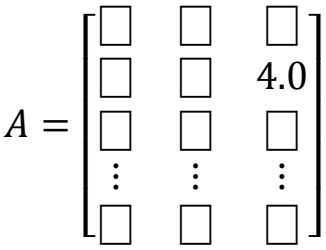
```
for(int i = 0; i < n; i++) {
     A[i] = new double [m];
}
```



This is allocating space for a 1D array of `m` `double` type elements at each of the `n` rows of A.

You can access the individual values stored inside A using two sets of square brackets:

```
A[1][2] = 4.0;
```



Remember that in C the indices start at 0!

Note that C does not distinguish between rows and columns as such, all it sees are the first and second (etc) indices. You need to make sure that your first index iterates only as far as you have allocated it (ie: in this example the first index can only take values between 0 and `n`, and the second index can only take values between 0 and `m`). It's up to you whether you conceptually think of the first index as being the row number or the column number. Just be consistent!

Good Housekeeping

Whenever you allocate something you must also make sure you free the memory when you're finished. Failing to do this is called a "memory leak" and is a big programming no-no. Deallocation is done just like allocation in reverse:

Deletion

```
for(int i = 0; i < n; i++) {
   delete [] A[i];
}

delete [] A;
```

Allocation

```
A = new double* [n];

for(int i = 0; i < n; i++) {
   A[i] = new double [m];
}
```

The first call delete statement removes the memory associated with the second index (in the diagrams above, the row i). It does this for each of the rows (just as we allocated space for each row earlier). The last call to delete removes the 1D array of `double*` objects, just as we initially allocated a 1D array of these objects.

It's a good idea to delete memory at the same level in the code as you allocate it. In other words, if you allocate an array in the main function, delete it later on in the main function too (as opposed to inside a function). Likewise, if you need to allocate an array inside a function, it's a good idea to delete it inside that same function too. This isn't a strict rule, it just makes debugging memory errors a lot more straightforward and lessens the chance that you'll have a memory leak.

Function scope and returns

There are three possible methods in which a function can give information back to where it is called from. **The first** is to alter a global variable. This is not generally considered good programming practise, so we will try and stay away from that!

The second is to use the return keyword to give back a variable or value from the function. This means that somewhere the equals sign (the assignment operator) will be used to make sure that the returned value is stored somewhere. For example:

In the calling function

```
double x = 3.0,
       y = 4.0,
       h = 0.0;

h = pythagorus(x,y);
```

In double pythagorus (double a, double b)

```
double hyp = 0, hypSqr = 0;

hypSqr = a*a + b*b;
hyp = power(hypSqr,0.5);

return hyp;
```

There are several points to note here. First of all note the scope of the variable names. When we are in the calling function the variable that we want to square is called x. When it is inside the square function it seems to have changed its name to a. This is not really the case but highlights the important issue of *function scope*.
When variables are passed through the argument list of a function it is an assignment process. This means that the pythagorus function goes through the argument list and says:

a = whatever value is passed in at the time of calling, here it is the *value* of x, or 3.0
b = whatever value is passed in at the time of calling, here it is the *value* of y, or 4.0

It then calculates the answer by taking the square root of the two squared sides and returns it to the calling function. At this stage, the calling function says:

h = whatever value the pythagorus function has returned, here it is the *value* 5.0

The use of the equals sign is important. Values passed into a function have to go through the equals sign when they come through the argument list, and the same for values returned from a function. This means that if we changed the value for x later on, the value of h would not change automatically, we'd have to recall the pythagorus function to calculate it again.

Most of this is quite self-evident, but the effects it has are important. This kind of working means that:

a) We can only return one variable from a function (note C is different to Matlab in this respect). So how do we return more than one answer from a single function call?
b) What about if we want to alter the value of an input variable during the function call?

The answer to both these ideas is **the third way** to get information from a function, and works by passing the addresses of variables instead of their values. If we pass in the address of a variable in the argument list of a function, we can manipulate the content of that address all we like, and the address itself doesn't change. This is done using the ampersand (address) operator. Consider a new argument list:

```
void pythagorus (double a, double b, double &hyp, double &hypSq)
```

The statement `double &hyp` is read as "the address of the variable passed in in the third argument position, the value of which I will refer to as hyp". Now we could alter our pythagorus function to return both the squared hypotenuse value as well as the hypotenuse itself – that's two outputs from one function. This is how we'd do it:

In the calling function

```
double x = 3.0,
       y = 4.0,
       h = 0.0,
       h2 = 0.0;

pythagorus(x,y,h,h2);
```

In void pythagorus (double a, double b, double &hyp, double &hypSq)

```
hypSq = a*a + b*b;
hyp = power(hypSq,0.5);
```

Note the differences:

- The pythagorus function now takes four arguments
- It doesn't return anything (has type void)

- We don't declare the variables double hyp = 0, hypSqr = 0 inside the pythagorus function because now they are passed in through the argument list.
- We do declare a new double h2 variable in the calling function so that we can pass it in to the argument list of the pythagorus function.

Note the similarities:

- We can still use the variables hyp and hypSq inside the function as we normally would, the only difference is in the argument list.

Important:

- Now that we're using addresses to pass information around, we have to use variables to do it. Constants – like numbers – don't have an address, so calling the function like this would be illegal:

  ```
  pythagorus(x,y,0.0,0.0);
  ```

  because there's no variable to store the returned values of hyp and hyp2 in. However, this:

  ```
  pythagorus(3.0,4.0,h,h2);
  ```

  is legal because the only two variables passed in by their address (using the & operator in the argument list) are the third and fourth arguments.

Exercises

Shown below are three routines written in C/C++ code. Use these code snippets to refresh your memory on C coding, and to identify which aspects might be new to you.

1) Figure out what each function is doing, and decide on better function names for them
2) Highlight any areas whose syntax you don't understand, and see if it's explained earlier in this document.
3) Suggest improvements to the routines that you (as an experienced, thorough, good programmer) would make.
4) Highlight what could go wrong and potential places for errors to occur, and think about how you would test for them or prevent them.

```
void Function1(double **A, int n, int m, double **B, double &C)
{
    assert(A);
    assert(m>0);
    assert(n>0);
    assert(B);

    C = 0.0;

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            C += A[i][j]*B[i][j];
        }
    }
}
```

```
void Function2(double **A, int n, int m, double **B, int l, double **C)
{
    for (int i = 0; i < n; i++) {
        for (int k = 0; k < l; k++ ) {
            C[i][k] = 0.0;
            for(int j = 0; j < m; j++) {
                C[i][k] += A[i][j]*B[k][i];
            }
        }
    }
}
```

```
void Function3(double **A, int m, int n, double **B)
{
    for (int j = 0; j < m; j++) {
        for (int i = 0; i < n; i++) {
            B[j][i] = A[i][j];
        }
    }
}
```