P4: Matrices and Vectors

Random Vector - 23/02/21

To create a vector of random double I had to use the random and vector libraries, this gave me the functions I required.

When initially initialising dre I used the default_random. However, due to issues with my MinGW installation this was deterministic. To fix this I instead initialised it with the time to get a true random string.

```cpp
#include <random>
#include <ctime>
#include <vector>

int main()
{
    std::default_random_engine dre(std::time(NULL));
    std::uniform_real_distribution<double> dr(10, 20);

    std::vector<double> v;
    int vec_size;

    cout << "Enter the size of your vector." << endl;
    cin >> vec_size;
    for (int i = 0; i < vec_size; i++)
    {
        v.push_back(dr(dre));
    }

    for (int i = 0; i < vec_size; i++)
    {
        cout << v[i] << ", ";
    }
}
```

Matrix - 23/02/21

To create a matrix I used a vector of vectors. This allowed me to make it easily scalable. My implementation means it has to be a square matrix but as this is intended for a tridiagonal matrix this is not an issue.

To correctly initialise the matrix I had to change the code. I used a helpful suggestion I found on stackoverflow to initialise it.

```
    int mat_size;
    cout << "Enter the size of your Matrix." << endl;
    cin >> mat_size;

    std::vector<std::vector<double>> M(mat_size); //Luchian Grigore at
https://stackoverflow.com/questions/12375591/vector-of-vectors-to-create-matrix
    for (int i = 0; i < mat_size; i++)
    {
        M[i].resize(mat_size);
    }

    for (int i = 0; i < mat_size; i++)
    {
        for (int j = 0; j < mat_size; j++)
        {
            M[i][j] = dr(dre);
        }
    }

    for (int i = 0; i < mat_size; i++)
    {
        for (int j = 0; j < mat_size; j++)
        {
            cout << M[i][j] << " ";
        }
        cout << endl;
    }
```

Linear Equation Solver - 23/02/21

To convert the function from c to c++ I converted all the the pointers to vectors. I also found it useful to change the naming convention to something that is more intuitive.

```cpp
std::vector<double> TridiagonalSolve(double c1, std::vector<double> &diag,
std::vector<double> &vect, int n)
{
    int i;
    std::vector<double> off_diag(n);
    std::vector<double> output(n);
    double id;

    // Set the off diagonal elements
    for (i = 0; i < n; i++)
    {
        off_diag[i] = c1;
    }

    //forward bit
    off_diag[0] /= diag[0]; //if /0 rearrange equations
    vect[0] /= diag[0];

    for (i = 1; i < n; i++)
    {
        id = diag[i] - off_diag[i - 1] * c1;
        off_diag[i] /= id; // Last value calculated is redundant.
        vect[i] = (vect[i] - vect[i - 1] * c1) / id;
    }

    // Now back substitute.
    output[n - 1] = vect[n - 1];

    for (i = n - 2; i >= 0; i--)
    {
        output[i] = vect[i] - off_diag[i] * output[i + 1];
    }

    return output;
}
```

Tridiagonal Solver - 24/02/21

To test the implementation of the function I created a main program that would create a tridiagonal matrix and a vector. Find the product. Then using the matrix and the product it finds the original vector again.

```cpp
int main()
{
    //determine size of the matrix and vector
    int size;
    cout << "Enter the size of your Matrix." << endl;
    cin >> size;

    //generate tridiagonal matrix and vector
    std::vector<double> vec = VectorGenerate(size);
    std::vector<std::vector<double>> tridiag = TridiagonalGenerate(size);

    cout << endl
         << "Your tridiagonal matrix is:" << endl;
    TridiagonalPrint(size, tridiag);

    cout << "Your vector is:" << endl;
    VectorPrint(size, vec);

    //multiply tridiagonal matrix and vector
    std::vector<double> product = multiplication(size, vec, tridiag);

    cout << endl
         << "The product vector is:" << endl;
    VectorPrint(size, product);

    //solve for original vector from matrix and product
    std::vector<double> diag;
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            if (i == j)
            {
                diag.push_back(tridiag[i][j]);
            }
        }
    }

    std::vector<double> original = TridiagonalSolve(tridiag[0][1], diag,
product, size);

    cout << endl
         << "The original vector was:" << endl;
    VectorPrint(size, original);
}
```

The result on the command line shows that it is correctly calculating the original vector.

```
Enter the size of your Matrix.
6

Your tridiagonal matrix is:
10.7061 17.1776       0       0       0       0
17.1776 13.0758 17.1776       0       0       0
      0 17.1776 18.2992 17.1776       0       0
      0       0 17.1776 10.7528 17.1776       0
      0       0       0 17.1776 10.3502 17.1776
      0       0       0       0 17.1776 10.0471

Your vector is:
19.1133 17.4045 15.8379 16.5649 11.1185   13.23

The product vector is:
503.594 827.953 873.334 641.165 626.884 323.911

The original vector was:
19.1133 17.4045 15.8379 16.5649 11.1185   13.23
```

VectorGenerate() was taken from my main function on page 32.
TridiagonalGenerate() was a modified version of the main function on page 33. Modified so it would only fill the diagonal.

```
    double rand = dr(dre);
    for (int i = 0; i < mat_size; i++)
    {
        for (int j = 0; j < mat_size; j++)
        {
            if (i - 1 <= j && j <= i + 1)
            {
                mat[i][j] = rand;
            }
            else
            {
                mat[i][j] = 0;
            }
            if (i == j)
            {
                mat[i][j] = dr(dre);
            }
        }
    }
```

VectorPrint() and TridiagonalPrint() were also taken from the main functions.

The multiplication was also handled by its own function. It would take in the reference to the matrix and vector and create a vector product.

```
std::vector<double> multiplication(int size, std::vector<double> &vec,
std::vector<std::vector<double>> &tridiag)
{
    std::vector<double> product;
    for (int i = 0; i < size; i++)
    {
        long double sum = 0;
        for (int j = 0; j < size; j++)
        {
            sum += tridiag[i][j] * vec[j];
        }

        product.push_back(sum);
    }
            return product;
}
```

Complex Matrices - 24/02/21

To implement my complex class from the previous lab I included a header file with the definition for the complex class. I also ported across the code to print the matrix in cartesian form.

```
void print_cartesian(complex a)
{
    if (a.Im() >= 0)
    {
        std::cout << std::setw(7) << a.Re() << " + j" << std::setw(7) << a.Im();
    }
    else
    {
        std::cout << std::setw(7) << a.Re() << " - j" << std::setw(7) <<
a.conj().Im();
    }
}
```

I then found and replaced every double data type and changed it to the complex data type. To generate a random complex number I created a function that would return the constructor with two random doubles.

```
complex ComplexRandom(void)
{
    return complex(dr(dre), dr(dre));
}
```

The outputfor this worked, displaying it in complex form. However, it is not particularly intuitive to follow so if I had more time on this lab I would create a morem user friendly interface.

```
Enter the size of your Matrix.
4

Your tridiagonal matrix is:
15.9028 + j11.9031 11.3228 + j19.0814        0 + j        0        0 + j        0
11.3228 + j19.0814 16.3886 + j18.9199 11.3228 + j19.0814        0 + j        0
      0 + j        0 11.3228 + j19.0814  18.512 + j13.5472 11.3228 + j19.0814
      0 + j        0        0 + j        0 11.3228 + j19.0814 11.3188 + j12.0052

Your vector is:
16.8854 + j12.0728 18.0009 + j13.3699 17.6107 + j11.6462 10.8323 + j12.9583

The product vector is:
73.5263 + j887.848 -19.9459 + j1486.49 -7.6713 + j1302.46 -55.7826 + j744.621

The original vector was:
16.8854 + j12.0728 18.0009 + j13.3699 17.6107 + j11.6462 10.8323 + j12.9583
```