ECE 204 *Numerical methods*

Project 1

In this project, we will look at a C++ program for approximating solutions to a Markov chain; this will be similar to solving $x = f(x)$ using fixed-point iteration. We will also investigate the properties of stochastic matrices. This project will be submitted entirely on Crowdmark.

Programming example

In C++, the functional library has a template class that takes a function as an argument to a constructor. The type is of the form:

```
std::function<return_type(param1_type, param2_type, …)>
```

While this does not look like `int` or `double`, it is in fact a type in C++.

For example, you could define the function

```
double quadratic( double x ) {
    return x*x - 3.0*x + 2.7;
}
```

You could now assign this function to a local variable `f` inside of `main()` or some other function:

```
// Initialize 'f' to the function 'quadratic'
//  - Now, when you call 'f', it will call the function quadratic(...)
std::function<double(double)> f{ quadratic };
std::cout << f( 1.1 ) << std::endl;
```

The output would be `0.61`. Because `f` is a local variable, it can now be assigned a new value, so suppose there is a second function

```
double quadratic2( double x ) {
    return x*x + 3.0*x - 8.3;
}
```

You can now continue to execute the following code:

```
// Assign to 'f' the function 'quadratic2'
f = quadratic2;
std::cout << f( 1.1 ) << std::endl;
```

The output is now `-3.79`.

There are functions in the standard library that take an argument and return one, such as std::sin defined in the `cmath` library, but this function is overloaded, so if you want to assign this function to a variable, you must indicate which version is to be used:

```
f = static_cast<double(*)(double)>( std::sin );
std::cout << f( 1.1 ) << std::endl;
```

The output is now `0.891207`.

We can now define a function that takes such a function as an argument:

```
void print(
    std::function<double(double)> f,
    double                        lower,
    double                        upper,
    unsigned int                  n
) {
    double h{ (upper - lower)/n };

    for ( unsigned int k{0}; k <= n; ++k ) {
        double x{ lower + k*h };

        std::cout << "f(" << x << ") = " << f( x ) << std::endl;
    }
}
```

Inside `main()`, we can now call this function:

```
int main() {
    print( quadratic, -1.0, 1.0, 4 );
    std::cout << std::endl;
    print( static_cast<double(*)( double )>(std::sin), -1.0, 1.0, 4 );

    return 0;
}
```

The output yields the quadratic polynomial and the sine function:

```
f(-1) = 6.7
f(-0.5) = 4.45
f(0) = 2.7
f(0.5) = 1.45
f(1) = 0.7

f(-1) = -0.841471
f(-0.5) = -0.479426
f(0) = 0
f(0.5) = 0.479426
f(1) = 0.841471
```

In addition, C++ now allows lambda expressions that allow you to specify a function inline. These are called *lambda* expressions:

```
int main() {
    print( quadratic, -1.0, 1.0, 4 );
    std::cout << std::endl;
    print( [](double x){ return x*x; }, -1.0, 1.0, 4 );

    return 0;
}
```

The output, as you may expect, is

```
f(-1) = 1
f(-0.5) = 0.25
f(0) = 0
f(0.5) = 0.25
f(1) = 1
```

This means that you do not have to declare, and then define the function. Instead, you simply author the function where it is needed. If a function is sufficiently terse, this is sometimes a good option. As for where not to use lambda expressions, this is probably a good example, where the lambda expression returns the minimum of the two arguments, and then immediately calls it on two values; in this case, 3 and 4:

```
std::cout << [](double x, double y){ return (x <= y) ? x : y; }( 3, 4 )
          << std::endl;
```

We can now write a function that performs fixed-point iteration:

```
double fixed_point(
    std::function<double(double)> f,
    double x0,
    double eps_step,
    unsigned int max_iterations
) {
    for ( unsigned int k{1}; k <= max_iterations; ++k ) {
        double x1{ f( x0 ) };

        if ( std::abs( x0 - x1 ) < eps_step ) {
            return x1;
        } else {
            x0 = x1;
        }
    }

    throw std::runtime_error{
        "Fixed-point iteration did not converge"
    };
}
```

We can now call it:

```
std::cout << fixed_point( quadratic,  0.0, 1e-5, 100 ) << std::endl;
std::cout << fixed_point( quadratic2, 0.0, 1e-5, 100 ) << std::endl;
std::cout << fixed_point( static_cast<double(*)(double)>( std::cos ),
                          0.0, 1e-5, 100 ) << std::endl;
std::cout << fixed_point( static_cast<double(*)(double)>( std::sin ),
                          0.1, 1e-5, 100 ) << std::endl;
```

Some of these throw exceptions, so we may want to try to catch them. We can also pass it a lambda expression:

```
std::cout << fixed_point( [](  double x ){ std::sin(x) + 3.0 },
                          0.0, 1e-10, 100 ) << std::endl;
```

All of this is already implemented at replit.com; however, all of this is meant to introduce to you some of the fascinating and interesting features of C++. Hopefully, this small introduction is useful.

Programming project

In this component of the project, you will have to author a function that uses a templated linear algebra package that allows you to define matrices and vectors and to perform basic linear algebra operations on them. You will then be required to author a function similar to the above fixed-point iteration functions but one that continues to apply a stochastic matrix to an initial vector until that operation converges. All the source code can be found at replit.com, which you can fork or copy.

A *Markov chain* describes a system that may be in one of *n* states, and at each step, the next state that the system goes into depends on the current state the system is in, together with probabilities that one of the states will go into each of one of the *n* states. Such a scenario can be described by an $n \times n$ matrix where:

1. All the entries are greater-than or equal-to zero (0).
2. The sum of each column equals one (1).

An example of a Markov chain, described on Wikipedia, is any board game that depends entirely on the roll of the dice. One game, where there is no competition, is "Snakes and Ladders", a game based on *Moksha Patam*, a game from India. Each move is decide by the throw of a die, with a 1/6 chance of each possible value; however, sometimes a throw of the die my result in an advance beyond the given location (a ladder) and in some cases it is a step back (a snake). There is are 100 locations on the board, and the goal is to get to position 100. Thus, there are 100 states. We will, however, start with a simpler system that has two states.

For example, the matrix $A = \begin{pmatrix} 0.3 & 0.1 \\ 0.7 & 0.9 \end{pmatrix}$ says that:

1. If we are in State 1 (look at Column 1), then there is a 30% chance we will remain in State 1, and a 70% chance we will switch to State 2.
2. If we are in State 2 (look at Column 2), then there is a 10% chance we will switch to State 1, and a 90% chance we will remain in State 2.

Any square matrix of positive entries where the sum of each of the columns is equal to one (1) is said to be a stochastic matrix. Which of the following are stochastic matrices?

$$\begin{pmatrix} 0.3 & 0.7 \\ 0.2 & 0.8 \end{pmatrix}, \begin{pmatrix} 0.4 & 0.2 \\ 0.6 & 0.8 \end{pmatrix}, \begin{pmatrix} 0.3 & 1.2 \\ 0.7 & -0.2 \end{pmatrix}, \begin{pmatrix} 0.129 & 0.026 & 0.131 & 0.101 \\ 0.366 & 0.474 & 0.379 & 0.064 \\ 0.173 & 0.088 & 0.327 & 0.578 \\ 0.332 & 0.412 & 0.163 & 0.257 \end{pmatrix}$$

Mathematically, one way to test if a matrix with only non-negative entries is stochastic is that its transpose should leave a vector of ones (1s) unchanged, so we note that

$$A^{\mathrm{T}}\mathbf{1}_4 = \begin{pmatrix} 0.129 & 0.366 & 0.173 & 0.332 \\ 0.026 & 0.474 & 0.088 & 0.412 \\ 0.131 & 0.379 & 0.327 & 0.163 \\ 0.101 & 0.064 & 0.578 & 0.257 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}.$$

Your assigned project will be to write a function as follows:

```
template <unsigned int n>
vec<n> markov_chain(
    matrix<n, n> A,
    vec<n> v0,
    double eps_step,
    unsigned int max_iterations
);
```

where the matrix and vector classes are provided for you. You can access the entries of the vector using `v0( j )` and the entries of the matrix using `A( i, j )`; however, you can also use operators as per linear algebra, so you can do the following:

```
v1 = A*v0;
```

assuming `v0` and `v1` is of type `vec<n>`.

Both classes defines the dimension in the template, so you must declare variables as follows:

```
matrix<2, 2> A{ {3.2, 4.5}, {2.7, -4.5} };
vec<2> v{ 0.3, 0.8 };
std::cout << A*v << std::endl;
```

Your function will first check if A represents a stochastic matrix, meaning that all entries are non-negative (greater than or equal to zero) and the sum of each of the columns is equal to one. As a sum of floating-point numbers may not equal exactly one due to truncation error, you will test if

$$\left| 1 - \sum_{i=1}^{m} a_{i,j} \right| < m\varepsilon_{\text{step}}$$

for each Column j. If not, throw an invalid argument exception (see the `stdexcept` library). Note that the above assumes linear algebra conventions where indices go from 1 to m and not 0 to $m - 1$ for an $m \times n$ matrix.

Next, if A represents a stochastic matrix, repeatedly assign $\mathbf{v}_{k+1} \leftarrow A\mathbf{v}_k$ and you will continue until $\|\mathbf{v}_{k+1} - \mathbf{v}_k\|_2 < \varepsilon_{\text{step}}$ (in the linear algebra package provided, you will use the `norm(…)` function). You will return the vector $\mathbf{v}_{k+1}$. If you have iterated the maximum number of iterations, you will throw an exception identical to that above. A test is provided in the source code, and you will use this function in the next component of this project.

Hint: Your implementation will be very close to that of the fixed-point iteration function above, only now you will also be checking if the matrix is stochastic.

Investigation component

We will now proceed to using MATLAB. You can use either MATLAB or GNU Octave. The first is available on all engineering computers on campus, and the latter is available for free online. The function `rand(m,n)` produces an $m \times n$ matrix where each entry is a normalized double-precision floating-point number uniformly chosen from the interval (0, 1). Given a matrix A, the function `sum( A )` returns a row vector containing the sum of the columns of A. Try this out using:

```
>> A = rand( 2, 5 )
>> sum( A )
```

Now, in MATLAB, you create a row vector using the notation

```
>> u = [1.2 4.5 2.3 5.1 6.2]
```

or you can also separate the entries with commas:

```
>> u = [1.2, 4.5, 2.3, 5.1, 6.2]
```

To create a column vector, you must either separate the entries by semi-colons, or take the transpose of a row matrix, so either of these produce the same output:

```
>> v = [4.2; 8.2; 1.7; 9.3; 4.5]
>> v = [4.2 8.2 1.7 9.3 4.5]'
```

As you may have guessed, the apostrophe is used to denote the transpose. This is actually quite memorable and almost intuitive.

If you divide a matrix by a scalar, each entry of that matrix is divided by that scalar.

```
>> A = [1 2; 3 4]
    A =
       1   2
       3   4

>> A / 1.25    % Divide each entry of 'A' by 1.25
    ans =
       0.80000   1.60000
       2.40000   3.20000
```

As you may have guessed, `%` is the to-the-end-of-line comment symbol in Matlab like `//` in C++.

If you element-wise divide (using `./`) a matrix by a row vector, each entry of each column is divided by the corresponding entry of the row vector:

```
>> A ./ [2 5]        % Column 1 is divided by 2, Column 2 is divided by 5
    ans =
       0.50000   0.40000
       1.50000   0.80000

>> A ./ [2 5]'       % Row 1 is divided by 2, Row 2 is divided by 5
    ans =
       0.50000   1.00000
       0.60000   0.80000
```

In MATLAB, create a 6 × 6 random square matrix and then convert that matrix into a stochastic matrix by ensuring that the sum of each of the columns is equal to 1 using the tools above.

Given a matrix *A*, you can find the eigenvalues using the `eigs(A)` command. The output depends on what the output is being assigned to. If the function is just being called, or the function is being called and the output is assigned to a variable, then the output will be a column vector.

```
>> A = rand(4, 4)
    A =
       0.379347   0.309201   0.526832   0.984033
       0.246169   0.568130   0.853846   0.944002
       0.410905   0.835912   0.475877   0.858179
       0.464809   0.324757   0.095701   0.491504
>> eigs( A )
    ans =
       2.071784
      -0.441420
       0.378827
      -0.094335
```

By default, if you do not assign the output of any code in MATLAB to a variable, it is automatically assigned to the variable `ans`; however, we can instead choose to assign the output to a variable (which needs not be declared like C++):

```
>> lambda = eigs( A )
    lambda =
       2.071784
      -0.441420
       0.378827
      -0.094335
```

However, if you assign the output to a vector of two variables, the first variable is assigned an orthonormal matrix containing the eigenvectors of *A* as columns, and the second variable is assigned a diagonal matrix containing the eigenvalues on the diagonal. The k^{th} eigenvalue corresponds to the eigenvector in Column *k*.

```
>> [U, D] = eigs( A )
    U =
      -0.462640  -0.547734  -0.420843  -0.662549
      -0.594395  -0.600565   0.616461   0.094234
      -0.588475   0.378023   0.486582  -0.494899
      -0.293866   0.443179  -0.453988   0.554278

    D =
       2.071784         0          0          0
              0  -0.441420          0          0
              0         0   0.378827          0
              0         0          0  -0.094335
```

You can extract the k^{th} eigenvector by asking for all entries (using a colon) in the k^{th} column.

```
>> u3 = U(:,3)     % Extract the 3rd eigenvector
    u3 =
      -0.42084
       0.61646
       0.48658
      -0.45399
>> A*u3            % Multiply the 3rd eigenvector by A
    ans =
      -0.15943
       0.23353
       0.18433
      -0.17198
>> D(3,3)*u3       % Multiply the 3rd eigenvector by the 3rd eigenvalue
    ans =
      -0.15943
       0.23353
       0.18433
      -0.17198
```

Important: In some applications of stochastic matrices, a “stochastic matrix” is actually defined as a matrix where all rows add to 1, and instead of calculate Au, we calculate u^TA, so we multiply A by a row vector on the left (instead of the more customary multiplying a matrix by a column vector on the right). However, for the purposes of this course, we will use the definition of the sum of the columns equaling to one, as this uses the customary matrix-vector multiplication.