Name: Chum Yuthika

SID: 11910127

1. Task 3

Analysis:

Vector arithmetic such as addition, subtraction, scalar multiplication and scalar division are implemented by overloading corresponding operators. On the other hand, Dot product and cross product are implemented by a static function. Precautions are in place to make sure that the operation are valid. Also, only cross product of vector in three dimensions are supported. Also, bear in mind that row vector and column vector are not differentiated.

Matrix arithmetic such as addition, subtraction, scalar multiplication and scalar division, and matrix-vector multiplication are also implemented by overloading operators. Operations such as transposition, conjugation, element-wise multiplication, matrix-matrix multiplication are implemented by a static function. Also, it’s important to bear in mind that these operations don’t directly make changes to the input matrices, instead they return the result of the operation as a new matrix. In addition, it is assumed that vectors in matrix-vector multiplication are of valid type-that is, row or column vectors depending on the order of the multiplication. Moreover, an assumption was made that in conjugation operation, we only need to conjugate the matrix’s elements. As a result, no additional transposition is applied to get the conjugate transpose of a matrix. In addition, note that conjugation operation is only define for complex matrices.

Code:

Vector Arithmetic:

friend Vector<T> operator/ (const Vector<T> &a\_vec, const T& scalar)  
{  
 if (scalar == static\_cast<T>(0.0))  
 throw std::runtime\_error("Math error: Attempted to divide by zero");  
  
 Vector<T> quotient\_vec(a\_vec.getDim());  
   
 for (int i = 0; i < a\_vec.getDim(); i++)  
 quotient\_vec.set(i, a\_vec.get(i) / scalar);  
   
 return quotient\_vec;  
}  
  
friend Vector<T> operator\* (const T &scalar, const Vector<T> &a\_vec)  
{  
 Vector<T> product\_vec(a\_vec.getDim());  
  
 for (int i = 0; i < a\_vec.getDim(); i++)  
 product\_vec.set(i, scalar \* a\_vec.get(i));  
  
 return product\_vec;  
}  
  
friend Vector<T> operator\* (const Vector<T> &a\_vec, const T &scalar)  
{  
 Vector<T> product\_vec(a\_vec.getDim());  
  
 for (int i = 0; i < a\_vec.getDim(); i++)  
 product\_vec.set(i, a\_vec.get(i) \* scalar);  
  
 return product\_vec;  
}  
  
friend Vector<T> operator+ (const Vector<T> &a\_vec, const Vector<T> &b\_vec)  
{  
 if (a\_vec.getDim() != b\_vec.getDim())  
 throw std::invalid\_argument("Vectors are compatible");  
  
 Vector<T> sum\_vec(a\_vec.getDim());  
  
 for (int i = 0; i < a\_vec.getDim(); i++)  
 sum\_vec.set(i, a\_vec.get(i) + b\_vec.get(i));  
  
 return sum\_vec;  
}  
  
friend Vector<T> operator- (const Vector<T> &a\_vec, const Vector<T> &b\_vec)  
{  
 if (a\_vec.getDim() != b\_vec.getDim())  
 throw std::invalid\_argument("Vectors are compatible");  
  
 Vector<T> difference\_vec(a\_vec.getDim());  
  
 for (int i = 0; i < a\_vec.getDim(); i++)  
 difference\_vec.set(i, a\_vec.get(i) - b\_vec.get(i));  
  
 return difference\_vec;  
}  
  
static T Dot\_Product(const Vector<T> &a\_vec, const Vector<T> &b\_vec)  
{  
 // Check that the number of dimensions match.  
 if (a\_vec.getDim() != b\_vec.getDim())  
 throw std::invalid\_argument("Vector dimensions do not match.");  
  
 // Compute the dot product.  
 T dot\_product = static\_cast<T>(0.0);  
  
 for (int i = 0; i < a\_vec.getDim(); i++)  
 dot\_product += (a\_vec.get(i) \* b\_vec.get(i));  
  
 return dot\_product;  
}  
  
static Vector<T> Cross\_Product(const Vector<T>& a\_vec, const Vector<T>& b\_vec)  
{  
 if (a\_vec.getDim() != b\_vec.getDim())  
 throw std::invalid\_argument("Vector dimensions do not match.");  
  
 if (a\_vec.getDim() != 3)  
 throw std::invalid\_argument("Vectors are not three-dimensional");  
  
 std::vector<T> cross\_product;  
 cross\_product.push\_back((a\_vec.get(1) \* b\_vec.get(2)) - (a\_vec.get(2) \* b\_vec.get(1)));  
 cross\_product.push\_back(-((a\_vec.get(0) \* b\_vec.get(2)) - (a\_vec.get(2) \* b\_vec.get(0))));  
 cross\_product.push\_back((a\_vec.get(0) \* b\_vec.get(1)) - (a\_vec.get(1) \* b\_vec.get(0)));  
  
 Vector<T> cross\_product\_vec(cross\_product);  
 return cross\_product\_vec;  
}

Matrix Arithmetic:

//addition  
friend Matrix<T> operator+ (const Matrix<T> &X, const Matrix<T> &Y)  
{  
 HaveSameDim(X, Y);  
 Matrix<T> Sum(X.getRow(), X.getCol());  
  
 for (int i = 0; i < Sum.getRow(); i++)  
 {  
 for (int j = 0; j < Sum.getCol(); j++)  
 {  
 Sum.get(i, j) = X.get(i, j) + Y.get(i, j);  
 }  
 }  
  
 return Sum;  
}  
//subtraction  
friend Matrix<T> operator- (const Matrix<T> &X, const Matrix<T> &Y)  
{  
 HaveSameDim(X, Y);  
 Matrix<T> Difference(X.getRow(), X.getCol());  
  
 for (int i = 0; i < Difference.getRow(); i++)  
 {  
 for (int j = 0; j < Difference.getCol(); j++)  
 {  
 Difference.get(i, j) = X.get(i, j) - Y.get(i, j);  
 }  
 }  
  
 return Difference;  
}  
//scalar multiplication  
friend Matrix<T> operator\* (const Matrix<T> &X, const T &scalar)   
{  
 Matrix<T> Product(X.getRow(), X.getCol());  
  
 for (int i = 0; i < Product.getRow(); i++)  
 {  
 for (int j = 0; j < Product.getCol(); j++)  
 {  
 Product.get(i, j) = X.get(i, j) \* scalar;  
 }  
   
 }  
  
 return Product;  
}  
  
friend Matrix<T> operator\* (const T &scalar, const Matrix<T> &X)   
{  
 Matrix<T> Product(X.getRow(), X.getCol());  
  
 for (int i = 0; i < Product.getRow(); i++)  
 {  
 for (int j = 0; j < Product.getCol(); j++)  
 {  
 Product.get(i, j) = scalar \* X.get(i, j);  
 }  
 }  
  
 return Product;  
}  
  
//scalar division  
friend Matrix<T> operator/ (const Matrix<T> &X, const T &scalar)   
{  
 if (scalar == static\_cast<T>(0.0))  
 throw std::runtime\_error("Math error: Attempted to divide by zero");  
  
 Matrix<T> Quotient(X.getRow(), X.getCol());  
  
 for (int i = 0 ; i < Quotient.getRow(); i++)  
 {  
 for (int j = 0; j < Quotient.getCol(); j++)  
 {  
 Quotient.get(i, j) = X.get(i, j) / scalar;  
 }  
 }  
  
 return Quotient;  
}  
  
//transposition  
static Matrix<T> Transpose(const Matrix<T> &X)   
{  
 Matrix<T> Transposed(X.getCol(), X.getRow());  
  
 for (int i = 0; i < X.getRow(); i++)  
 {  
 for (int j = 0; j < X.getCol(); j++)  
 {  
 Transposed.get(j, i) = X.get(i, j);  
 }  
 }  
  
 return Transposed;  
}  
  
//conjugation / transpose conjugate?  
template <typename U = T, IF(is\_complex<U>)>  
static Matrix<T> Conjugate(const Matrix<T> &X)  
{  
 Matrix<T> Conjugated(X.getRow(), X.getCol());  
  
 for (int i = 0; i < Conjugated.getRow(); i++)  
 {  
 for (int j = 0; j < Conjugated.getCol(); j++)  
 {  
 T &complex = X.get(i, j);  
 Conjugated.get(i, j) = std::conj(complex);  
 }  
 }  
  
 return Conjugated;

}  
//element-wise multiplication  
static Matrix<T> Elementwise\_Multiplication(const Matrix<T> &X, const Matrix<T> &Y)   
{  
 HaveSameDim(X, Y);  
 Matrix<T> Product(X.getRow(), X.getCol());  
  
 for (int i = 0; i < Product.getRow(); i++)  
 {  
 for (int j = 0; j < Product.getCol(); j++)  
 {  
 Product.get(i, j) = X.get(i, j) \* Y.get(i, j);  
 }  
   
 }  
   
 return Product;  
}  
//matrix-matrix multiplication  
friend Matrix<T> operator\* (const Matrix<T> &X, const Matrix<T> &Y)  
{  
 IsCompatible(X, Y);  
 Matrix<T> Product(X.getRow(), Y.getCol());  
  
 for (int i = 0; i < X.getRow(); i++)  
 {  
 for (int j = 0; j < Y.getCol(); j++)  
 {  
 Product.get(i, j) = 0;  
 for (int k = 0; k < X.getCol(); k++)  
 {  
 Product.get(i, j) += X.get(i, k) \* Y.get(k, j);  
 }  
 }  
 }  
  
 return Product;  
}  
  
//matrix-vector multiplication   
friend Vector<T> operator\* (const Matrix<T> &X, const Vector<T> a\_vec)  
{  
 if (X.getCol() != a\_vec.getDim())  
 throw std::invalid\_argument("Matrix and vector are not compatible!");  
  
 Vector<T> product\_vec(X.getRow());  
  
 for (int i = 0; i < X.getRow(); i++)  
 {  
 product\_vec.set(i, 0);  
 for (int j = 0; j < X.getCol(); j++)  
 {  
 product\_vec.set(i, product\_vec.get(i) + X.get(i, j) \* a\_vec.get(j));  
 }  
 }  
  
 return product\_vec;  
}  
  
friend Vector<T> operator\* (const Vector<T> &a\_vec, const Matrix<T> &X)  
{  
 if (a\_vec.getDim() != X.getRow())  
 throw std::invalid\_argument("Vector and matrix are not compatible!");  
  
 Vector<T> product\_vec(X.getCol());  
  
 for (int j = 0; j < X.getCol(); j++)  
 {  
 product\_vec.set(j, 0);  
 for (int i = 0; i < X.getRow(); i++)  
 {  
 product\_vec.set(j, product\_vec.get(j) + a\_vec.get(i) \* X.get(i, j));  
 }  
 }  
  
 return product\_vec;  
}

Result & Verification:

Matrix Arithmetic:

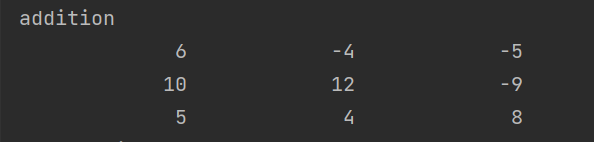
M\_1 = M\_2 =

M\_C =

Scalar\_1 = 9.0 scalar\_2 = 2.0

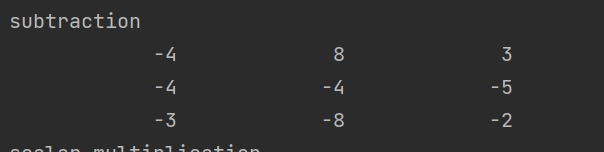
Addition:

M\_1 + M\_2



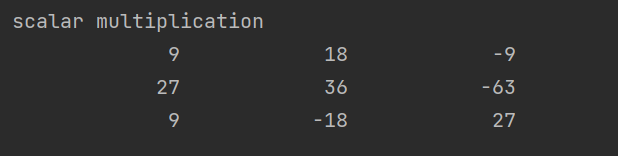
Subtraction:

M\_1 – M\_2

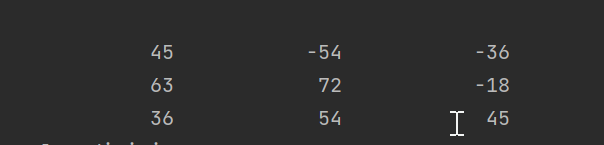


Scalar multiplication

M\_1 \* scalar\_1

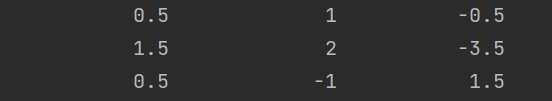


scalar\_1 \* M\_2



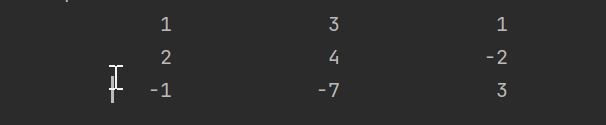
Scalar division

M\_1 / scalar\_2

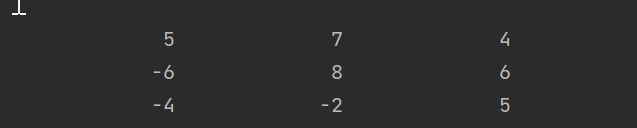


Transposition

Transpose(M\_1)

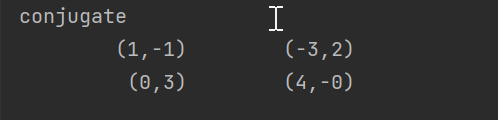


Transpose(M\_2)



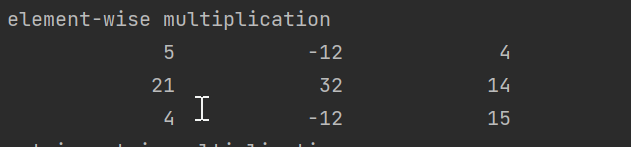
Conjugate

Conjugate(M\_C)



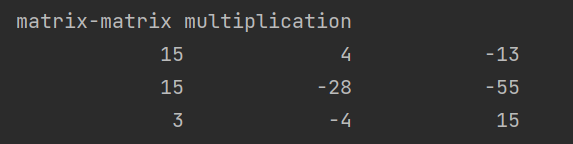
Element-wise multiplication:

Elementwise\_Multplication(M\_1, M\_2)



Matrix-matrix multiplication

M\_1 \* M\_2



Vector Arithmetic:

v\_1\_vec = v\_2\_vec =

Addition:

v\_1\_vec + v\_2\_vec:



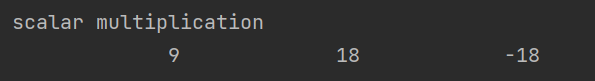
Subtraction:

v\_1\_vec – v\_2\_vec:



Scalar multiplication:

v\_1\_vec \* scalar\_1:

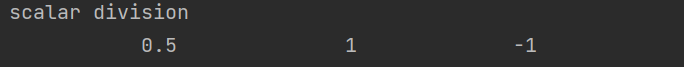


Scalar\_2 \* v\_2\_vec



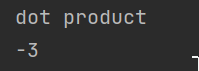
Scalar division:

v\_1\_vec / scalar\_2



Dot Product:

v\_1\_vec dot v\_2\_vec



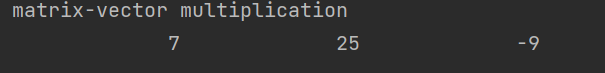
Cross Product:

v\_1\_vec cross v\_2\_vec

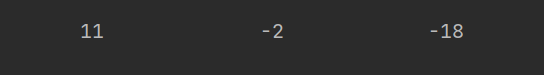


Matrix-vector multiplication:

M\_1 \* v\_1\_vec



v\_1\_vec \* M\_2



Difficulties & Solutions

\_I had some difficulties with conjugate.

5. Task 5

Analysis:

Determinant is computed by basically recursively computing determinant of submatrices; specifically, the cofactors. The general formula can be written as

det(A) = , where Ci, j = (-1)i+j \* det(A(i | j)) . Note that det(A(i | j)) is the determinant of submatrix of A that is got by ignoring the i-th row and the j-th coloumn.

The inverse can be computed by finding the adjoint matrix. The elements of an adjoint matrix can be found by computing all cofactors for the original matrix and transposing it. Finally, we can use adjoint matrix special property, which is

. Thus, we can find the inverse of A by dividing the adjoint of A by its determinant;

The eigenvalues of a matrix can be computed by QR decomposition. Bear in mind that we can only apply this method reliably for symmetric matrices. The goal of QR decomposition is to decompose a matrix A into a product of an Upper-triangular matrix, R and an orthogonal matrix, Q. To find R, we apply the householder method for each column of A which is finding a reflection matrix Pi such that Pi\*Pi-1\*…A transform the i-th column of A to become column of an upper triangular. The formular for computing P is where is the normalized form of . can be computed by where is the columns vector and is the vector which we want to reflect upon. After reiterating QR\_Decompose() and transforming A by A = Q \* R, we can get the eigenvalues of A by looking at the diagonal elements of resultant matrix.

After computing the eigenvalues of a matrix, we can compute the corresponding eigenvector by inverse power iteration method. The basic of the method is iteratively applying a formula to a vector, , where is the eigenvalue and A is our original matrix. While we can do it by power iteration method, inverse power iteration method can converge faster.

Code:

//determinant  
static T Determinant(const Matrix<T> &X)  
{  
 if(!IsSquare(X))  
 throw std::invalid\_argument("The matrix is not square! Non-square matrices do not have determinant!");  
  
 T determinant = 0;  
 int n = X.getCol();  
  
 if (n == 1)  
 return X.get(0, 0);  
 else  
 {  
 for (int j = 0; j < n ; j++)  
 {  
 determinant += (((0+j) % 2 == 0) ? 1 : -1) \* X.get(0, j) \* Determinant(SubMatrix(X, 0, j));  
 }  
 }  
  
 return determinant;  
}  
//find cofactor  
static Matrix<T> SubMatrix(const Matrix<T> &X, int row, int col)  
{  
 //ignore i = row and j = col  
 int m = X.getRow();  
 int n = X.getCol();  
 Matrix<T> SubMatrix(m-1, n-1);  
  
 for (int i = 0; i < m; i++)  
 {  
 for (int j = 0; j < n; j++)  
 {  
 if (i != row && j != col)  
 {  
 if (j < col && i < row)  
 SubMatrix.get(i, j) = X.get(i, j);  
 else if (j < col && i > row)  
 SubMatrix.get(i-1, j) = X.get(i, j);  
 else if (j > col && i < row)  
 SubMatrix.get(i, j-1) = X.get(i, j);  
 else if (j > col && i > row)  
 SubMatrix.get(i-1, j-1) = X.get(i, j);  
  
 }  
 }  
 }  
  
 return SubMatrix;  
}  
  
//get adjoint  
static Matrix<T> Adjoint(const Matrix<T> &X)  
{  
 if (!IsSquare(X))  
 throw std::invalid\_argument("The matrix is not square! Non-square matrices do not have an adjoint matrix!");  
  
 if (IsZero(X))  
 throw std::invalid\_argument("The matrix is a Zero matrix! Zero matrices do not have an adjoint matrix");  
  
 int n = X.getCol();  
 Matrix<T> Adj(n, n);  
  
 if (n == 1)  
 {  
 Adj.get(0, 0) = static\_cast<T>(1.0);  
 return Adj;  
 }  
  
 for (int i = 0; i < n; i++)  
 for (int j = 0; j < n; j ++)  
 Adj.get(j, i) = (((i + j) % 2 == 0) ? 1 : -1) \* Determinant(SubMatrix(X, i, j));  
  
 return Adj;  
  
}  
//find inverse  
static Matrix<T> Inverse(const Matrix<T> &X)  
{  
 if (!IsSquare(X))  
 throw std::invalid\_argument("The matrix is not square! Non-square matrices do not have an inverse!");  
  
 int n = X.getCol();  
 Matrix<T> Inv(n, n);  
 Matrix<T> Adj(n, n);  
 T det = Determinant(X);  
  
 if (det == static\_cast<T>(0.0))  
 {  
 throw new std::invalid\_argument("The matrix is singular! Singular matrices do not have an inverse!");  
 }  
  
 Adj = Adjoint(X);  
  
 Inv = Adj / det;  
  
 return Inv;  
  
}  
  
static T Trace(const Matrix<T> &X)  
{  
 if (!IsSquare(X))  
 throw std::invalid\_argument("The matrix is not square! Unable to compute trace!");  
  
 int n = X.getCol();  
 T trace = static\_cast<T>(0);  
 for (int i = 0; i < n; i++)  
 trace += X.get(i, i);  
  
 return trace;  
}

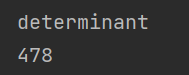
static void QR\_Decompose(const Matrix<T> &A, Matrix<T> &Q, Matrix<T> &R)  
{  
 Matrix<T> A\_Copied = A;  
  
 if (!IsSquare(A\_Copied))  
 throw std::invalid\_argument("The matrix is not square! Unable to perform QR decomposition!");  
  
 int n = A\_Copied.getCol();  
  
 std::vector<Matrix<T>> Ps;  
 for (int j = 0; j < n - 1; j++)  
 {  
 Vector<T> a\_vec (n - j);  
 Vector<T> b\_vec (n - j);  
  
 for (int i = j; i < n; i++)  
 {  
 a\_vec.set(i-j, A\_Copied.get(i, j));  
 b\_vec.set(i-j, static\_cast<T>(0.0));  
 }  
 b\_vec.set(0, static\_cast<T>(1.0));  
  
 //length of a-vector  
 T a\_norm = Vector<T>::Norm(a\_vec);  
  
 //sign  
 int sign = -1;  
 if (a\_vec.get(0) < static\_cast<T>(0.0))  
 sign = 1;  
  
 //compute n-vector  
 Vector<T> n\_vec = Vector<T>::Normalize(a\_vec - (sign \* a\_norm \* b\_vec));  
  
 //convert n-vector to matrix to transpose  
 Matrix<T> N\_Mat (n - j, 1);  
 for (int i = 0; i < n - j; i++)  
 N\_Mat.get(i, 0) = n\_vec.get(i);  
  
 //transpose n\_mat  
 Matrix<T> N\_Mat\_T = Transpose(N\_Mat);  
  
 //create an identity matrix  
 Matrix<T> I (n - j, n - j);  
 SetToIdentity(I);  
  
 //Compute P\_Temp  
 Matrix<T> P\_Temp = I - static\_cast<T>(2.0) \* N\_Mat \* N\_Mat\_T;  
  
 //form the P matrix with original dimensions  
 Matrix<T> P (n, n);  
 SetToIdentity(P);  
  
 for (int row = j; row < n; row++)  
 for (int col = j; col < n; col++)  
 P.get(row, col) = P\_Temp.get(row - j, col - j);  
  
 //store result to Ps  
 Ps.push\_back(P);  
  
 //Apply transformation to inputMatrix  
 A\_Copied = P \* A\_Copied;  
 }  
  
 //compute Q  
 Q = Ps.at(0);  
  
 for (int i = 1; i < n - 1; i++)  
 Q = Q \* Transpose(Ps.at(i));  
  
 //compute R  
 int p\_num = Ps.size();  
 R = Ps.at(p\_num - 1);  
  
 for (int i = p\_num - 2; i >= 0; i--)  
 {  
 R = R \* Ps.at(i);  
 }  
  
 R = R \* A;  
}  
  
static void Eigenvalues(const Matrix<T> &A, std::vector<T> &eigenvalues)  
{  
 //male Copy of A  
 Matrix<T> A\_Copied = A;  
  
 //verify A is square  
 if (!IsSquare(A\_Copied))  
 throw std::invalid\_argument("The matrix is not square! Unable to compute eigenvalues!");  
  
 //verify A is symmetric  
 if (!IsSymmetric(A\_Copied))  
 throw std::invalid\_argument("Unable to compute eigenvalues for non-symmetric matrices");  
  
 int n = A\_Copied.getCol();  
  
 //create an identity matrix  
 Matrix<T> I (n, n);  
 SetToIdentity(I);  
  
 //create matrices to store Q and R  
 Matrix<T> Q (n, n);  
 Matrix<T> R (n, n);  
  
 int max\_iteration = 10e3;  
 int iteration\_cnt = 0;  
 while (iteration\_cnt < max\_iteration)  
 {  
 QR\_Decompose(A\_Copied, Q, R);  
  
 A\_Copied = R \* Q;  
  
 //check if A is close enough to an upper-triangular  
 if (IsCloseToUEnough(A\_Copied))  
 break;  
  
 iteration\_cnt++;  
 }  
  
 //eigenvalues is the diagonal elements of A  
 for (int i = 0; i < n; i++)  
 {  
 eigenvalues.push\_back(A\_Copied.get(i, i));  
 }  
}  
  
//find eigenvector by inverse power iteration method  
static void Eigenvectors(const Matrix<T> &A, const T &eigenvalue, Vector<T> &eigenvector)  
{  
 //verify A is square  
 IsSquare(A);  
  
 std::random\_device myRandomDevice;  
 std::mt19937 myRandomGenerator(myRandomDevice());  
 std::uniform\_int\_distribution<int> myDistribution(1.0, 10.0);  
  
 int n = A.getCol();  
  
 Matrix<T> I(n, n);  
 SetToIdentity(I);  
  
 Vector<T> v\_vec(n);  
 for (int i = 0; i < n; i++)  
 v\_vec.set(i, static\_cast<T>(myDistribution(myRandomGenerator)));  
  
 int max\_iteration = 100;  
 int iteration\_cnt = 0;  
 T min\_epsilon = static\_cast<T>(1e-9);  
 T epsilon = static\_cast<T>(1e6);  
 Vector<T> prev\_vec(n);  
  
 while ((iteration\_cnt < max\_iteration) && (epsilon > min\_epsilon))  
 {  
 prev\_vec = v\_vec;  
  
 v\_vec = Vector<T>::Normalize(Inverse(A - (eigenvalue \* I)) \* v\_vec);  
  
 epsilon = Vector<T>::Norm((v\_vec - prev\_vec));  
  
 iteration\_cnt++;  
 }  
  
 eigenvector = v\_vec;  
}

Result & Verification:

M\_Sym =

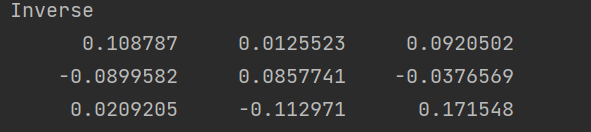
\_Determinant:

Determinant(M\_2)



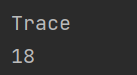
\_Inverse:

Inverse(M\_2)



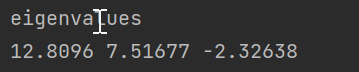
\_Trace:

Trace(M\_2)



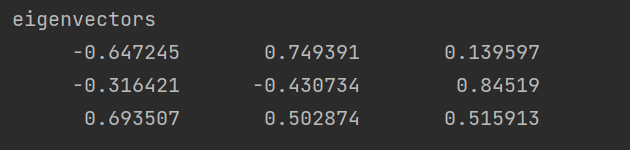
\_Eigenvalues:

Eigenvalues(M\_Sym, eigenvalues)



\_Eigenvectors:

Eigenvector(M\_Sym, eigenvalues.at(i), eigenvector\_vec)



Note that the three rows correspond to eigenvectors for each eigenvalues

Difficulties & Solutions:

I had difficulties with finding eigenvalues and eigenvectors.