Program construction in C++ for Scientific Computing - Project 2

Pia Callmer, Jonas Fey

October 2022

Task 1 - Evaluation of the exponential for real numbers

The first task is to implement the exponential function for real numbers with the series representation

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

For computationally efficiency, we calculated each summand n with help of the previous summand $(n-1)$, by multiplying it with $\frac{x}{n}$. The calculation of the series stops if the absolute difference between the approximation e_n^x and e_{n-1}^x is below the tolerance. This is the absolute value of the last computed summand n.
We compare our results against the exponential function of the standard library for multiple x values and a tolerance of `tol` $= 10^{-10}$ and some results are displayed in Table 1.

x	error	N iterations
1	$8.15 \cdot 10^{-13}$	15
5.3	$4.15 \cdot 10^{-12}$	31
-6.2	$1.37 \cdot 10^{-11}$	33
156.3	$4.79 \cdot 10^{52}$	445
13523.7	`nan`	125
-47.3	-2255.77	158

Table 1: Error and number of iterations used for approximating the scalar exponential function

As one can see in the table, the errors between the implemented and the given exponential function for values which are close to 0 are very small. However, going into the positive direction, the value e^b growths exponentially for large b. Comparing the values for $x = 156.3$ one can see that the error $4.79 \cdot^{52}$ is large. The actual value, however, is $e^{156.3} = 7.58975 \cdot 10^{67}$. Therefore, the values are still very close to another, as there are 15 digits which are identical, so an error at 10^{52} does not change the value with a magnitude of 10^{67} very much. The value for $x = 13523.7$ is `nan` as both functions return `inf` for this large value. Finally, for very small values, the series experiences a problem. Looking at the value for $x = -47.3$ the error is very large, especially considering we are looking for $e^{-47.3}$ which is a very small value between 0 and 1. The problem is that the series evaluates the terms $\frac{x^n}{n!}$ in every iteration n. For small n this is a problem, as the exponential term will be much bigger than the factorial term. If n is large enough, the factorial term will outweigh the exponential and values < 1 will be added. However, until this is the case, large values $>> 1$ will be added to the estimated solution. For large values the accuracy for these values will lie in the front values, just as observed in the accuracy for $x = 156.3$. This means, that once the smaller values will be added, the accuracy of the large values will have distorted the total accuray, resulting in the unexpected large value.

Task 2 - Evaluation of the exponential for matrices

The second task is to approximate the exponential of a square matrix $\mathbf{A} \in \mathbb{R}^{m \times m}$ by

$$\exp(\mathbf{A}) = \sum_{n=0}^{\infty} \frac{\mathbf{A}^n}{n!}.$$

with a specified tolerance. For the stopping criterion we make use of the maximum norm for matrices, which is

$$||\mathbf{A}||_{\max} := \max_{i=1..m, j=1,..n} |a_{ij}|.$$

The error is defined as the maximum norm of the element-wise difference between the exponential approximations between two following iterations, which is the same as the maximum norm of the last summand. For comparison the `r8mat_exp.cpp` file is used, which implements MATLAB's algorithm for the matrix exponential.

The class `Matrix` was implemented for this, which can be seen in Appendix A.2 and A.3. The operators `=`, `+` and `*` were overloaded to work with matrices and to simplifiy the implementation of the exponential. To evaluate the accuracy, error matrices are created which show the differences for each value between the implemented exponential matrix and the exponential matrix calculated by `r8mat_exp.cpp`. The error matrices are shown in the output when running the function. The observed results are similar to the ones in Task 1. Matrices with all values close to 0 are very similar. This can be seen in the example shown in Equation 1. Here, the accuracy is, as expected, in the range of 10^{-10}. Matrices with even a single large value (> 30) will cause large errors but also large values. Therefore their accuracy can be still acceptable. And finally, large negative values encounter the same accuracy problem as described in Task 1 and will therefore result in large errors. This is shown in Equation 2. Here, the large magnitude of the negative values, overshadows the factorial for small n and causes large values in the result of the series approximation. Therefore, the error explodes quickly and is so high for values which should be between 0 and 1.

$$\mathbf{A}_1 = \begin{bmatrix} 1.3 & -4.2 & 9.81 \\ 2.4 & 4.78 & 0 \\ 6.7 & 12 & 1 \end{bmatrix}, \qquad \exp(\mathbf{A}_1) = \begin{bmatrix} 21091.5 & 26325.1 & 20856.9 \\ 8243 & 10291.6 & 8150.06 \\ 24214.3 & 30227.8 & 23943 \end{bmatrix} \tag{1}$$

$$\mathbf{A}_{1,\text{error}} = \begin{bmatrix} -1.09139 \cdot 10^{-10} & -1.74623 \cdot 10^{-10} & -1.09139 \cdot 10^{-10} \\ -5.82077 \cdot 10^{-11} & -8.54925 \cdot 10^{-11} & -5.54792 \cdot 10^{-11} \\ -1.52795 \cdot 10^{-10} & -2.32831 \cdot 10^{-10} & -1.38243 \cdot 10^{-10} \end{bmatrix}$$

$$\mathbf{A}_2 = \begin{bmatrix} -19.44 & -5.37 & -4.29 & -21.39 \\ -24.39 & -15.38 & -15.41 & -21.58 \\ -21.26 & -11.18 & -23.33 & -2.04 \\ -13.23 & -20.82 & -11.15 & -24.3 \end{bmatrix} \tag{2}$$

$$\mathbf{A}_{2,\text{error}} = \begin{bmatrix} -3.11564 \cdot 10^{9} & -9.10903 \cdot 10^{9} & -7.38592 \cdot 10^{9} & 1.4478 \cdot 10^{10} \\ -1.2618 \cdot 10^{10} & -8.57881 \cdot 10^{9} & -9.78819 \cdot 10^{9} & 1.54037 \cdot 10^{10} \\ -7.13885 \cdot 10^{9} & 6.16003 \cdot 10^{9} & -8.46352 \cdot 10^{8} & -6.71691 \cdot 10^{9} \\ 2.92466 \cdot 10^{9} & -2.65621 \cdot 10^{9} & -1.17808 \cdot 10^{9} & -8.35705 \cdot 10^{9} \end{bmatrix}$$

A Appendix – Code files

A.1 main.cpp

```
#include <iostream>
#include <cmath>

#include "matrix.hpp"
using namespace std;
#include "r8lib.h"
#include "r8mat_expm1.h"


// Approximate exponential of real number
double myexp(double x, double tol=1e-10){

    double exp_n = 1; // set first component to n=0
    int n = 1; // This is used for displaying number of iterations used
       ↪ for approximation
    double err = tol*2; // Set error larger than tolerance to ensure at
       ↪ least one summand of the approximation

    // Store previous components to avoid computational expense
    double prev_val = exp_n;

    // Iterate until tolerance is reached
    while (err > tol) {

        // Calculate next summand n by using previous summand (n-1)
        prev_val = (prev_val*x)/n;

        // Add new component to approximation series
        exp_n += prev_val;

        if (isinf(exp_n)){
            break;
        }

        // Measure error by measuring absolute difference between the
           ↪ approximated exponentials between iterations
        // this is the absolute value of the last summand added to the
           ↪ series
        err = abs(prev_val);
        n +=1 ;
    }
    cout << "number␣of␣iterations␣used:␣" << n << "␣with␣tolerance:␣" <<
       ↪ tol<<endl;
    return exp_n;
}


// Compute matrix exponential with series up to a given tolerance
Matrix matexp(Matrix& M, double tol=1e-10){
```

```
    // First exponential component is the identity matrix
    Matrix exp_n = Matrix(M.Cols());
    exp_n.fillNumber(0);
    exp_n.fillDiagonal(1);
    double n = 1.0;
    double err = tol*2;


    // Store previous components to avoid computational expense
    Matrix prev_val = exp_n;

     // Iterate until tolerance is reached
    while (err > tol) {

        // Compute next summand n with previous summand n-1
        prev_val *= M;
        prev_val *= 1.0/n;

        // Update series by new summand
        exp_n += prev_val;

        // The error is here defined as the maximal difference between
           ↪ elements of new and old calculated matrix exponential
           ↪ approximations
        // this is the maximal value of the last summand added to the
           ↪ series
        err = prev_val.maxnorm();

        if (isinf(err))
            break;

        n +=1.0 ;
    }
    cout << "number␣of␣iterations␣used:␣" << n<< endl;
    return exp_n;

}


int main(){

    cout << "Executing␣Task␣1" << endl << endl;
    double myval; double testval;
    double testvalues[] = {1.0,5.3,-6.2,156.3,13523.74, -47.3};
    for(auto& e : testvalues){
        cout << "----------------------" << endl << "Evaluating␣e^" << e <<
           ↪  endl;
        myval = myexp(e);
        testval = exp(e);
        cout << "res:␣" << myval << endl;
        cout << "exp␣function:␣" << testval << endl;
        cout << "Difference:␣" << abs(myval - testval) << endl << "
           ↪ ----------------------" << endl << endl;
    }
```

```
cout << "Executing␣Task␣2"<< endl;


cout << endl << endl << endl << "----------------------" << endl <<
    ↪ endl << endl;

// Building Test Matrices

// Taking exemplary values
double v1[9] = {1.3,2.4,6.7,-4.2,4.78,12,9.81,0.0,1.0};

// Matrix with bad condition number and single large value
double v2[4] = {37, -0.002, 3.4, 0.0345};

// Large Matrix 7x7 with random big numbers (up to 100)
double v3[49];
for (int i=0;i<49;i++){
     v3[i] = rand() % 100;
}

// Matrix with random decimal numbers between 0 and -50
double v4[25];
for (int i=0;i<25;i++){
     v4[i] = -((double)(rand() % 5000)) / 100;
}

// Matrix with random numbers between -5 and 5 with decimals
double v5[16];
for (int i=0;i<16;i++){
     v5[i] = (((double)(rand() % 1000)) / 100) - 5;
}

// Matrix with random numbers between -25 and 0 with decimals
double v6[16];
for (int i=0;i<16;i++){
     v6[i] = -(((double)(rand() % 2500)) / 100);
}

// Creating and filling matrices
Matrix T1 = Matrix(3, v1, (sizeof(v1) / sizeof(*v1)));
Matrix T2 = Matrix(2, v2, (sizeof(v2) / sizeof(*v2)));
Matrix T3 = Matrix(7, v3, (sizeof(v3) / sizeof(*v3)));
Matrix T4 = Matrix(5, v4, (sizeof(v4) / sizeof(*v4)));
Matrix T5 = Matrix(4, v5, (sizeof(v5) / sizeof(*v5)));
Matrix T6 = Matrix(4, v6, (sizeof(v6) / sizeof(*v6)));

Matrix matrices[6]= {T1,T2,T3,T4,T5,T6};
double* vectors[6] = {v1,v2,v3,v4,v5,v6};
int sizes[6] = {9,4,49,25,16,16};
```

```
    for (int i = 0; i < (sizeof(matrices) / sizeof(*matrices)); i++){

        cout << endl << "------------------------" << endl;
        cout << endl << "Given matrix: " << endl;

        Matrix M = matrices[i];
        double* v = vectors[i];
        M.printMatrix();

        // Calculate exponential with our function
        Matrix expM = matexp(M);
        cout << endl << "Exponential of given matrix: " << endl;
        expM.printMatrix();

        // Checking with given function
        double* checkres = r8mat_expm1(M.Cols(), vectors[i]);
        Matrix checkmat = Matrix(M.Cols());
        checkmat.fillMatrix(checkres, sizes[i]);
        cout << endl << "Given result: " << endl;
        checkmat.printMatrix();

        // Calculating the errors
        checkmat *= -1;
        cout << endl << "Error Matrix: " << endl;
        expM += checkmat;
        expM.printMatrix();

        cout << "------------------------" << endl;

    }

 return 0;
}
```

A.2 matrix.hpp

```
#ifndef MATRIX_HPP
#define MATRIX_HPP


// Header file for Matrix object
class Matrix {
private:
    int rows;
    int cols;
    double **matx;
public:
    //Constructors
    Matrix(int m); // Creates mxm Matrix with all entries 0
    Matrix(int m, double* a, int arraySize); // Creates mxm Matrix with
       ↪ values given by vector a with size arraySize
    Matrix(const Matrix&); // Copy constructor

    // Destructor
    ~Matrix();

    //Operation overloading
    Matrix& operator=(const Matrix&);
    Matrix& operator+=(const Matrix&);
    Matrix& operator*=(const Matrix&);
    Matrix& operator*=(const double);

    //Getter (member) functions
    int Rows() const {return rows;};
    int Cols() const {return cols;};
    double** Matx() const {return matx;};

    //Member functions
    double p2norm() const;
    double maxnorm() const;
    void printMatrix() const;
    void fillMatrix(double *, int);
    void fillNumber(double );
    void fillDiagonal(double);
};
#endif
```

A.3 matrix.cpp

```
#include <iostream>
#include <iostream>
#include <cmath>

#include "matrix.hpp"
using namespace std;


//Constructor for a square matrix with dimensions m x m
Matrix::Matrix(int m){
    rows = m;
    cols = m;

    // Allocate new Memory
    matx = new double*[rows];
    for(int i = 0; i < rows; i++){
        matx[i] = new double[cols];
        for(int j = 0 ; j < cols ; j++){
            matx[i][j] = 0; // Assume 0 Matrix as initial value
        }
    }

}

// Constructor for a square matrix with dimensions m x m and values given
   ↪ by vector a
Matrix::Matrix(int m, double* a, int arraySize){
    if(m*m == arraySize){
        rows = m;
        cols = m;

        // Fill Matrix with a
        matx = new double*[rows];
        for(int i = 0; i < rows; i++){
            matx[i] = new double[cols];
            for(int j = 0 ; j < cols ; j++){
                matx[i][j]= a[j*cols+i];
            }
        }
    } else cout << "ERROR␣Matrix␣Constructor:␣Dimension␣of␣a␣does␣not␣fit␣
       ↪ into␣Matrix␣with␣size:␣" << m << "x" << m << endl;
}


// Copy constructor, construct similar matrix with same attributes as
   ↪ given matrix M
Matrix::Matrix(const Matrix& M){

    //take over the attributes
    rows = M.Rows();
    cols = M.Cols();
```

```
    //build new matrix
    matx = new double*[rows];

    // loop over each element
    for (int i = 0 ; i< rows ; i++){
        // allocate the memory
        matx[i] = new double[cols];
        for(int j = 0 ; j < cols ; j++)
            matx[i][j] = M.Matx()[i][j]; // Copy values
    }
}


// Destructor for the matrix
Matrix::~Matrix() {
    for (int i = 0; i < rows; i++) {
        delete[] matx[i];
    }
        delete[] matx;
}


//Overrule copy operator
Matrix& Matrix::operator=(const Matrix& M){
    if (this != &M){
        // Create new Matrix with the number of rows and coloums being the
           ↪  same as M
        rows = M.Rows();
        cols = M.Cols();
        matx = new double*[rows];

        // Filling the values
        for (int i = 0 ; i< rows ; i++){
            matx[i] = new double[cols];

            for(int j = 0 ; j < cols ; j++)
                // copy values
                matx[i][j] = M.Matx()[i][j];
            }
        }
    return *this; // dereferencing
}

//Overload operator +=
Matrix& Matrix::operator+=(const Matrix& M ){
    if (rows == M.Rows() && cols == M.Cols()){

         // loop over each element
        for (int i = 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++)
                matx[i][j] += M.Matx()[i][j]; // addition element-wise
        }
    } else
        cout << "␣ERROR␣+=␣:␣The␣matrix␣shapes␣do␣not␣match␣for␣addition!␣
```

```
            ↪ Left␣Matrix␣" << rows  << "␣x␣" << cols << "␣Right␣matrix␣"
            ↪ << M.Rows() <<"␣x␣"<< M.Cols() << endl;
    return *this;

}

//Overload operator *= for Matrix multiplications
Matrix& Matrix::operator*=(const Matrix& M ){
    if (cols == M.Rows()){

        double ** lmatx = new double*[rows];

        // Copy Matrix
        lmatx = matx;

        // Create resulting matrix of size: rows x M.Cols()
        matx = new double*[rows];
        for(int i = 0 ; i < M.Cols() ; i++){
            matx[i] = new double[M.Cols()];
        }

         // loop over each element
        for (int i= 0; i< rows; i++ ){
            for (int j = 0 ; j< M.Cols() ; j++){
                double sum = 0;

                //get the linear combination to compute element i,j
                for (int k=0; k< rows; k++)
                    sum += lmatx[i][k]*M.Matx()[k][j];
                matx[i][j] = sum;
            }
        }
        //Set the new amount of columns (rows stay the same)
        cols = M.Cols();
    }
    else
        cout << "␣ERROR␣*=:␣The␣matrix␣shapes␣do␣not␣match␣for␣
            ↪ multiplication!␣Left␣Matrix␣" << rows  << "␣x␣" << cols << "
            ↪ ␣Right␣matrix␣" << M.Rows() <<"␣x␣"<< M.Cols() << endl;
    return *this;

}

// Overload operator *= To multiply matrix with scalar value
Matrix& Matrix::operator*=(const double a ){

         // loop over each element
        for (int i = 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++)
                matx[i][j] *= a; // multiply element-wise
        }
    return *this;
}
```

```
// Member function to print matrix:
void Matrix::printMatrix() const{
  for (int i = 0 ; i< rows ; i++){
    cout << "|";
    for(int j = 0 ; j < cols-1 ; j++)
        cout << matx[i][j] << ", " ;
    cout << matx[i][cols-1] << "|" << endl ;
    }
}

// Implementing the 2 norm, i.e. sum over all elements, take the square
   ↪ element-wise and take the root of the sum
double Matrix::p2norm() const{
    double res = 0;
     for (int i = 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++)
                // Add square of each element
                res += matx[i][j]*matx[i][j];
    }
    // Take the root
    res = sqrt(res);
    return res;
}

//Maximum norm returns the absolute largest value of the matrix
double Matrix::maxnorm() const{
    // Initialize max value
    double current_max = -1.0;
    for (int i = 0 ; i< rows ; i++){
        for(int j = 0 ; j < cols ; j++){
            // Update if absolute value is larger
            if (abs(matx[i][j]) > current_max){
                current_max = abs(matx[i][j]);
            }
        }
    }
    return current_max;
}


// Function which fills all elements with the vector a of length n*m
   ↪ column-wise
void Matrix::fillMatrix(double *a, int arraySize){

    // Check if dimension align
    if ( rows*cols == arraySize ) {
        for (int i= 0 ; i< rows ; i++){
            for(int j = 0 ; j < cols ; j++)
                // Fill value with value given by the vector
                matx[i][j]= a[j*cols+i];
        }
    } else
```

```
        cout << "ERROR␣FILLMATRIX:␣The␣length␣of␣the␣array␣does␣not␣match␣the
            ↪ ␣matrix␣shape:␣" << rows << "␣x␣" << cols << "␣Got␣shape:␣" <<
            ↪ a << "␣Expected:␣" << rows*cols  << endl;
}


// Function which fills all elements of the matrix with the same double a
void Matrix::fillNumber(double a){
    for (int i= 0 ; i< rows ; i++){
        for(int j = 0 ; j < cols ; j++)
            matx[i][j]= a;
    }
}

// Fill diagonal of matrix with value a
void Matrix::fillDiagonal(double a){
    int smallersize;
    // if non-square matrix find min(rows, cols)
    if(rows < cols){
        smallersize = rows;
    }
    else {
        smallersize = cols;
    }

    for (int i= 0 ; i< smallersize ; i++){
        matx[i][i]= a; // fill diagonal
    }
}
```