Códigos

- main.js

```
 const { Matrix, inverse } = require('ml-matrix');
const io = require('console-read-write');
const fs = require('fs');
const util = require('util');
const CholeskyDecomposition = require('./methods/CholeskyDecomposition');
const LUDecomposition = require('./methods/LUDecomposition');
const Jacobi = require('./methods/Jacobi');
const GaussSeidel = require('./methods/GaussSeidel');
const createOutputFile = require('./utils/createOutputFile');

util.inspect.defaultOptions.depth = null;

async function main() {
  console.log('Digite o nome do arquivo:(ele deve estar na mesma pasta do executavel)');
  const fileName = await io.read();
  // const fileName = 'matrixGS.txt';

  let buffer;
  try {
    buffer = fs.readFileSync(`./${fileName}`);
  } catch (error) {
    console.log('Arquivo não encontrado!');
    return;
  }

  const bufferAsString = buffer.toString();

  let [
    n,
    ICOD,
    IDET,
    ...rest
  ] = bufferAsString.split('\n');

  const shouldCalculateDeterminant = !parseInt(IDET);
  ICOD = parseInt(ICOD);
  n = parseInt(n);

  const matrix_elements = rest.slice(0, n);
  const vector_elements = rest[n];
  const TOLm = rest[n + 1] || -1;

  const matrixA = matrix_elements.map((line) => line.split(' ').map((a) => parseInt(a)));
  const vectorB = vector_elements.split(' ').map((a) => [parseInt(a)]);

  try {
    const matrix = new Matrix(matrixA);
    inverse(matrix);
  } catch (error) { console.log({ error });
    console.log('Essa matriz é singular!');
    return;
  }

  const params = {
    n,
    matrixA,
    vectorB,
    shouldCalculateDeterminant,
    tol: Math.pow(10, TOLm),
  };
```

```
  console.log({ params });

  let answer;

  switch(ICOD) {
    case 1:
      answer = LUDecomposition(params);
      break;
    case 2:
      answer = CholeskyDecomposition(params);
      break;
    case 3:
      answer = Jacobi(params);
      break;
    case 4:
      answer = GaussSeidel(params);
      break;
    default:
      break;
  }

  createOutputFile(answer, ICOD);
}

main().then(response => console.log(response));
```

- utils/createEmptyMatrix.js

```
function createMatrix(n,m) {
  const matrix = [];
  for (let i = 0; i < n; i++) {
    matrix.push([]);
    for (let j = 0; j < m; j++) {
      matrix[i].push(0);
    }
  }
  return matrix;
}

module.exports = createMatrix;
```

- utils/createOutputFile.js

```
const fs = require('fs');

function createOutputFile(answer, ICOD) {
  let fileString = '';

  switch (ICOD) {
    case 1:
    case 2: {
      const {
        vectorX,
        determinant,
      } = answer || {};

      fileString += 'Solução X:\n';
      fileString += `${vectorX.map(([a]) => a).join(' ')}\n`;
      if (determinant) {
        fileString += `Determinante: ${determinant}\n`;
      }
      break;
    }
    case 3:
    case 4: {
      const {
        vectorX,
        determinant,
        iterations,
        errors,
        residues,
      } = answer || {};

      fileString += 'Solução X:\n';
      fileString += `${vectorX.map(([a]) => a).join(' ')}\n`;
      if (errors && errors.length) {
        fileString += 'Erros:\n';
        errors.forEach((error) => {
          fileString += `${error}\n`;
        });
      }
      if (determinant) {
        fileString += `Determinante: ${determinant}\n`;
      }
      fileString += `Iterações: ${iterations}\n`;
      fileString += 'Histórico de variação do erro:\n';
      residues.forEach((residue) => {
        fileString += `${residue}\n`;
      });
      break;
    }
    default:
      fileString += `ICOD ${ICOD} inválido!\n`;
      break;
  }
  console.log({ file: fileString });
  fs.writeFileSync('answer.txt', fileString);
}

module.exports = createOutputFile;
```

- utils/generateY.js

```
const createEmptyMatrix = require('./createEmptyMatrix');

function generateY({ n, matrix, vectorB }) {
  const vectorY = createEmptyMatrix(vectorB.length , 1);

  for (let i = 0; i < n; i++) {
    let sum = 0;
    for (let j = 0; j <= i; j++) {
      if (j === i) {
        vectorY[i][0] = (vectorB[i][0] - sum)/matrix[i][j];
        sum = 0;
      } else {
        sum += matrix[i][j]*vectorY[j][0];
      }
    }
  }

  return vectorY;
}

module.exports = generateY;
```

- utils/getResidue.js

```
function getNorm(newVectorX,oldVectorX) {
  let norm = 0;
  for (let i = 0; i < newVectorX.length; i++) {
    if (oldVectorX) norm += (newVectorX[i][0] - oldVectorX[i][0])*(newVectorX[i][0] - oldVectorX[i][0]);
    else norm += newVectorX[i][0]*newVectorX[i][0];
  }

  return Math.sqrt(norm);
}

function getResidue(newVectorX, oldVectorX) {
  return getNorm(newVectorX,oldVectorX)/getNorm(newVectorX);
}

module.exports = getResidue;
```

- utils/getXfromY.js

```
const createEmptyMatrix = require('./createEmptyMatrix');

function getXFromY({ vectorY, matrix }) {
  const n = vectorY.length;

  const vectorX = createEmptyMatrix(n , 1);

  for (let i = n - 1; i>=0; i-=1) {
    let sum = 0;
    for (let j = i + 1; j < n; j+=1) {
      sum += matrix[i][j]*vectorX[j][0];
    }

    vectorX[i][0] = (vectorY[i][0] - sum )/ matrix[i][i];
  }

  return vectorX;
}

module.exports = getXFromY;
```

- utils/isSymetric.js

```
function isSymetric(matrix) {
  for (let i = 0; i < matrix.rows; i++) {
    for (let j = 0; j <= i; j++) {
      if (matrix[i][j] !== matrix[j][i]) {
        return false;
      }
    }
  }
  return true;
}

module.exports = isSymetric;
```

- utils/LUHelper.js

```
const createEmptyMatrix = require('./createEmptyMatrix');

function determinantLU(matrix, pivot) {
  const n = matrix.length;
  let det = 1;
  for (let i = 0; i < n; i++) {
    det *= matrix[i][i];
  }

  if (pivot % 2) {
    console.log('Inverteu o sinal do det');
    det = -det;
  }

  console.log({ det });
  return det;
}

function LUDecomposition(n, matrixA, shouldCalculateDeterminant) {

  let pivot = 0;
  const matrixL = createEmptyMatrix(n,n);
  for (let i = 0; i < n; i++) matrixL[i][i] = 1;

  for (let k = 0; k < n - 1; k++) {
    // agora vou percorrer todas as linhas e pegar o maior valor na posição k,k
    let biggerValue = matrixA[k][k];
    let biggerLine = k;
    for (let i = k + 1; i < n; i++) {
      if (Math.abs(biggerValue) < Math.abs(matrixA[i][k])) {
        biggerValue = matrixA[i][k];
        biggerLine = i;
      }
    }

    //se biggerLine nao for k, trocar essas linhas
    if (biggerLine !== k) {
      const temp = matrixA[k];
      matrixA[k] = matrixA[biggerLine];
      matrixA[biggerLine] = temp;
      pivot++;
    }

    for (let i = k + 1; i < n; i++) {
      const m = - matrixA[i][k]/matrixA[k][k];
      matrixL[i][k] = -m;
      for (let j = k; j < n; j++) {
        matrixA[i][j] = m * matrixA[k][j] + matrixA[i][j];
      }
    }
  }

  return {
    matrixL,
    matrixU: matrixA,
    determinant: shouldCalculateDeterminant ? determinantLU(matrixA, pivot) : undefined,
  };
}

module.exports = LUDecomposition;
```

- utils/transpose.js

```
const createEmptyMatrix = require('./createEmptyMatrix');

function transpose(matrix) {
  const n = matrix.length;
  const transposed = createEmptyMatrix(n,n);

  for (let i = 0; i < n; i++) {
    for (let j = 0; j < n; j++) {
      transposed[i][j] = matrix[j][i];
    }

  }
  return transposed;
}

module.exports = transpose;
```

- methods/CholeskyDecomposition.js

```
const isSymetric = require('../utils/isSymetric');
const transpose = require('../utils/transpose');
const generateY = require('../utils/generateY');
const getXFromY = require('../utils/getXfromY');
const createEmptyMatrix = require('../utils/createEmptyMatrix');
const LUDecomposition = require('../utils/LUHelper');

function CholeskyDecomposition({ n, matrixA, vectorB, shouldCalculateDeterminant }) {
  if (!isSymetric(matrixA)) throw new Error('Matrix isn\'t symetric');

  const matrixG = createEmptyMatrix(n,n);

  for (let i = 0; i < n; i++) {
    let sum = 0;
    for (let j = 0; j < i; j++) {
      sum += matrixG[i][j]*matrixG[i][j];
    }

    const newValue = Math.sqrt(matrixA[i][i] - sum);
    if (newValue <= 0) throw new Error('Matrix isn\'t definite positive!');
    matrixG[i][i] = newValue;

    for (let j = i + 1; j < n; j++) {
      sum = 0;
      for (let k = 0; k < i; k++) sum += matrixG[i][k]*matrixG[j][k];
      matrixG[j][i] = (1/matrixG[i][i]) * (matrixA[i][j]-sum );
    }
  }

  const matrixGT = transpose(matrixG);

  const vectorY = generateY({ n, matrix: matrixG, vectorB });

  console.log({ matrixG, matrixGT, vectorY });

  const vectorX = getXFromY({ vectorY, matrix: matrixGT });

  const { determinant } = shouldCalculateDeterminant ? LUDecomposition(n, matrixA, shouldCalculateDeterminant) : undefined;

  return {
    vectorX,
    determinant,
  };
}

module.exports = CholeskyDecomposition;
```

- methods/GaussSeidel.js

```
const createEmptyMatrix = require('../utils/createEmptyMatrix');
const LUDecomposition = require('../utils/LUHelper');
const getResidue = require('../utils/getResidue');

function calculateNewPossibleSolution(matrixA, vectorX, vectorB) {
  const newVectorX = createEmptyMatrix(vectorX.length, 1);

  for (let i = 0; i < vectorX.length; i++) {
    let sum1 = 0;
    for (let j = 0; j < i; j++) {
      sum1 += matrixA[i][j]*newVectorX[j][0];
    }

    let sum2 = 0;
    for (let k = i + 1; k < matrixA.length; k++) {
```

```
      sum2 += matrixA[i][k]*vectorX[k][0];
    }

    newVectorX[i][0] = (vectorB[i][0] - sum1 - sum2)/matrixA[i][i];
  }

  return newVectorX;
}

function GaussSeidel({ n, matrixA, vectorB, shouldCalculateDeterminant, tol }) {
  let possibleSolution = createEmptyMatrix(n,1);
  let i = 0;
  let residue;
  const residues = [];

  while (i < 10000) {
    i += 1;
    const newPossibleSolution = calculateNewPossibleSolution(matrixA, possibleSolution, vectorB);

    residue = getResidue(newPossibleSolution, possibleSolution);
    residues.push(Number.isNaN(residue) ? Infinity : residue);

    if (residue > tol) {
      possibleSolution = newPossibleSolution;
    } else {
      const { determinant } = shouldCalculateDeterminant ? LUDecomposition(n, matrixA, shouldCalculateDeterminant) : undefined;

      return {
        vectorX: newPossibleSolution,
        iterations: i,
        determinant,
        residue: Number.isNaN(residue) ? Infinity : residue,
        residues,
      };
    }
  }

  const { determinant } = shouldCalculateDeterminant ? LUDecomposition(n, matrixA, shouldCalculateDeterminant) : undefined;

  return {
    vectorX: possibleSolution,
    iterations: i,
    determinant,
    residue: Number.isNaN(residue) ? Infinity : residue,
    residues,
    errors: ['possibilidade de não convergência']
  };
}

module.exports = GaussSeidel;
```

- methods/Jacobi.js

```
 const createEmptyMatrix = require('../utils/createEmptyMatrix');
const LUDecomposition = require('../utils/LUHelper');
const getResidue = require('../utils/getResidue');

function calculateNewPossibleSolution(matrixA, vectorX, vectorB) {
  const newVectorX = createEmptyMatrix(vectorX.length, 1);

  for (let i = 0; i < vectorX.length; i++) {
    let sum = 0;
    for (let j = 0; j < vectorX.length; j++) {
      if (i !== j) sum += matrixA[i][j]*vectorX[j][0];
    }

    newVectorX[i][0] = (vectorB[i][0] - sum)/matrixA[i][i];
  }

  return newVectorX;
}

function Jacobi({ n, matrixA, vectorB, shouldCalculateDeterminant, tol }) {
  let possibleSolution = createEmptyMatrix(n,1);
  let i = 0;
  let residue;
  const residues = [];
  while (i < 10000) {
    i += 1;
    const newPossibleSolution = calculateNewPossibleSolution(matrixA, possibleSolution, vectorB);

    residue = getResidue(newPossibleSolution, possibleSolution);
    residues.push(Number.isNaN(residue) ? Infinity : residue);

    if (residue > tol) {
      possibleSolution = newPossibleSolution;
    } else {

      const { determinant } = shouldCalculateDeterminant ? LUDecomposition(n, matrixA, shouldCalculateDeterminant) : undefined;

      return {
        vectorX: newPossibleSolution,
        iterations: i,
        determinant,
        residue: Number.isNaN(residue) ? Infinity : residue,
        residues,
      };
    }
  }

  const { determinant } = shouldCalculateDeterminant ? LUDecomposition(n, matrixA, shouldCalculateDeterminant) : undefined;

  return {
    vectorX: possibleSolution,
    iterations: i,
    determinant,
    residue: Number.isNaN(residue) ? Infinity : residue,
    residues,
    errors: ['possibilidade de não convergência']
  };
}

module.exports = Jacobi;
```

- methods/LUDecomposition.js

```
 const generateY = require('../utils/generateY');
const LUDecomposition = require('../utils/LUHelper');
const getXFromY = require('../utils/getXfromY');

function resolveXThroughLUDecomposition({ n, matrixA, vectorB, shouldCalculateDeterminant }) {

  const {
    matrixL,
    matrixU,
    determinant,
  } = LUDecomposition(n, matrixA, shouldCalculateDeterminant);

  const vectorY = generateY({ n, matrix: matrixL, vectorB });

  console.log({ matrixL, matrixU, vectorY });

  const vectorX = getXFromY({ vectorY, matrix: matrixU });

  return {
    vectorX,
    determinant,
  };
}

module.exports = resolveXThroughLUDecomposition;
```