Hilbert Matrix

# Overview

The tasks for part one are completed using three java files: HilbertOps.java, LUFactorization.java, and QRFactorization.java. The two Factorization classes, LUFactorization and QRFactorization, contain the methods to factorize matrices. LUFactorization.java contains **lu\_fact()**, which returns an LUFactorization Object. QRFactorization.java contains **qr\_fact\_househ()**, and **qr\_fact\_givens()**, both of which return a QRFactorization Object.

The QRFactorization and LUFactorization Objects both encapsulate all of the pertinent information about their factorization, including the factor matrices and the error (||LU – A||, for example).

These Objects can be passed to the **solve\_lu\_b()** and the **solve\_qr\_b()** methods, respectively, to solve the equation Ax = b, given a b vector.

HilbertOps.java contains all of the helper methods used throughout this part of the project. The Ops class in the General package of the project also contains some helper methods used in this part, however they were placed in another package because they are universally useful throughout the project.

# Conventions

In code, a matrix is represented as an array of arrays of 64 bit, signed, floating-point numbers. Each individual array within the matrix represents a row of the matrix. Vectors are, at times, represented as simple arrays of 64 bit, signed, floating-point numbers, and are independent of an orientation. The transpose of a vector, when it is needed, is accomplished by the way the data is treated. The orientation of a vector is assumed to be the only orientation that would make the problem solvable.

Although the actual required functions that enact the logic of the project (**solve\_lu\_b**, for example), cannot be called directly from the command line, there are wrapper classes that go by the same name that allow command line execution. For information about how to run individual parts of the project, see the readme file for part 1.

1. **lu\_fact()** is implemented as a static method in the LUFactorization class. It returns and LUFactorization Object, which contains the L and U matrices, and the error ||LU – A||.
2. **qr\_fact\_househ()** and **qr\_fact\_givens()** are implemented in the QRFactorization class. They both return QRFactorization Objects, which contain the Q and R matrices, as well as the error ||QR – A||
3. **solve\_lu\_b()** and **solve\_qr\_b()** are both implemented in the HilbertOps class. It uses backwards substitution to find the solution, rather than taking the inverse of any matrices. Taking the inverse of the matrices would defeat the purpose of the algorithm, as we factored the matrices to avoid the computationally expensive of taking an inverse every time the equation needed to be solved.
4. In order to run the program and print the solutions, see the Readme part 1.md file.

For LU factorization:

For QR factorization using Householder reflections:

For QR factorization using Givens rotations:

1. Written components
   * 1. Calculating the inverse of a matrix is a computationally expensive task. Although Calculating an LU or QR factorization would not allow for performance gains when solving the equation once, the same factorization could be used next time to solve for a different, b. This makes factorization worthwhile, as solving a system using an LU or QR factorization requires far fewer operations than would calculating an inverse and then multiplying.
     2. By using and LU or QR factorizations, one can reduce the time complexity of solving a system over time. Additionally, the methods used to solve an equation using one of these factorizations avoids multiplying by what could easily be an ill-conditioned matrix. Calculating the QR factorization of a matrix can be done without dramatic error increase, and it creates an orthonormal matrix, which is a well-conditioned matrix.