MATH 2605 Project

Ryan Ashcraft
John Ough
Tanner Smith

MATH 2605, Spring 2011, Section C1

Professor Luz Vela-Arevalo

April 15, 2011

Part I
Hilbert Matrix

For part one we are working with a Hilbert matrix. A Hilbert matrix is a matrix with all entries being unit fractions found by the following formula:

$$\frac{1}{row + column - 1}$$

We are trying to find different decompositions of square Hilbert matrices with sizes ranging from 2x2 to 20x20. The decompositions are found for LU, QR using Householder reflections, and QR using Givens rotations. Then we solve the system for Hx=b for a vector b which is filled with 1's. We solve for x using different ways based on each decomposition that we did. Then we find the errors and compare them using the infinite norms of llLU-Hll, llQR-Hll, and llHx-bll. After these errors are computed we plot them on a graph to compare their values.

Hilbert Matrix

We made the code to create our Hilbert matrix by using loops which can give us the row and column positions to create the unit fractions for each position in the matrix. As our loop progresses it stores the appropriate values in a 2-dimensional array which we will pass into a constructor to create a Matrix object. We take in an integer (columns) an a parameter in the constructor then make a square Hilbert matrix of size columns x columns.

Code for Hilbert Matrix

```
public Hilbert(int columns) {
       size = columns;
       b = new Matrix(columns, 1, 1);
       array = new double[columns][columns];
       for (int i = 0; i < array.length; i++) {
              for(int j = 0; j < array[i].length; j++) {
                     array[i][j] = (1.0 / (i + j + 1));
              }
       }
       matrix = new Matrix(array);
}
```

Question 1 - LU Decomposition

In this question we will be finding the LU decomposition of a Hilbert matrix then solving for Hx=b. After finding the solution x, we found the errors of IILU-HII and IIHx-bII for each iteration of Hilbert matrices.

To begin, our code decomposes H into the two matrices L and U. To implement LU decomposition we had our code use the Crout algorithm to solve to LU.

```
public void LUDecomp() {
       double[][] LU = matrix.getArrayCopy();
       int size = matrix.getRowDimension();
       size = matrix.getColumnDimension();
       double[] LUrowi;
       double[] LUcolj = new double[size];
       for (int j=0; j < size; j++) {
              for (int i = 0; i < size; i++) {
                     LUcolj[i] = LU[i][j];
              }
              for (int i = 0; i < size; i++) {
                     LUrowi = LU[i];
                     int kmax = Math.min(i, j);
                     double temp = 0.0;
                     for(int k = 0; k < kmax; k++) {
                                   temp += LUrowi[k] * LUcolj[k];
                     }
                     LUrowi[j] = LUcolj[i] -= temp;
              }
              int p = j;
              for (int i = j + 1; i < size;i ++) {
                     if(Math.abs(LUcolj[i]) > Math.abs(LUcolj[p])) {
                            p = i;
                     }
              }
              if (p != j) {
                     for (int k = 0; k < size; k++) {
                            double t = LU[p][k];
                            LU[p][k] = LU[j][k];
                            LU[j][k] = t;
                     }
              }
              if (j < size && LU[j][j] != 0.0) {
                     for(int i = j + 1; i < size; i++) {
                            LU[i][j] /= LU[j][j];
                     }
              }
       }
       l = new Matrix(size,size);
       double[][] L = l.getArray();
       for (int i = 0; i < size; i++) {
              for (int j = 0; j < size;j ++) {
                     if (i > j) {
                            L[i][j] = LU[i][j];
                     } else if (i == j) {
                            L[i][j] = 1.0;
```

```
                } else {
                    L[i][j] = 0.0;
                }
            }
        }
        u = new Matrix(size, size);
        double[][] U = u.getArray();
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                if (i <= j) {
                    U[i][j] = LU[i][j];
                } else {
                    U[i][j] = 0.0;
                }
            }
        }
}
```

After we find the LU decomposition of H, the solution x to the system Hx = LUx = b can be solved by solving the system of equations Ly = b by forward substitution for y, and then solving the system of equations Ux = y by backward substitution for x.

```
public Matrix solveLU() {
        Matrix y = new Matrix(size, 1, 0);
        for (int i = 0; i < size; i++) {
            double t = 0.0;
            for (int j = 0; j < i; j++) {
                t += l.get(i, j) * y.get(j, 0);
            }
            y.set(i, 0, (b.get(i, 0) - t) / l.get(i, i));
        }
        Matrix x = new Matrix(size, 1, 0);
        for (int j = x.getRowDimension() - 1; j >= 0; j--) {
            double t = 0.0;
            for (int k = j + 1; k < y.getRowDimension(); k++) {
                t += u.get(j, k) * x.get(k, 0);
            }
            x.set(j, 0, (y.get(j, 0) - t) / u.get(j, j));
        }
        return x;
}
```

The two errors of the LU decomposition are found using the infinite norms of llLU-Hll and llHx-bll. The first is the infinite norm of a matrix which is found by calculating the maximum value for a row of LU-H. The code for this error will calculate LU-H then loop through each column and add up the values then checks it against a max variable. If the new sum is bigger, max is replaced. The second is the infinite norm of a vector which is the maximum absolute value of the entries in Hx-b. This code takes in a x vector, calculates Hx-b, then loops through it comparing the absolute value of each entry against a max variable.

```
public double error1QR() {
	Matrix a = q.times(r).minus(matrix);
	double max = 0;
	double temp = 0;
	for (int i = 0; i < a.getRowDimension(); i++) {
		temp = 0;
		for (int j = 0; j < a.getColumnDimension(); j++) {
			temp += Math.abs(a.get(i, j));
		}
		if (temp > max) {
			max = temp;
		}
	}
	return max;
}

public double error2(Matrix x) {
	double max = 0;
	Matrix a = matrix.times(x).minus(b);
	for (int i = 0; i < a.getRowDimension(); i++) {
		if (Math.abs(a.get(i, 0)) > max) {
			max = a.get(i, 0);
		}
	}
	return max;
}
```

Question 2 - QR Decomposition using Householder Reflections

In this question we will be finding the QR decomposition of a Hilbert matrix by using Householder reflections. We then use the decomposition to solve for Hx=b. After finding the solution x, we found the errors of IIQR-HII and IIHx-bII for each iteration of Hilbert matrices.

The program first calculates the decomposition Q and R from a given Hilbert matrix. Our algorithm loops through the matrix n times and performs the following calculations at each step. It calculates the 2 norm for each column n. Then it creates the n-th Householder reflection by dividing that column by the norm and adding one to the diagonal. After we have the Householder reflection is performs the transformation to the remaining columns. Then is stores the negative norm as a diagonal value for R. It then separates QR into it's components Q and R.

```
public void householderQR() {
	double[][] QR = matrix.getArrayCopy();
	double[] Rdiag = new double[size];
	for (int k = 0; k < size; k++) {
		double norm = 0;
		for (int i = k; i < size; i++) {
			norm = Maths.hypot(norm, QR[i][k]);
		}
		if (norm != 0.0) {
			if (QR[k][k] < 0) {
				norm =- norm;
```

```
            }
            for (int i = k; i < size; i++) {
                QR[i][k] /= norm;
            }
            QR[k][k] += 1.0;
            for (int j = k + 1; j < size; j++) {
                double scale = 0.0;
                for (int i = k; i < size; i++) {
                    scale += QR[i][k] * QR[i][j];
                }
                scale =- scale / QR[k][k];
                for (int i = k; i < size; i++) {
                    QR[i][j] += scale * QR[i][k];
                }
            }
        }
        Rdiag[k] =- norm;
    }
    double[][] R = new double[size][size];
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i < j) {
                R[i][j] = QR[i][j];
            } else if (i == j) {
                R[i][j] = Rdiag[i];
            } else {
                R[i][j] = 0.0;
            }
        }
    }
    r = new Matrix(R);
    double[][] Q = new double[size][size];
    for (int k = size - 1; k >= 0; k--) {
        for (int i = 0; i < size; i++) {
            Q[i][k] = 0.0;
        }
        Q[k][k] = 1.0;
        for (int j = k; j < size; j++) {
            if (QR[k][k] != 0) {
                double temp = 0.0;
                for (int i = k; i < size; i++) {
                    temp += QR[i][k] * Q[i][j];
                }
                temp =- temp / QR[k][k];
                for (int i = k; i < size; i++) {
                    Q[i][j] += temp * QR[i][k];
                }
            }
        }
    }
    q = new Matrix(Q);
}
```

To calculate x for each Hilbert matrix we solve the equation Rx=Q^tb. To start solving for x we first solve for the right side of the equation, Q^tb. Then we can use back substitution for Rx=y to solve for x since R is an upper triangular matrix.

```
public Matrix solveQR() {
	Matrix y = q.transpose().times(b);
	Matrix x = new Matrix(size, 1, 0);
	for (int j = x.getRowDimension()-1;j>=0;j--)
	{
		double t = 0.0;
		for(int k = j + 1; k < y.getRowDimension(); k++) {
			t += r.get(j, k) * x.get(k, 0);
		}
		x.set(j, 0, (y.get(j, 0) - t) / r.get(j, j));
	}
	return x;
}
```

The two errors of the QR decomposition are found using the infinite norms of IIQR-HII and IIHx-bII. The first is the infinite norm of a matrix which is found by calculating the maximum value for a row of QR-H. The code for this error will calculate QR-H then loop through each column and add up the values then checks it against a max variable. If the new sum is bigger, max is replaced. The second error is the infinite norm of a vector which is the maximum absolute value of the entries in Hx-b. This code takes in a x vector, calculates Hx-b, then loops through it comparing the absolute value of each entry against a max variable.

```
public double error1QR() {
	Matrix a = q.times(r).minus(matrix);
	double max = 0;
	double temp = 0;
	for (int i = 0; i < a.getRowDimension(); i++) {
		temp = 0;
		for (int j = 0; j < a.getColumnDimension(); j++) {
			temp += Math.abs(a.get(i, j));
		}
		if (temp > max) {
			max = temp;
		}
	}
	return max;
}

public double error2(Matrix x) {
	double max = 0;
	Matrix a = matrix.times(x).minus(b);
	for (int i = 0; i < a.getRowDimension(); i++) {
		if (Math.abs(a.get(i, 0)) > max) {
			max = a.get(i, 0);
		}
	}
	return max;
}
```

Question 3 - QR Decomposition using Givens Rotations

In this question we will be finding the QR decomposition of a Hilbert matrix by using Givens rotations. We then use the decomposition to solve for Hx=b. After finding the solution x, we found the errors of IIQR-HII and IIHx-bII for each iteration of Hilbert matrices.

To decompose each Hilbert matrix we used the following code to calculate Q and R using Givens rotations. The code will loop through the matrix n-1 times eliminating zeros below the diagonal to create R. It will start by copying the Hilbert matrix into R then it will start the loop and take each element of R below the diagonal starting with the bottom left and create a Givens rotation matrix. It will then multiply by the current R to zero each spot. Along the way it will save each rotation matrix and multiply them all together at the end to find Q.

```
public void givensQR() {
       r = new Matrix(matrix.getArrayCopy());
       ArrayList<Matrix> gs = new ArrayList<Matrix>();
       for (int i = 0; i < size - 1; i++) {
              for (int j = size - 1; j > i; j--) {
                     Matrix g = Matrix.identity(size, size);

                     double x = r.get(j - 1, i);
                     double y = r.get(j, i);
                     double cos = x / Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));
                     double sin = -y / Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2));

                     g.set(j - 1, j - 1, cos);
                     g.set(j - 1, j, -sin);
                     g.set(j, j - 1, sin);
                     g.set(j, j, cos);

                     gs.add(g);
                     r = g.times(r);
              }
       }
       q = new Matrix(gs.get(0).getArrayCopy());
       for (int i = 1; i < gs.size(); i++) {
              q=gs.get(i).times(q);
       }
       double[][] temp = r.getArray();
       for (int i = 0; i < temp.length - 1; i++) {
              for (int j = 0; j < temp[0].length; j++) {
                     temp[i][j] =- temp[i][j];
              }
       }
       temp = q.getArray();
       for (int i = 0; i < temp.length - 1; i++) {
              for (int j = 0; j < temp[0].length; j++) {
                     temp[i][j] =- temp[i][j];
              }
       }
}
```

To calculate x for each Hilbert matrix we solve the equation Rx=G^tb. To start solving for x we first solve for the right side of the equation, Q^tb. Then we can use back substitution for Rx=y to solve for x since R is an upper triangular matrix.

```
public Matrix solveQR2() {
	Matrix y = q.transpose().times(b);
	Matrix x = new Matrix(size, 1, 0);
	for (int j = x.getRowDimension() - 1; j >= 0; j--) {
		double t = 0.0;
		for (int k = j + 1; k < y.getRowDimension(); k++) {
			t += r.get(j, k) * x.get(k,0);
		}
		x.set(j,0,(y.get(j,0)-t)/r.get(j,j));
	}
	return x;
}
```

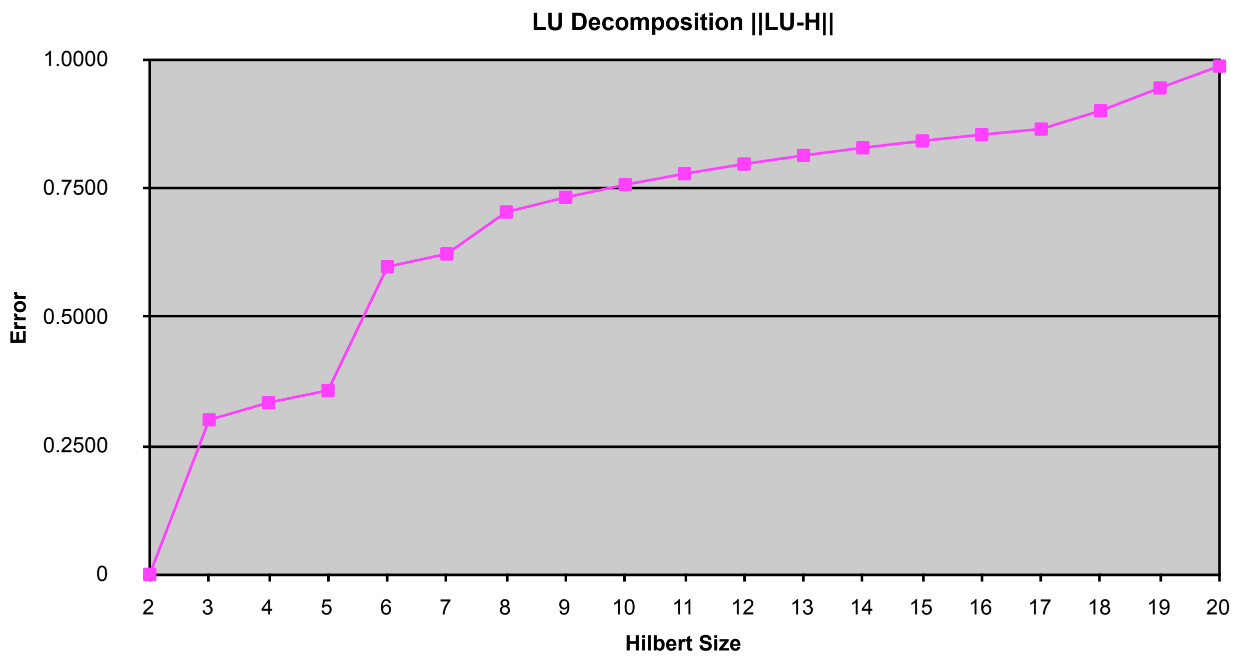
The two errors of the QR decomposition are found using the infinite norms of IIQR-HII and IIHx-bII. The first is the infinite norm of a matrix which is found by calculating the maximum value for a row of QR-H. The code for this error will calculate QR-H then loop through each column and add up the values then checks it against a max variable. If the new sum is bigger, max is replaced. The second error is the infinite norm of a vector which is the maximum absolute value of the entries in Hx-b. This code takes in a x vector, calculates Hx-b, then loops through it comparing the absolute value of each entry against a max variable.

```
public double error1QR2() {
	Matrix a = q.transpose().times(r).minus(matrix);
	double max = 0;
	double temp = 0;
	for (int i = 0; i < a.getRowDimension(); i++) {
		temp = 0;
		for (int j = 0; j < a.getColumnDimension(); j++) {
			temp += Math.abs(a.get(i ,j));
		}
		if (temp > max) {
			max = temp;
		}
	}
	return max;
}

public double error2(Matrix x)
{
	double max = 0;
	Matrix a = matrix.times(x).minus(b);
	for (int i = 0; i < a.getRowDimension(); i++) {
		if (Math.abs(a.get(i, 0)) > max) {
			max = a.get(i, 0);
		}
	}
	return max;
}
```

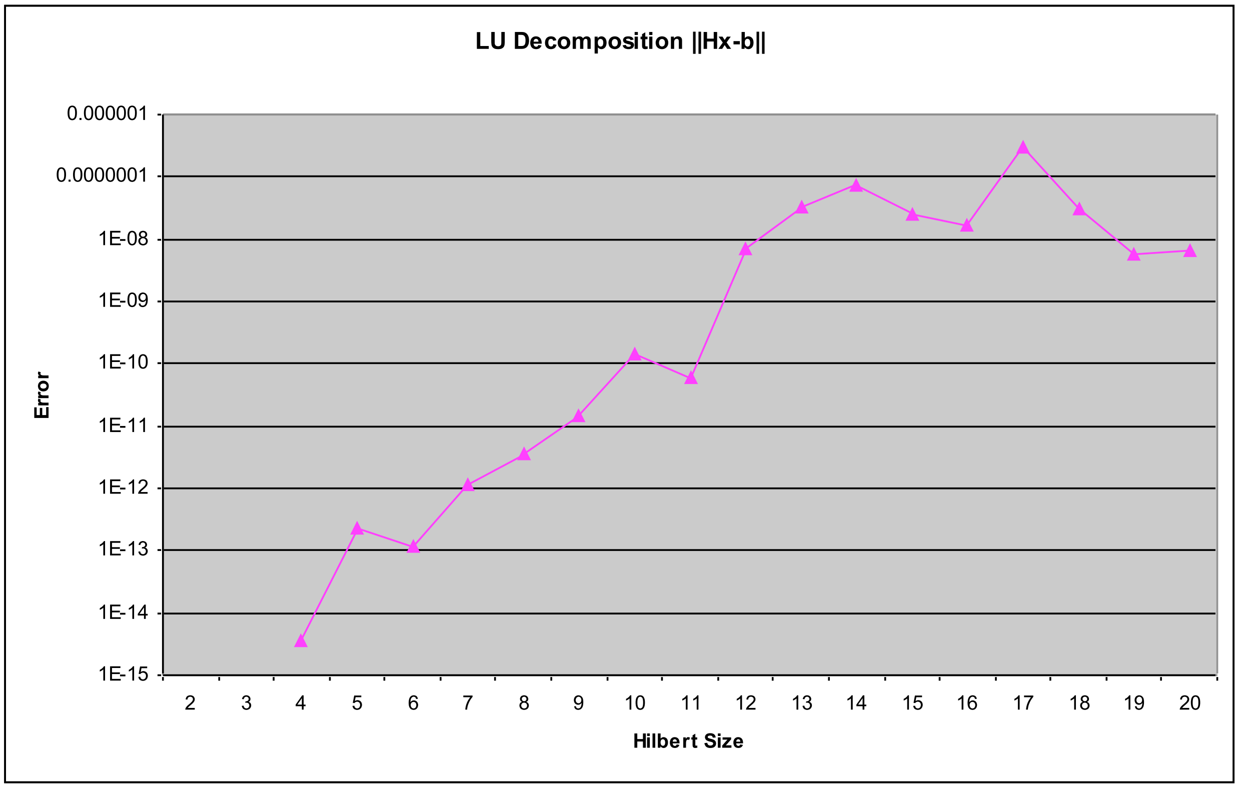
Results

Hilbert Matrix Size	LU Decomposition ||LU-H||	LU Decomposition ||Hx-b||	QR Decomposition Householder ||QR-H||	QR Decomposition Householder ||Hx-b||	QR Decomposition Givens ||QR-H||	QR Decomposition Givens ||Hx-b||
2	0	0	2.22E-16	4.44E-16	1.11E-16	4.44E-16
3	0.3	0	3.33E-16	8.88E-16	3.05E-16	8.88E-16
4	0.333333	3.55E-15	2.78E-16	3.55E-15	3.61E-16	1.07E-14
5	0.357143	2.32E-13	3.89E-16	1.14E-13	3.89E-16	2.84E-14
6	0.597222	1.14E-13	6.11E-16	5.68E-14	3.05E-16	1.71E-13
7	0.622222	1.14E-12	9.16E-16	1.59E-12	4.02E-16	1.14E-12
8	0.703596	3.64E-12	1.03E-15	1.64E-11	3.19E-16	5.46E-12
9	0.732005	1.46E-11	1.22E-15	2.18E-11	5.34E-16	4.73E-11
10	0.756515	1.46E-10	1.85E-15	3.64E-11	3.61E-16	1.24E-10
11	0.777883	5.82E-11	6.94E-16	1.75E-09	8.05E-16	1.51E-09
12	0.79668	6.98E-09	2.10E-15	1.49E-08	1.14E-15	8.85E-09
13	0.813346	3.26E-08	9.58E-16	4.77E-07	3.54E-16	1.14E-07
14	0.828227	7.36E-08	1.97E-15	3.03E-09	9.71E-16	1.35E-08
15	0.841596	2.42E-08	2.53E-15	1.16E-09	6.31E-16	2.61E-08
16	0.853674	1.68E-08	3.04E-15	7.17E-08	1.36E-15	3.82E-08
17	0.864639	2.98E-07	3.00E-15	9.31E-09	1.52E-15	3.73E-09
18	0.900251	2.98E-08	3.96E-15	2.51E-08	8.60E-16	1.73E-06
19	0.94448	5.70E-09	4.28E-15	3.54E-08	7.32E-16	8.01E-08
20	0.986821	6.52E-09	1.67E-15	3.80E-07	9.37E-16	2.79E-08

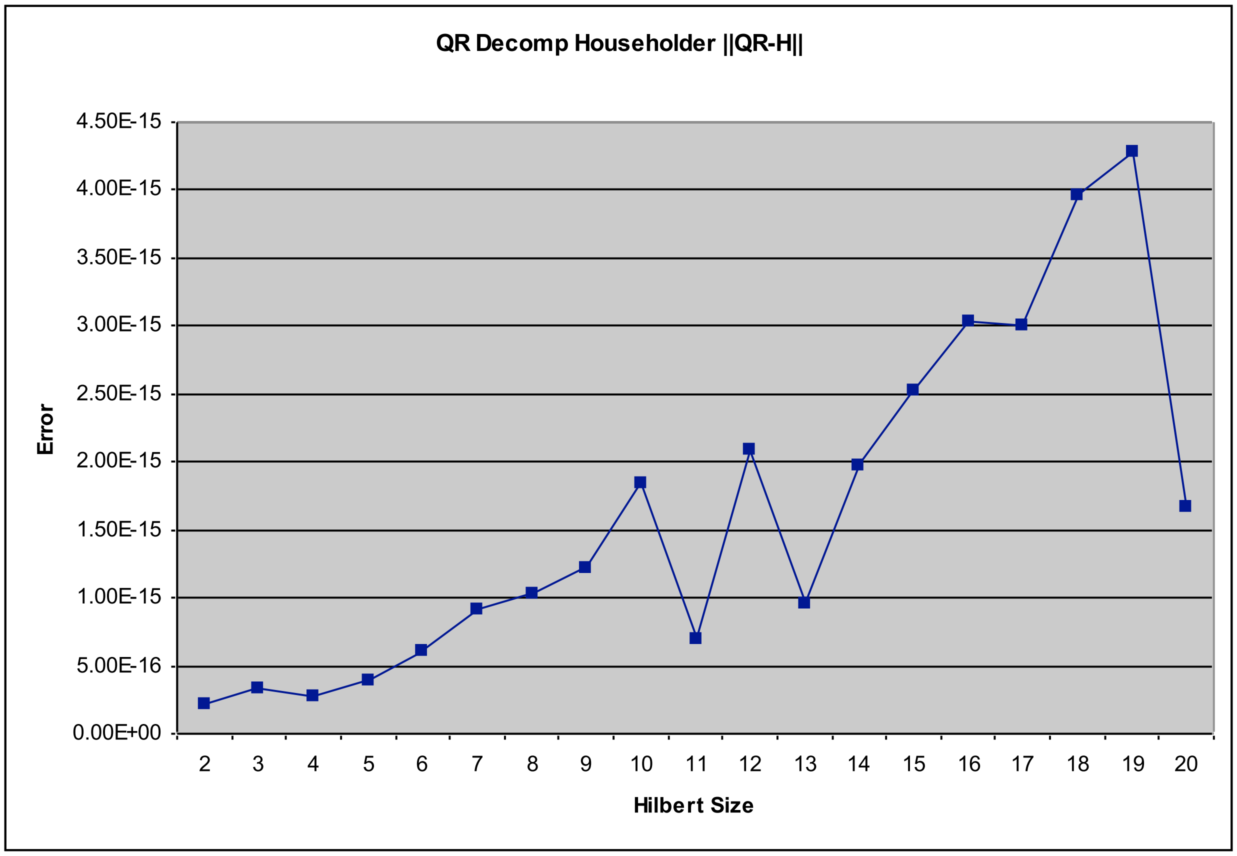
These are the results of the three parts in a chart with each row as a different size Hilbert matrix. Below are the charts plotting each error as y and the Hilbert matrix size as x. When comparing the error of IILU-HII, and the two errors IIQR-HII a pattern is noticed from the slopes of the lines. It shows that the LU decomposition has a very steady trend line going up even for larger Hilbert matrices. For the other two errors of II QR-HII it shows that when calculating smaller matrices, the slope is fairly flat and consistent. When calculating larger matrices around and above 10x10 the errors both jump up significantly. This shows that LU decomposition is better for smaller matrices, and QR would be better for matrices larger then 10x10. The next set of errors are found be solving for x using each decomposition then calculating IIHx-bII. For these errors, the three solutions are very similar to each other. All of the errors have a consistent rise till they level off when reaching Hilbert matrix sizes of 13x13 or above. The only observable difference is that the QR decomposition using Givens rotations seems to have a more consistent smooth trend both on the steep increase of error below 13x13, and after it starts to level off for the large sizes.

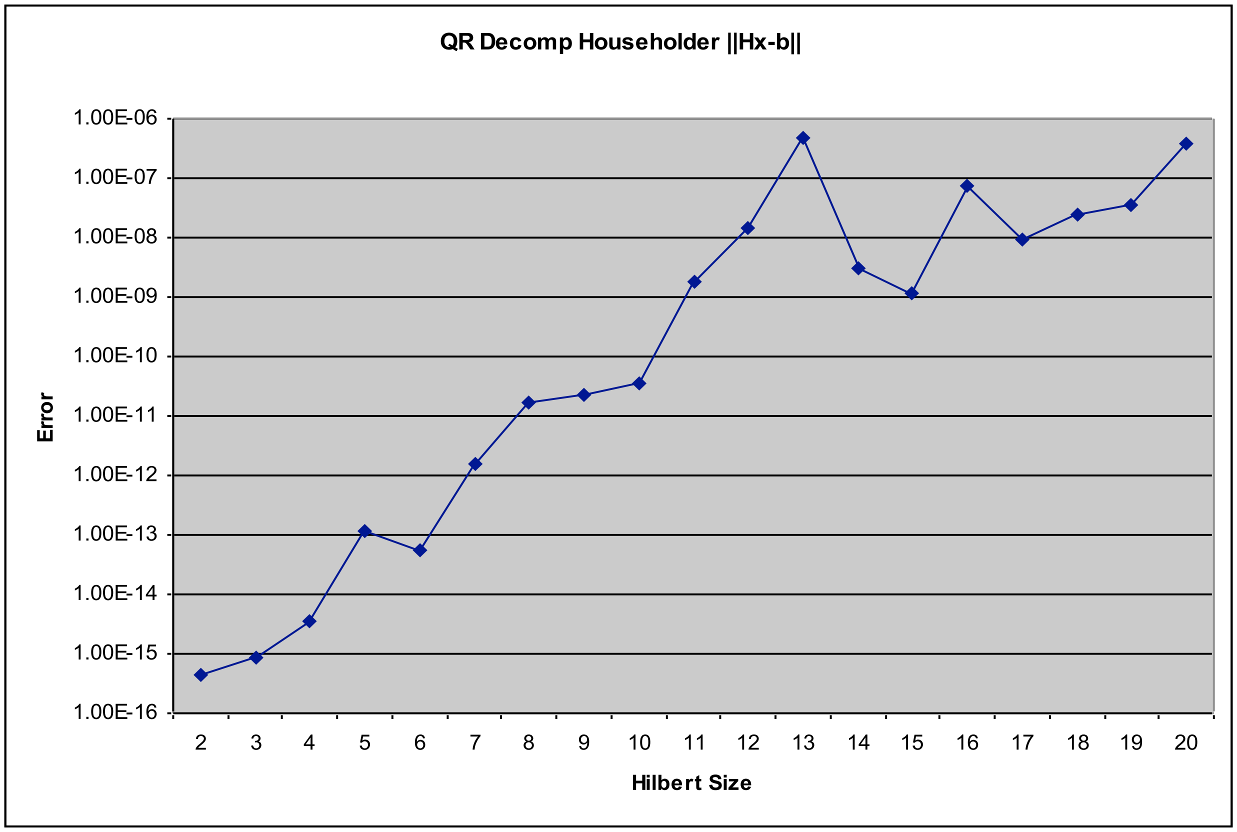


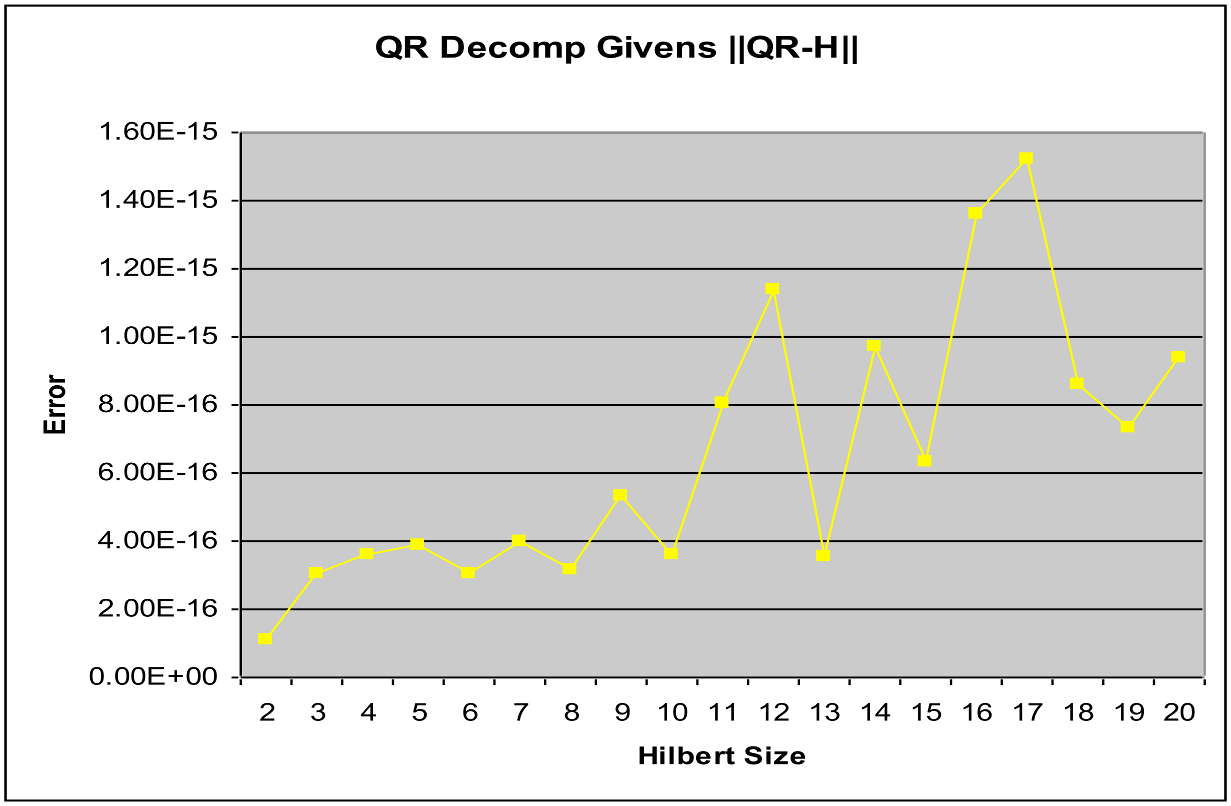
LU Decomposition ||LU-H||
1.0000
0.7500
0.5000
0.2500
0
Error
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Hilbert Size

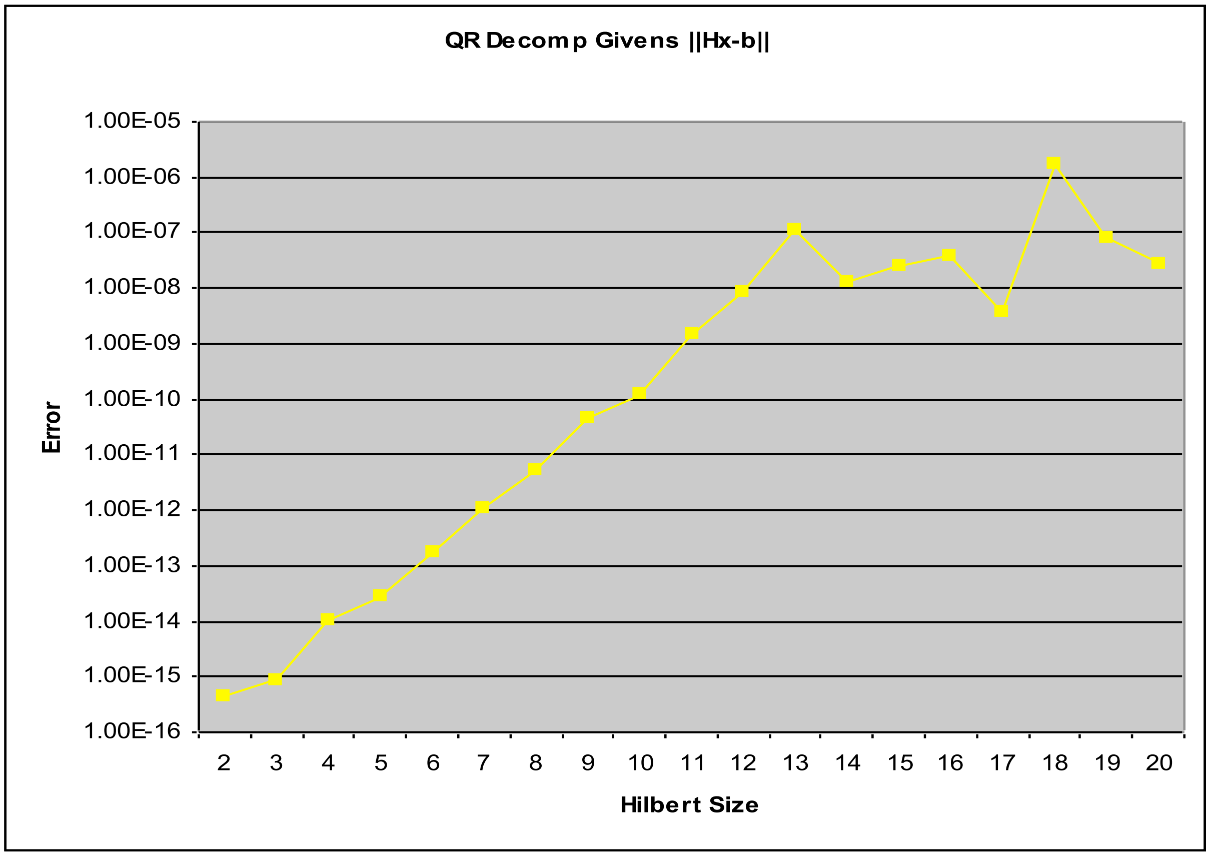


LU Decomposition ||Hx-b||
0.000001
0.0000001
1E-08
1E-09
1E-10
1E-11
1E-12
1E-13
1E-14
1E-15
Error
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
Hilbert Size


QR Decomp Householder ||QR-H||
4.50E-15
4.00E-15
3.50E-15
3.00E-15
2.50E-15
2.00E-15
1.50E-15
1.00E-15
5.00E-16
0.00E+00
Error
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Hilbert Size


QR Decomp Householder ||Hx-b||
1.00E-06
1.00E-07
1.00E-08
1.00E-09
1.00E-10
1.00E-11
1.00E-12
1.00E-13
1.00E-14
1.00E-15
1.00E-16
Error
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Hilbert Size


QR Decomp Givens ||QR-H||
Error
1.60E-15
1.40E-15
1.20E-15
1.00E-15
8.00E-16
6.00E-16
4.00E-16
2.00E-16
0.00E+00
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Hilbert Size


QR Decomp Givens ||Hx-b||
Error
1.00E-05
1.00E-06
1.00E-07
1.00E-08
1.00E-09
1.00E-10
1.00E-11
1.00E-12
1.00E-13
1.00E-14
1.00E-15
1.00E-16
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
Hilbert Size

Part II
Animation from Scratch

In part two, the program was to rotate three letters which were represented in a n x 3 matrix using linear transformations. The matrices that made up the shapes were made up by a number of X, Y, and Z points where the number of point sets was determined individually by the number of vertices to create each letter. Each of the letters had to rotate around a different axis, specifically the first letter had to rotate around the axis perpendicular to the frame, the second letter around the vertical axis, and the third letter around the horizontal axis.

Rotations

The individual letters were rotated by using the corresponding rotation matrix for the desired axis of rotation. The rotation matrices for each axis of rotation for any angle theta can be seen below.

$$x_{rot} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} y_{rot} = \begin{bmatrix} cos(\theta) & 0 & -sin(\theta) \\ 0 & 1 & 0 \\ sin(\theta) & 0 & cos(\theta) \end{bmatrix} z_{rot} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The code for each axis rotation matrix can be seen below.

Code for X-Axis Rotation

```
private static Matrix xRotatedMatrix(Matrix m, final int ROTATIONS, final int T, final
int TOTAL_FRAMES) {
	double angle = T * (ROTATIONS * Math.PI*2)/(TOTAL_FRAMES-1);

	Matrix rotationMatrix = new Matrix(new double[][] {
			{1.0, 0.0, 0.0},
			{0.0, Math.cos(angle), -Math.sin(angle)},
			{0.0, Math.sin(angle), Math.cos(angle)}
			});

	return rotatedMatrix(m, rotationMatrix);
}
```

Code for Y-Axis Rotation

```
private static Matrix yRotatedMatrix(Matrix m, final int ROTATIONS, final int T, final
int TOTAL_FRAMES) {
	double angle = T * (ROTATIONS * Math.PI*2)/(TOTAL_FRAMES-1);

	Matrix rotationMatrix = new Matrix(new double[][] {
			{Math.cos(angle), 0.0, -Math.sin(angle)},
			{0.0, 1.0, 0.0},
			{Math.sin(angle), 0.0, Math.cos(angle)}
```

```
                        });

        return rotatedMatrix(m, rotationMatrix);
}
```

Code for Z-Axis Rotation

```
private static Matrix zRotatedMatrix(Matrix m, final int ROTATIONS, final int T, final
int TOTAL_FRAMES) {
        double angle = T * (ROTATIONS * Math.PI*2)/(TOTAL_FRAMES-1);

        Matrix rotationMatrix = new Matrix(new double[][] {
                        {Math.cos(angle), -Math.sin(angle), 0.0},
                        {Math.sin(angle), Math.cos(angle), 0.0},
                        {0.0, 0.0, 1.0}
                        });

        return rotatedMatrix(m, rotationMatrix);
}
```

Code for Matrix Rotation

The code below does the actual rotation of the matrix that contains the points for the shapes. The method also contains the translation of the points to ensure that the points are rotated around the correct axis and displayed correctly in the end.

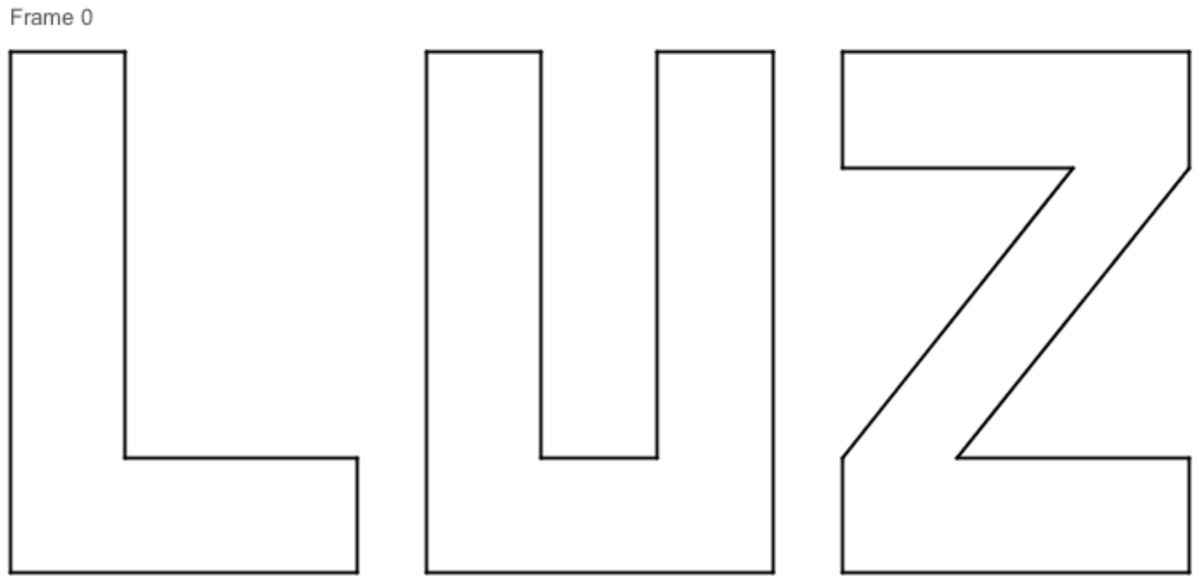
```
private static Matrix rotatedMatrix(Matrix m, Matrix r) {
        Matrix translationMatrix = new Matrix(m.getRowDimension(), m.getColumnDimension
());
        for (int c = 0; c < translationMatrix.getColumnDimension(); c++) {
                // T

                // x
                translationMatrix.set(0, c, 2.0);

                // y
                translationMatrix.set(1, c, 3.0);
        }

        Matrix rotatedMatrix = r.times(m.minus(translationMatrix)).plus
(translationMatrix);
        return rotatedMatrix;
}
```

Animation Sample Frames

Below are several frames from the animation and the matrices that correspond for the frames. Each matrix is shown by a 3 x n where each row is the X, Y, and Z component and the column is each point that makes up the specific shape.

For a full list of frames, see the attached document.

Frame 0



L

0.00	4.00	4.00	1.33	1.33	0.00	0.00
0.00	0.00	1.33	1.33	6.00	6.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00

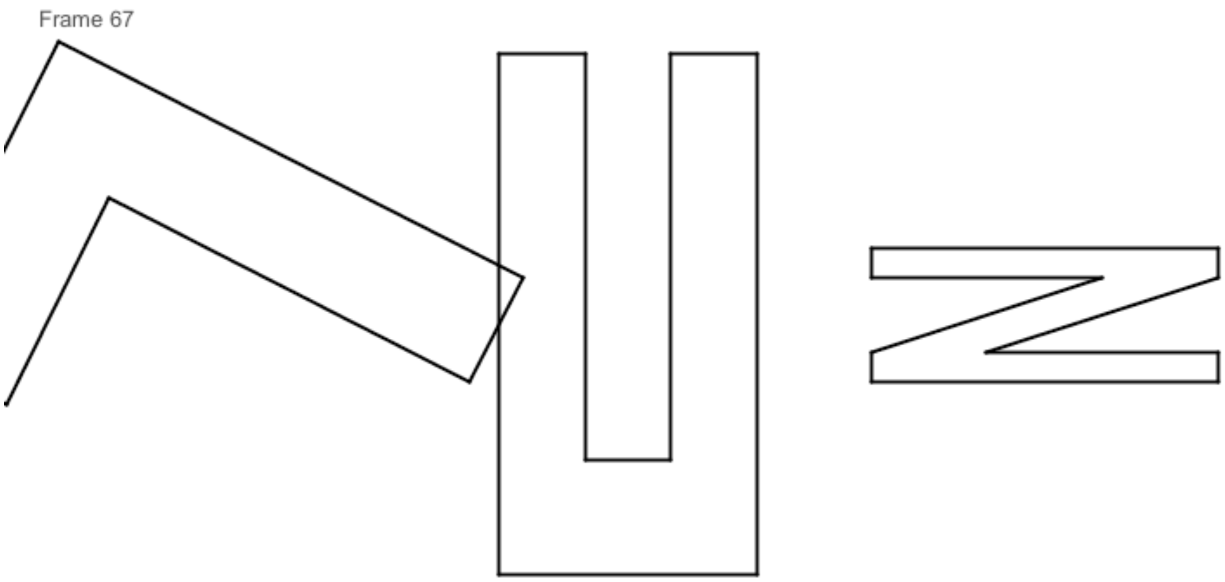
U

0.00	4.00	4.00	2.66	2.66	1.33	1.33	0.00	0.00
0.00	0.00	6.00	6.00	1.33	1.33	6.00	6.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Z

0.00	4.00	4.00	1.33	4.00	4.00	0.00	0.00	2.66	0.00	0.00
0.00	0.00	1.33	1.33	4.66	6.00	6.00	4.66	4.66	1.33	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Frame 67



L

0.23	-1.58	-0.40	0.82	4.98	5.58	0.23
6.14	2.58	1.98	4.36	2.24	3.42	6.14
0.00	0.00	0.00	0.00	0.00	0.00	0.00

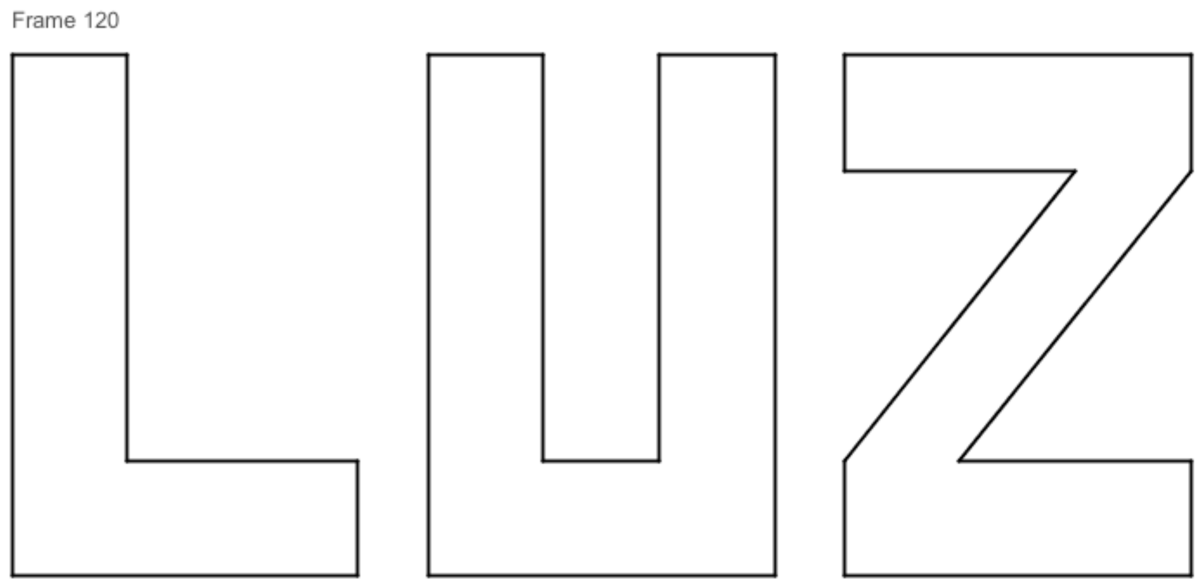
U

0.51	3.49	3.49	2.49	2.49	1.50	1.50	0.51	0.51
0.00	0.00	6.00	6.00	1.33	1.33	6.00	6.00	0.00
-1.34	1.34	1.34	0.44	0.44	-0.45	-0.45	-1.34	-1.34

Z

0.00	4.00	4.00	1.33	4.00	4.00	0.00	0.00	2.66	0.00	0.00
2.22	2.22	2.57	2.57	3.43	3.78	3.78	3.43	3.43	2.57	2.22
2.90	2.90	1.61	1.61	-1.60	-2.90	-2.90	-1.60	-1.60	1.61	2.90

Frame 120



```
L
  -0.00    4.00    4.00    1.33    1.33    0.00   -0.00
   0.00   -0.00    1.33    1.33    6.00    6.00    0.00
   0.00    0.00    0.00    0.00    0.00    0.00    0.00

U
   0.00    4.00    4.00    2.66    2.66    1.33    1.33    0.00    0.00
   0.00    0.00    6.00    6.00    1.33    1.33    6.00    6.00    0.00
   0.00   -0.00   -0.00   -0.00   -0.00    0.00    0.00    0.00    0.00

Z
   0.00    4.00    4.00    1.33    4.00    4.00    0.00    0.00    2.66    0.00    0.00
   0.00    0.00    1.33    1.33    4.66    6.00    6.00    4.66    4.66    1.33    0.00
   0.00    0.00    0.00    0.00   -0.00   -0.00   -0.00   -0.00   -0.00    0.00    0.00
```

Part III
Convolutional Codes

In part three we are working with convolutional codes which can be used as error control systems for telecommunications. We are implementing a ½ convolutional code to produce and decrypt bit strings. We feed a bit string x into a shift register which will take the input and create two output strings, y^0 and y^1. These two strings are multiplexed together to form the final output y which is the convolutional code word. The output y is a linear combination of the input stream x. The purpose of this question is to encode and decode streams of x and y. To encode a stream x, we can solve for a matrix A then use the equation where y=Ax. In the decoding process we use the same A as found for encoding, then use Jacobi iteration and Gauss-Seidel iteration.

Question 1 - Encoding

In question one we are asked to encode the following bit stream:

(1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0)

We can do this by using the equation y=Ax and use that to get the output stream y. To do this we first had to find two A's such that we could get y^0 and y^1.

To generate an A for y^0 and y^1 we used the following matrices:

$$y^0 = (1.0, 1.0, 0.0, 1.0)$$
$$y^1 = (1.0, 0.0, 1.0, 1.0)$$

Then once we have an A we can multiply by x to solve the equation y=Ax to solve both y's then concatenate them together to form the y stream.

```
public static Matrix convolude(Matrix x, Matrix a) {
	//Pad the X vector with zeros so it matches up with A
	double[][] newXArray = new double[1][a.getColumnDimension()];
	for (int i = 0; i < newXArray[0].length; i++) {
		if (i < x.getColumnDimension()) {
			newXArray[0][i] = x.get(0, i);
		} else {
			newXArray[0][i] = 0;
		}
	}
	Matrix newX = new Matrix(newXArray);
	//Multiply and perform binary addition
	Matrix y = a.times(newX.transpose());
	for (int i = 0; i < y.getRowDimension(); i++) {
		int value = (int)y.get(i, 0);
		if (value == 2) {
			y.set(i, 0, 0);
		} else if (value == 3) {
			y.set(i, 0, 1);
		}
```

```
        }
        return y;
}

public static Matrix createA(Matrix x, Matrix linearCombo) {
        int sizeOfA = x.getColumnDimension() +linearCombo.getColumnDimension() - 1;

        Matrix a = new Matrix(sizeOfA, sizeOfA, 0);

        int padding = 1 - linearCombo.getColumnDimension();
        for (int r = 0; r < a.getRowDimension(); r++) {
                for (int i = 0; i < linearCombo.getColumnDimension(); i++) {
                        if (i + padding >= 0) {
                                a.set(r, i + padding, linearCombo.get(0, i));
                        }
                }
                padding++;
        }
        return a;
}
```

Question 2 - Decode Convolution Code

In this question we were trying to decode the bit stream y into the input stream x. We can do this by using the same equation y=Ax but this time solving for x instead of y. The same equation works because we use the same matrix A as before. To solve the equation we use two different iterative methods, Jacobi and Gauss-Seidel.

The Jacobi iteration is solved by Sxplus=Tx+b. S is the diagonal of the matrix A, T is S-A, x is the current iteration, and xplus is the next iteration. We solve this in a loop solving the right side, Tx+b, then dividing by the diagonal of S to find xplus.

The Gauss-Seidel iteration uses the same equation Sxplus=Tx+b. Except this time S is the lower triangular matrix of A. So this time after we solve Tx+b we have to use back substitution to solve Sx=Tx+b for x.

One sample solution of decoding is taking the example value y=1101010111011100 output stream y, to its input stream x=10110. The Jacobi method could decode this stream accurately after 4 iterations. However the Gauss-Seidel method took 8 iterations to find the correct result.

A second example of decoding is the output stream y=10010111 to the input stream x=10101. This time the Jacobi correctly decoded the stream in 5 iterations. And the Gauss-Seidel method took longer as well, requiring 9 iterations to correctly decode the stream.

The third example of decoding is with the output stream y=11100011 to the input stream x=11001. The Jacobi method correctly decoded y in 5 iterations. Again the Gauss-Seidel takes longer using 9 iterations.

```
public static Matrix jacobiIteration(Matrix a, Matrix b) {
	Matrix s = new Matrix(a.getRowDimension(), a.getColumnDimension());
	for (int i = 0; i < a.getRowDimension(); i++) {
		s.set(i, i, a.get(i, i));
	}
	Matrix t = s.minus(a);
	Matrix x = new Matrix(a.getRowDimension(), 1, 0);
	Matrix xPlus = (Matrix) x.clone();
	for (int i = 0; i < ITERATIONS; i++) {
		xPlus = t.times(x).plus(b);
		for (int j = 0; j < xPlus.getRowDimension(); j++) {
			xPlus.set(j, 0, xPlus.get(j, 0) / s.get(j, j));
		}
		x = (Matrix) xPlus.clone();
	}
	return x;
}

public static Matrix gaussSeidelIteration(Matrix a, Matrix b) {
	Matrix s = new Matrix(a.getRowDimension(), a.getColumnDimension());
	for (int r = 0; r < a.getRowDimension(); r++) {
		for (int c = 0; c <= r; c++) {
			s.set(r, c, a.get(r, c));
		}
	}
	Matrix t = s.minus(a);
	Matrix x = new Matrix(a.getRowDimension(), 1, 0);
	Matrix xPlus = (Matrix) x.clone();
	for (int i = 0; i < ITERATIONS; i++) {
		Matrix rightSide = t.times(x).plus(b);
		for (int j = 0; j < t.getColumnDimension(); j++) {
			double temp = 0.0;
			for (int k = 0; k < j; k++)
				temp += s.get(j, k) * xPlus.get(k, 0);
			xPlus.set(j, 0, (rightSide.get(j, 0) - temp) / s.get(j, j));
		}
		x = (Matrix) xPlus.clone();
	}
	return x;
}
```

Question 3 - Comparison of Decoding

A pattern was found by comparing the iterations of the Jacobi and Gauss-Seidel methods used to decode bit streams. They each took a different amount of iterations to solve different streams but the Jacobi method was more efficient. It was consistently able to decode the bit stream y to the input stream x in less iterations then the Gauss-Seidel method used to decode that same stream.

Part IV
Convergence of Power Method

In part four, the program was to calculate the eigenvalues of at least 1,000 2×2 matrices with random components distributed in a given interval (i.e. the interval [-2, 2]). The Power method and the Inverse Power method were the methods used to calculate the dominant and recessive eigenvalues respectively of the given matrices with an accuracy of at least five digits. The number of iterations will be recorded for each matrix using both the Power and Inverse Power method to and be plotted in two graphs for the corresponding matrix. In each graph, the X-Axis will be the determinant of the matrix and the Y-Axis will be the trace of the matrix. The number of iterations for each method is represented by the color of the point in the corresponding graph.

Power Method

The Power method uses an initial approximation eigenvector to calculate the dominant eigenvector. The code executes iteratively multiplying the given matrix by the approximation eigenvector. The product of that execution is the new eigenvector approximation. Using the Rayleigh Quotient (seen below), the eigenvalue can be calculated from the eigenvector.

$$\lambda = \frac{Ax \cdot x}{x \cdot x}$$

Once the relative error (equation seen below) of the eigenvalue is calculated, the loop is repeated until the error reaches desired amount of error or the maximum number of iterations is reached (whichever comes first).

$$\left| \frac{\lambda^{(i+1)} - \lambda^{i}}{\lambda^{(i+1)}} \right| \times 100 \leq \varepsilon$$

Code for Power Method

```
public static double[] powerMethod(Matrix matrix, int desiredAccuracy) {
       Matrix approximation = new Matrix(new double[][]{{1},{1}});

       int iterations = 0;

       double relativeError = Double.MAX_VALUE;
       double previousEigenvalue = 0;
       double eigenvalue = 0;

       while (relativeError * 100 > (0.5 * Math.pow(10,2 - desiredAccuracy)) &&
iterations < POWER_METHOD_MAX_ITERATIONS)
       {
            //Approximate!
            approximation = matrix.times(approximation);

            //Use's the Rayleigh Equation to solve for the eigenvalue
            eigenvalue = rayleighEquation(matrix, approximation);

            //Calculate the accuracy
            relativeError = Math.abs((eigenvalue - previousEigenvalue) / eigenvalue);
            previousEigenvalue = eigenvalue;

            iterations++;
      }
      return new double[]{eigenvalue, iterations};
}
```

Code for Rayleigh Quotient

```
public static double rayleighEquation(Matrix matrix, Matrix eigenvector) {
      Matrix rayleighNominator = (matrix.times(eigenvector)).transpose().times
(eigenvector);
      Matrix rayleighDenominator = eigenvector.transpose().times(eigenvector);

      return rayleighNominator.getArray()[0][0] / rayleighDenominator.getArray()[0]
[0];
}
```

Inverse Power Method

The Inverse Power method is similar to the process through which the Power method calculates the dominant eigenvalue, though it is different by multiplying the inverse of the given matrix by the approximation eigenvector (instead of simply multiplying the given matrix). As before, the recessive eigenvalue is calculated from the Rayleigh Equation and the loop is repeated until the relative error reaches the desired amount of error or the maximum number of iterations is reached (whichever comes first).

Code for Inverse Power Method

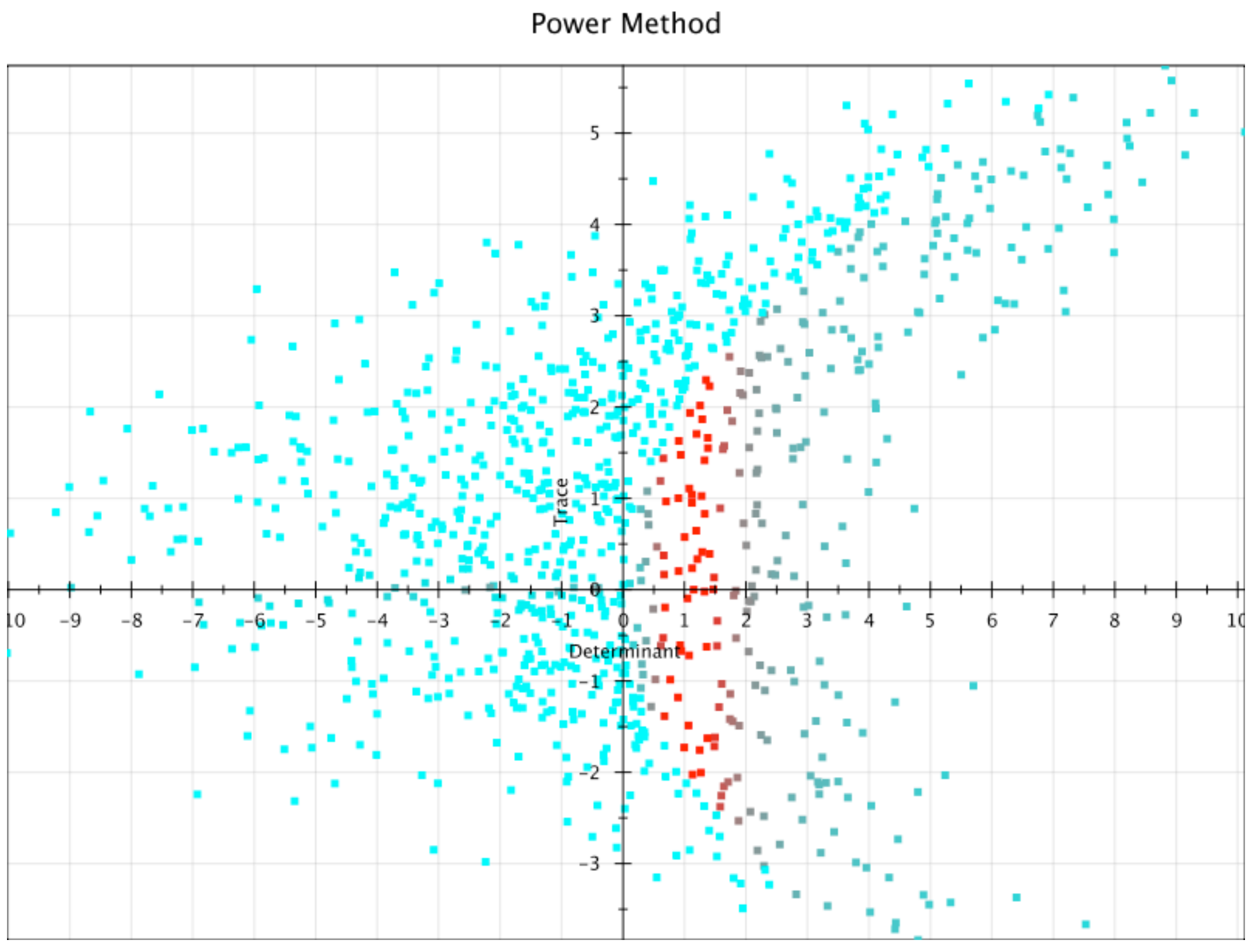
```
public static double[] inversePowerMethod(Matrix matrix, int desiredAccuracy) {
      Matrix approximation = new Matrix(new double[][]{{1},{1}});

      int iterations = 0;

      double relativeError = Double.MAX_VALUE;
      double previousEigenvalue = 0;
      double eigenvalue = 0;

      while (relativeError * 100 > (0.5 * Math.pow(10,2 - desiredAccuracy)) &&
iterations < POWER_METHOD_MAX_ITERATIONS)
      {
            //Approximate!
            approximation = matrix.inverse().times(approximation);

            //Use's the Rayleigh Equation to solve for the eigenvalue
            eigenvalue = rayleighEquation(matrix, approximation);
            //Calculate the accuracy
            relativeError = Math.abs((eigenvalue - previousEigenvalue) / eigenvalue);
            previousEigenvalue = eigenvalue;

            iterations++;
        }
        return new double[]{eigenvalue, iterations};
}
```

Results

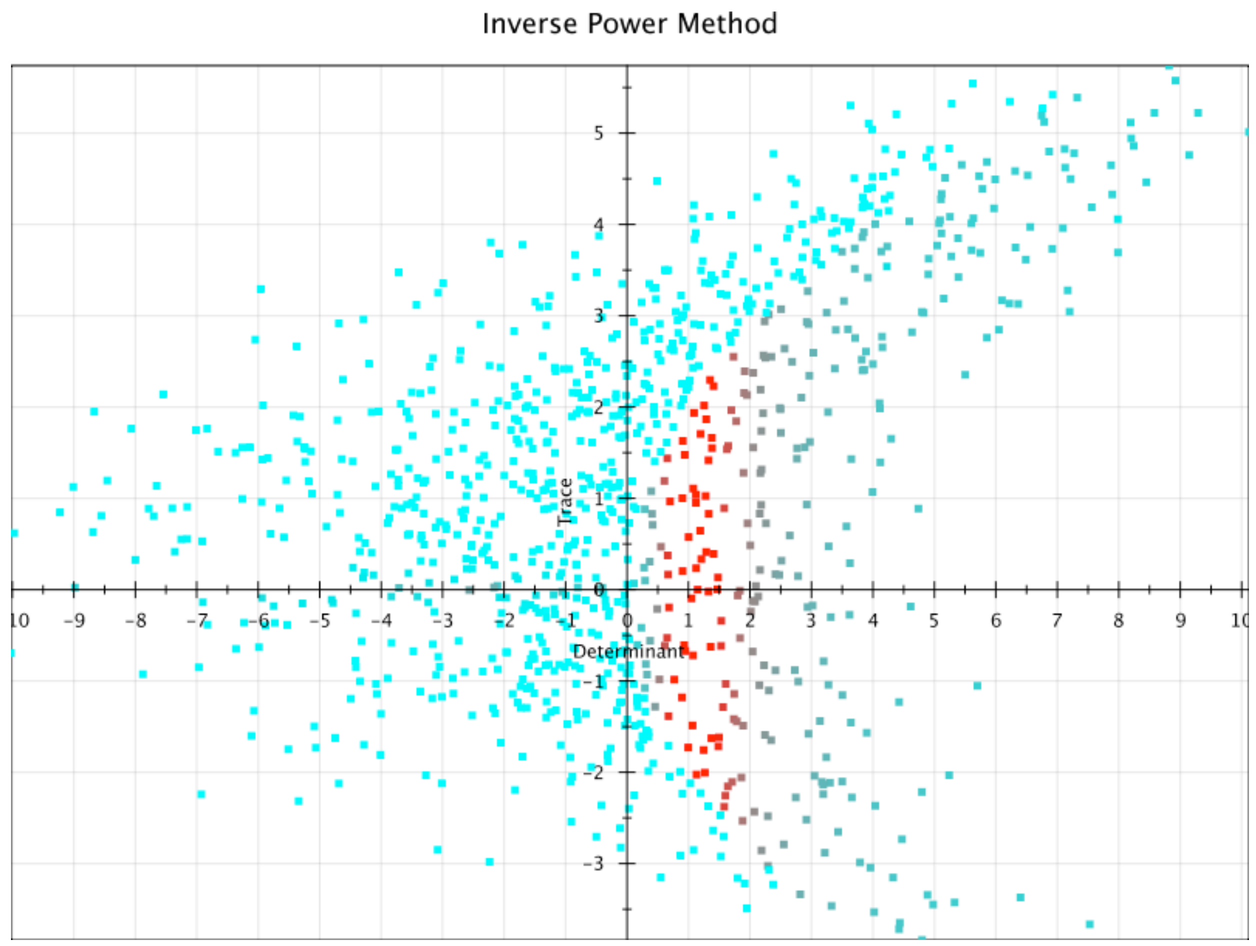
As previously said, each matrix generated was plotted in a determinant-trace graph where the color of each point that represented a matrix was colored based on the number of iterations either the Power method or the Inverse Power method took to calculate the eigenvalues to within a desired accuracy.

The two determinant-trace graphs can be seen below for both the Power method and the Inverse Power method. For the plots, 1,000 matrices were generated with values in an interval of [-2, 2] and either of the power methods ran for a maximum of 2,000 iterations or until the eigenvalues reached an accuracy of six decimal places. The points which are red have a higher number of iterations than the points which are blue.

Power Method Graph



Inverse Power Method Graph



As seen in the two plots above, the two graphs are very similar. In both graphs, a large number of large iteration matrixes exist where the determinant of the matrix is between zero and two and where the trace of the matrix is ranging from around -2.5 and 2.5. The difference between the two graphs is very slight, but can be seen in slight color changes of the number of iterations.

Both graphs feature a similar parabolic shape around the X-Axis. In relation to eigenvalues, the trace of a matrix is the sum of the matrix's eigenvalues and the determinant of a matrix is the product of the matrix's eigenvalues. The graphs show that when the determinant gets more negative the trace approaches zero, which makes sense due to what trace and the determinant of a matrix are relative to the eigenvalues. The relation can be seen again as the determinant gets more positive and as the trace diverges away from zero. The number of iterations can be seen increasing rapidly from the vertex of the parabolic-like grouping as the determinant increases, though once the determinant reaches two, the number of iterations decreases, but is still significantly higher than the rest of the graph.

Acknowledgements

For our java code we used a library which gave us a basic matrix class. This class provided us with methods to perform basic matrix operations such as times, minus, and divide. The library was created as a product of *The MathWorks and the National Institute of Standards and Technology* (NIST) which has been released to the public domain. The library can be found at http://math.nist.gov/javanumerics/jama/.