Text

Description automatically generated

HW2

Group Members: Jason Carlson, Alex Reust, Kevin de Szendeffy

Here we use scipy QR decomposition and test its runtime against n by n matrices:

Text

Description automatically generated

Histogram

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Here we test it against n by 2 matrices, which shows that if the number of columns is small relative to the number of rows, then the time complexity is not in fact cubic:

Graphical user interface

Description automatically generated

Lastly, here is an implementation of the QR decomposition algorithm Jason made in C++. There is also a graph showing how it performs against n by n matrices ranging from 1 by 1 to 400 by 400:

*/\**

*Description: The Following code implements basic QR decomposition and solves Xb = y*

*for b s.t. b minimizes d(Xb, y) where d is the euclidean metric.*

*Author: Jason Carlson*

*\*/*

*#include* <iostream>

*#include* <vector>

*#include* "math.h"

*#include* <time.h>

*#include* <fstream>

*using* *namespace* std;

*void* *print\_matrix*(vector<vector<*double*>>*&* matrix){

*for*(*int* *i=*0; *i* *<* matrix.*size*(); *++i*){

*for*(*int* *j=*0; *j* *<* matrix*[*0*]*.*size*(); *++j*){

*cout* *<<* matrix*[i][j]* *<<* " ";

        }

*cout* *<<* "\n";

    }

*cout* *<<*"\n";

}

*//Uses gramn-schmidt to get the Q matrix for a matrix X with n rows and m columns*

*//O(max(n,m)^3) time complexity.*

vector<vector<*double*>> *get\_Q*(vector<vector<*double*>>*&* X, *int* n, *int* m){

    vector*<*vector*<double>>* *Q*(n, *vector*<*double*>(m, 0));

*for*(*int* *col=*0; *col* *<* m; *++col*){

*//We first project the col'th column of X onto the span of the first j-1 columns of Q:*

        vector*<double>* *projection*(n, 0);

*for*(*int* *j=*0; *j* *<* *col*; *++j*){

*double* *inner\_product* *=* 0;

*for*(*int* *k=*0; *k* *<* n; *++k*){

*inner\_product* *+=* *Q[k][j]\**X*[k][col]*;

            }

*for*(*int* *k=*0; *k* *<* n; *++k*){

*projection[k]* *+=* *inner\_product\*Q[k][j]*;

            }

        }

*//Now we compute the difference vector between the projection and the actual column vector*

*//Note: total is used for storing the norm of the column*

*double* *total* *=* 0;

*for*(*int* *row=*0; *row* *<* n; *++row*){

*Q[row][col]* *=* X*[row][col]* *-* *projection[row]*;

*total* *+=* *Q[row][col]\*Q[row][col]*;

        }

*//Here we normalize the column added to Q*

*for*(*int* *row=*0; *row* *<* n; *++row*){

*Q[row][col]* *=* *Q[row][col]* */* *sqrt*(*total*);

        }

    }

*return* *Q*;

}

*//O(n^2) time complexity*

vector<vector<*double*>> *transpose*(vector<vector<*double*>>*&* Q, *int* n, *int* m){

    vector*<*vector*<double>>* *QT*(m, *vector*<*double*>(n, 0));

*for*(*int* *i=*0; *i* *<* m; *++i*){

*for*(*int* *j=*0; *j* *<* n; *++j*){

*QT[i][j]* *=* Q*[j][i]*;

        }

    }

*return* *QT*;

}

*//Assumes A and B are nonempty matrices in O(n^3) time complexity. Assumes the # of columns of A*

*//is the same as the number of rows of B.*

vector<vector<*double*>> *mult*(vector<vector<*double*>>*&* A, vector<vector<*double*>>*&* B){

    vector*<*vector*<double>>* *answer*(A.*size*(), *vector*<*double*>(B*[*0*]*.*size*()));

*for*(*int* *row=*0; *row* *<* A.*size*(); *++row*){

*for*(*int* *col=*0; *col* *<* B*[*0*]*.*size*(); *++col*){

*//inner product of A[row] with B[][col]*

*double* *inner\_product* *=* 0;

*for*(*int* *i=*0; *i* *<* A*[*0*]*.*size*(); *++i*){

*inner\_product* *+=* A*[row][i]\**B*[i][col]*;

            }

*answer[row][col]* *=* *inner\_product*;

        }

    }

*return* *answer*;

}

*//O(n^3) time complexity because of matrix multiplication (theoretically one can get it down to O(n^(2.3)) using Coppersmith-Winograd)*

vector<vector<*double*>> *get\_R*(vector<vector<*double*>>*&* Q, vector<vector<*double*>>*&* X, *int* n, *int* m){

*//R = Q^(T) X*

    vector*<*vector*<double>>* *QT* *=* *transpose*(Q, n, m);

*return* *mult*(*QT*, X);

}

*//decomposes a matrix X into its QR decomposition. n is the number of rows and n is the number of columns.*

pair<vector<vector<*double*>>, vector<vector<*double*>>> *decompose*(vector<vector<*double*>>*&* X, *int* n, *int* m){

    vector*<*vector*<double>>* *Q* *=* *get\_Q*(X, n, m);

    vector*<*vector*<double>>* *R* *=* *get\_R*(*Q*, X, n, m);

*return* *make\_pair*(*Q*, *R*);

}

vector<*double*> *get\_beta*(vector<vector<*double*>>*&* R, vector<vector<*double*>> QT, vector<vector<*double*>>*&* y){

*//Rb = Q^(T)y := A*

    vector*<double>* *beta*(R.*size*(), 0);

    vector*<*vector*<double>>* *A* *=* *mult*(QT, y);

*for*(*int* *row=*y.*size*()*-*1; *row* *>=* 0; *row--*){

*if*(*row* *==* R.*size*()*-*1){

*beta[row]* *=* *A[row][*0*]* */* R*[row][row]*;*//note: A is a column vector*

        }*else*{

*double* *prev\_sum* *=* 0;

*for*(*int* *col=*y.*size*()*-*1; *col* *>* *row*; *col--*){

*prev\_sum* *+=* R*[row][col]\*beta[col]*;

            }

*beta[row]* *=* (*A[row][*0*]* *-* *prev\_sum*) */* R*[row][row]*;

        }

    }

*return* *beta*;

}

*//Solves the least squares problem*

*int* *main*(){

*//Test 1----------------------------------------------------------*

    vector*<*vector*<double>>* *X* *=* {

        {2, 0, 0},

        {0, 3, 0},

        {0, 0, 4}

    };

    pair*<*vector*<*vector*<double>>*, vector*<*vector*<double>>>* *test* *=* *decompose*(*X*, 3, 3);

*cout* *<<* "Test Matrix 1: \n";

*print\_matrix*(*X*);

*cout* *<<* "Q: \n";

*print\_matrix*(*test*.*first*);

*cout* *<<* "R: \n";

*print\_matrix*(*test*.*second*);

*//Test 2----------------------------------------------------------*

*X* *=* {

        {2, 1, 7},

        {3.4, 3, 24.5},

        {1.23443, 2.543, 4}

    };

*test* *=* *decompose*(*X*, 3, 3);

*cout* *<<* "Test Matrix 2: \n";

*print\_matrix*(*X*);

*cout* *<<* "Q: \n";

*print\_matrix*(*test*.*first*);

*cout* *<<* "R: \n";

*print\_matrix*(*test*.*second*);

*cout* *<<* "Product of Q with R: \n";

    vector*<*vector*<double>>* *product* *=* *mult*(*test*.*first*, *test*.*second*);

*print\_matrix*(*product*);*//Notice this is the same as the original matrix (up to like 15 decimal places)*

    vector*<*vector*<double>>* *y* *=* {{1},{2},{3}};

    vector*<*vector*<double>>* *Q* *=* *test*.*first*;

    vector*<*vector*<double>>* *R* *=* *test*.*second*;

*//Rb = Q^(T)y*

    vector*<double>* *beta* *=* *get\_beta*(*R*, *transpose*(*Q*, *Q*.*size*(), *Q[*0*]*.*size*()), *y*);

*cout* *<<* "Beta:\n";

*for*(*int* *i=*0; *i* *<* *beta*.*size*(); *++i*){

*cout* *<<* *beta[i]* *<<* " ";

    }

*//The true values are 0.247179, 1.22031, -.102096 (which are the same)*

*//-------------------------------------------------------------------------------------*

*//Here we test the QR decomposition algorithm for various n by n matrices to exhibit O(n^3) time complexity*

*int* *n* *=* 1;

    vector*<double>* *times*;

*for*(; *n* *<=* 400; *n++*){

        vector*<*vector*<double>>* *X2*(*n*, *vector*<*double*>(*n*, 0));

*for*(*int* *i=*0; *i* *<* *n*; *++i*){

*for*(*int* *j=*0; *j* *<* *n*; *++j*){

*X2[i][j]* *=* (*double*) *rand*();

            }

        }

*clock\_t* *t* *=* *clock*();

*test* *=* *decompose*(*X2*, *n*, *n*);

*t* *=* *clock*() *-* *t*;

*times*.*push\_back*(((*float*)*t*)*/CLOCKS\_PER\_SEC*);

*//printf ("It took %.18f seconds. for n = %d\n",((float)t)/CLOCKS\_PER\_SEC, n);*

    }

    ofstream *data*("output.txt");

*for*(*int* *i=*0; *i* *<* *times*.*size*(); *++i*){

*data* *<<* *times[i]* *<<* "\n";

    }

*//See the output.txt file for the exact values*

*return* 0;

}

Chart, line chart

Description automatically generated

It should be noted that the scipy function is faster than the c++ code above. This is most likely due to it calling native c++ code that uses Strassen’s matrix multiplication algorithm, which is a bit better than O(n^3) (it is a conjecture that in theory one could get it down to O(n^(2+epsilon)) for all epsilon > 0).