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:

In [1]:
```
import numpy as np
import scipy.linalg
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import time
```

In [2]:
```
# Best practice source:
# https://numpy.org/doc/stable/reference/random/generated/numpy.random.seed.h
from numpy.random import MT19937
from numpy.random import RandomState, SeedSequence
rs = RandomState(MT19937(SeedSequence(72730)))
```

In [3]:
```
# Generate simulated data
# Must have rows > cols
rows=100
cols = 10
sigma_true = .5
x = np.random.normal(size=(rows, cols))
beta_true = np.multiply(np.random.normal(size=cols), np.random.uniform(-6,6,co
y = np.matmul(x, beta_true) + sigma_true*np.random.normal(size=rows)
```

In [4]:
```
# Use scipy qr decomposition to solve for betas
decomp = scipy.linalg.qr(x)
print("Dimensions of Q: ", decomp[0].shape)
print("Dimensions of R: ", decomp[1].shape)
```

```
Dimensions of Q:  (100, 100)
Dimensions of R:  (100, 10)
```

In [5]:
```
# Verify that Q is orthogonal
print("Q'Q: ", np.matmul(decomp[0], decomp[0].T).round(10))
print("Det of Q: ", scipy.linalg.det(decomp[0]))
```

```
Q'Q:  [[ 1. -0.  0. ... -0.  0. -0.]
 [-0.  1. -0. ... -0.  0. -0.]
 [ 0. -0.  1. ... -0.  0. -0.]
 ...
 [-0. -0. -0. ...  1. -0.  0.]
 [ 0.  0.  0. ... -0.  1.  0.]
 [-0. -0. -0. ...  0.  0.  1.]]
Det of Q:  1.0000000000000029
```

In [6]:
```
# Verify that we can recover X
np.allclose(np.dot(decomp[0], decomp[1]), x)
```

Out[6]: True

In [7]:

```
# Solve for beta using R%*%Beta = Q^t%*%y
qty = np.matmul(decomp[0].T, y.reshape(-1,1))
# Need to use the linearly indepdentent part of R, corresponding QtY
beta_hat = scipy.linalg.solve_triangular(decomp[1][:cols, :cols], qty[:cols])
```

In [8]:

```
# Verify that we're close to the true beta
print("Difference in Beta Hat vs Beta True:")
np.subtract(beta_hat, np.reshape(beta_true, (-1,1)))
```

```
Difference in Beta Hat vs Beta True:
```

Out[8]:

```
array([[ 0.03062098],
       [-0.01686938],
       [-0.00668422],
       [ 0.10214225],
       [-0.02472997],
       [-0.03930336],
       [ 0.06000058],
       [-0.0130657 ],
       [-0.00876292],
       [-0.01727208]])
```

In [9]:

```
# Verify that we're close to the fitted beta from sklearn
lfit = LinearRegression() #L2 reg is 1/C
# need numpy 2dim array for X
# X = x.reshape((n,1)).copy()
lfit.fit(x,y)
print("Differences in sklearn vs backsolve: ")
np.subtract(beta_hat, np.reshape(lfit.coef_, (-1,1)))
# Looking good!
```

```
Differences in sklearn vs backsolve:
```

Out[9]:

```
array([[ 2.50484113e-03],
       [-4.05947403e-03],
       [ 1.86747191e-03],
       [-3.68102959e-04],
       [-1.77282626e-04],
       [ 6.26297593e-04],
       [ 2.80990643e-03],
       [-1.11652546e-03],
       [ 4.49906040e-05],
       [ 8.06336569e-04]])
```

In [10]:

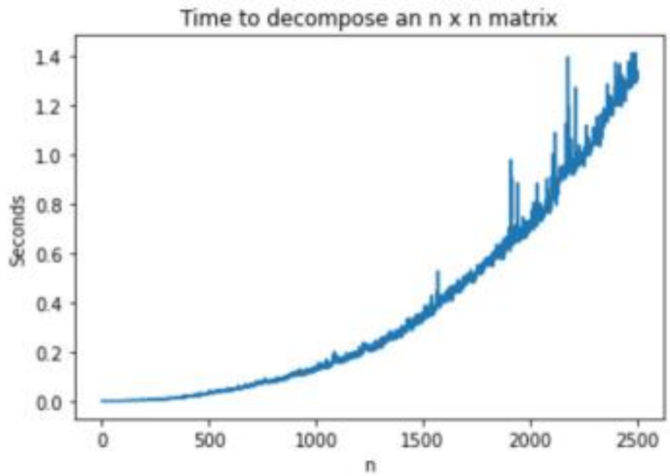
```
# Time the qr decomp as the size of a square matrix increases
row_space = np.linspace(1, 2500, 2500, dtype=int)
outputs = np.zeros(row_space.shape[0])
for i in range(row_space.shape[0]):
    n = row_space[i]
    data = np.random.normal(size=(n,n))
    start = time.time()
    tmp = scipy.linalg.qr(data)
    outputs[i] = time.time() - start
```

In [11]:

```
plt.plot(row_space, outputs)
plt.xlabel("n")
plt.ylabel("Seconds")
plt.title("Time to decompose an n x n matrix")
```

Out[11]:

```
Text(0.5, 1.0, 'Time to decompose an n x n matrix')
```



In [ ]:

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:

In [83]:
```
#####################################################
### imports
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn; seaborn.set()

import scipy as sp
import scipy.stats as sps

import math
from time import time

from sklearn.linear_model import LinearRegression
import statsmodels.api as sm

import itertools
```

QR Computation of Least Squares β

In [86]:
```
X = np.random.randn(1000, 2)
```

In [87]:
```
y = np.dot(X, np.array([0.5, 1]))
```

In [88]:
```
Q, R = np.linalg.qr(X)
```

In [89]:
```
np.dot(np.linalg.inv(R), np.dot(Q.T, y))
```

Out[89]:
```
array([0.5, 1. ])
```

In [90]:
```
linreg = LinearRegression().fit(X, y)
```

In [91]:
```
linreg.coef_
```

Out[91]:
```
array([0.5, 1. ])
```

Timing QR Decomposition

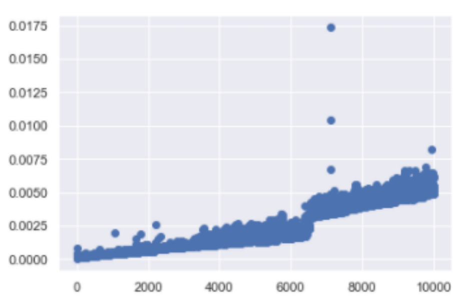
In [114]:
```
trials = 10
```

In [115]:
```
t = np.zeros(10000)
```

In [116]:
```
for i in range(len(t)):
    X = np.random.randn(10*i, 2)
    t1 = time()
    for j in range(trials):
        Q, R = np.linalg.qr(X)
    t2 = time()
    t[i] = (t2-t1)/trials
```

In [117]:
```
plt.scatter(range(len(t)),t)
```

Out[117]:
```
<matplotlib.collections.PathCollection at 0x11adedb2880>
```



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 v
ector
        //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 o
f 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 numbe
r 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<d
ouble>>& 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 de
cimal 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) t
ime 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();
```

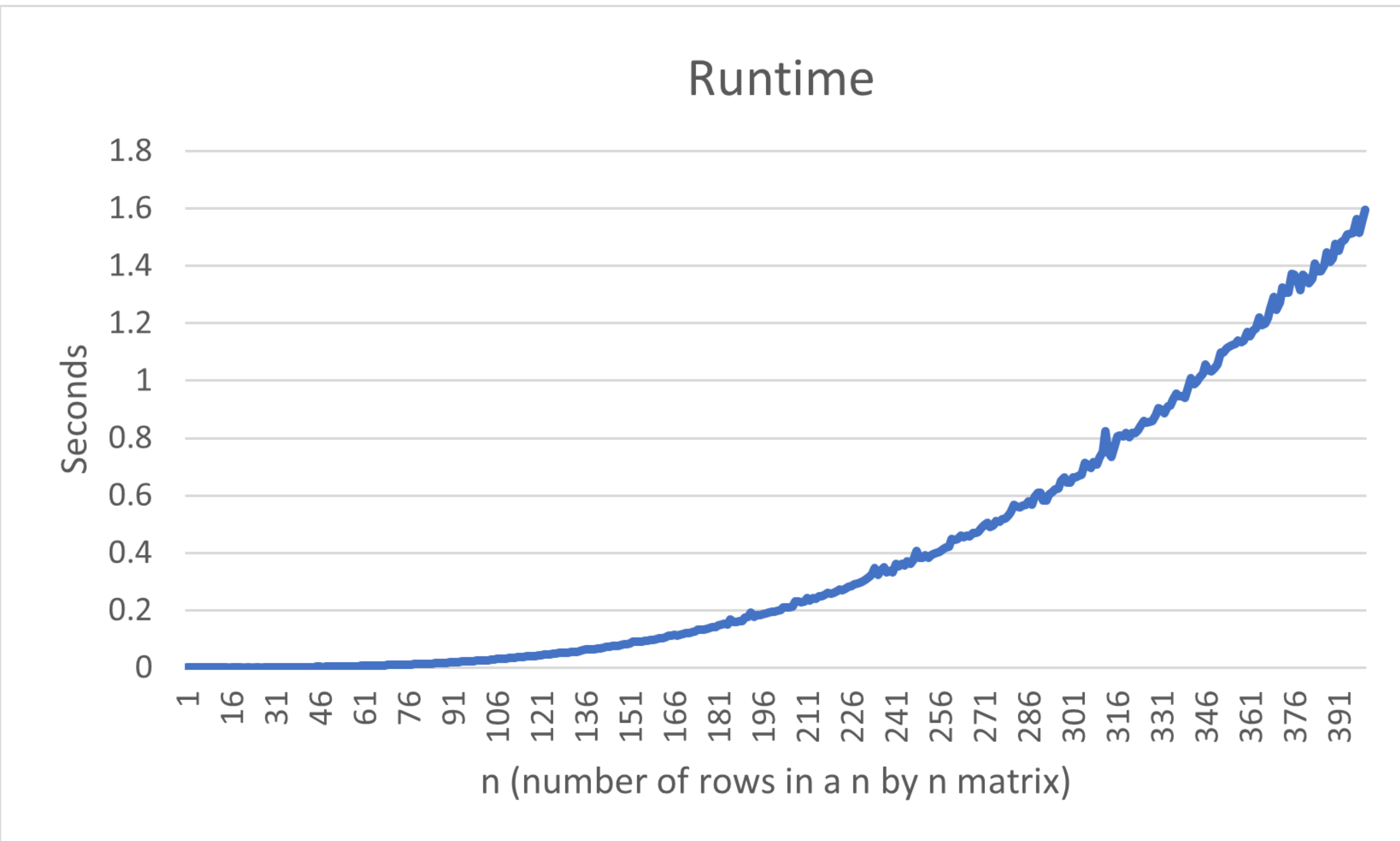
```
            }
        }
        clock_t t = clock();
        test = decompose(X2, n, n);
        t = clock() - t;
        times.push_back(((float)t)/CLOCKS_PER_SEC);
        //printf ("It took %.18f seconds. for n = %d\n",((float)t)/CLOCKS_PER_SEC, n);
    }

    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;
}
```



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).