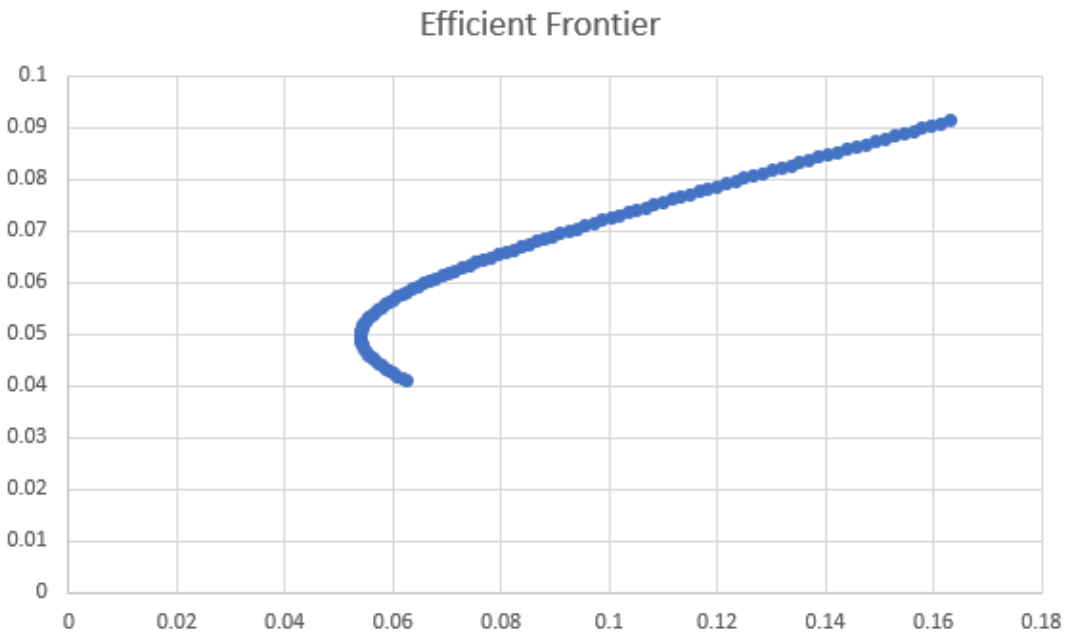
MFE C++ Assignment 6
Xin Zeng

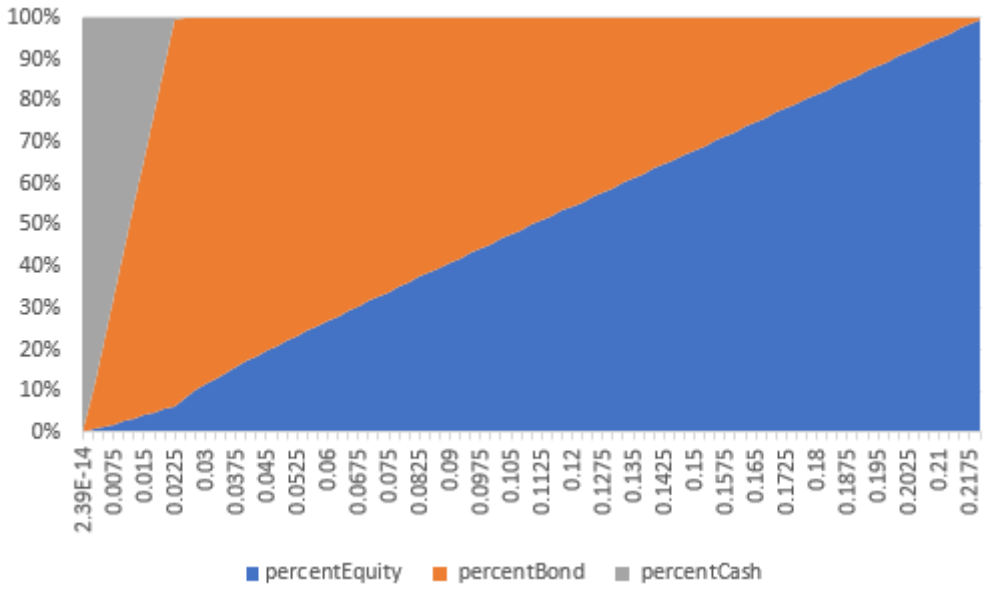
Question 1

Efficient Frontier Plot



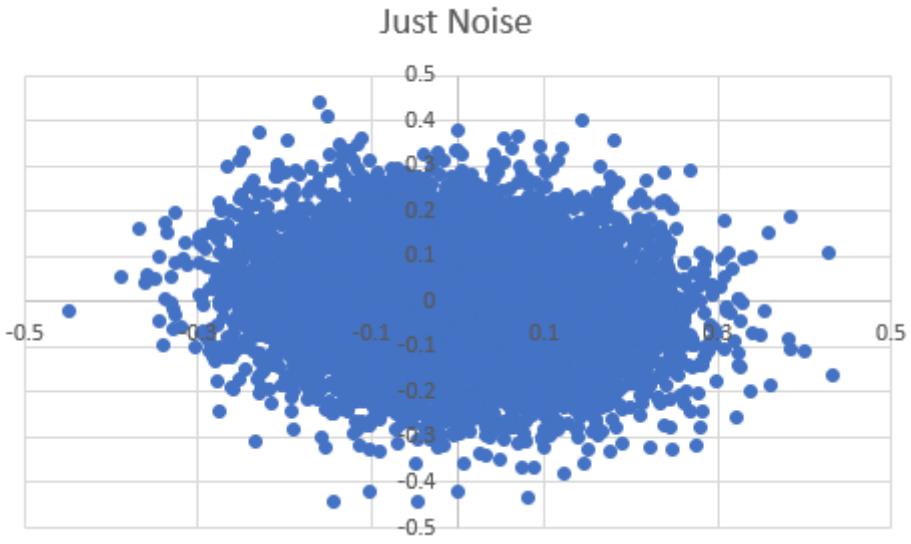
Return by Risk Plot

Volatility Plot

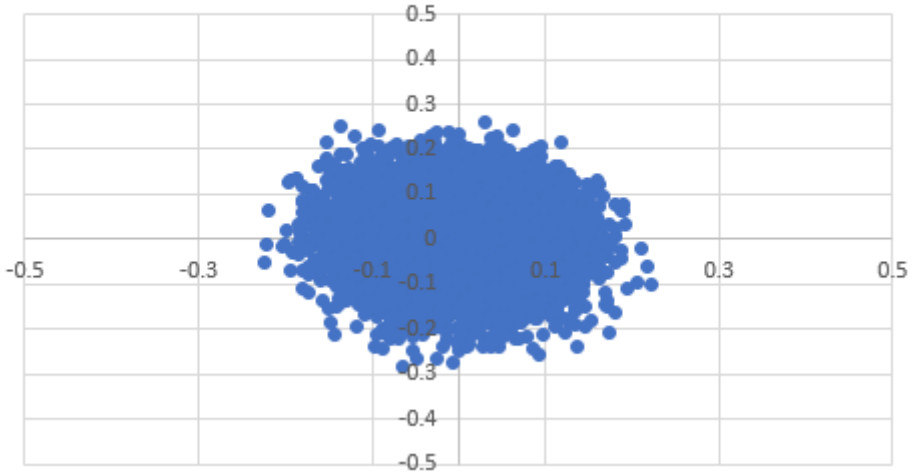


Question 2

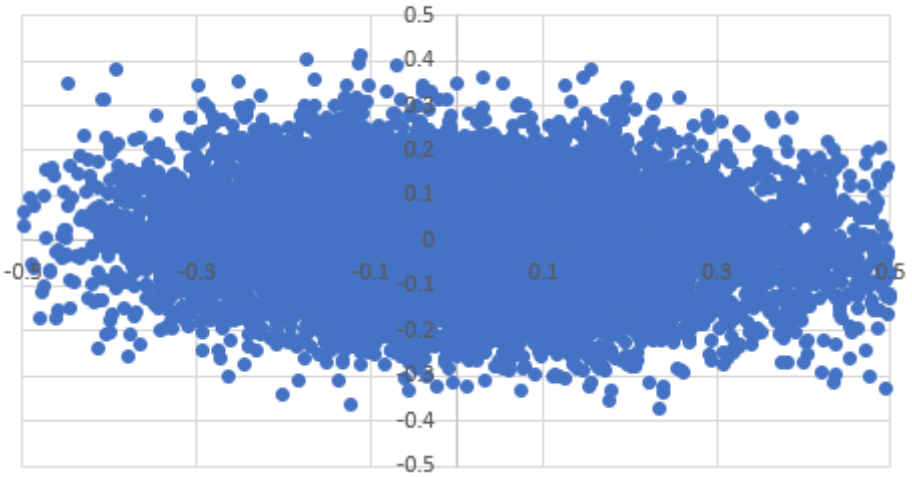
Scatter plots



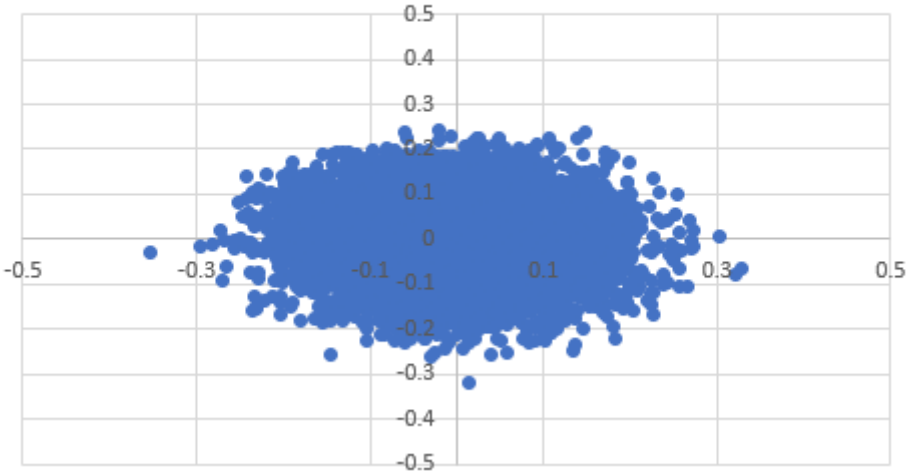
Noise and Value



Noise and Momentum



All Agents



1. Noise and Value as the least volatile plot. Since value agent sells when its index is positive and buys when its index is negative. Compared with random walk, this would reduce the volatility.
2. Noise and Momentum has the most volatile plot. Since momentum agent sells when its index is less than the previously observed index and buys when index is more than the previously observed index. Compared with random walk, this would increase the volatility.
3. Value doesn't dominate momentum. Momentum doesn't dominate value. There is no obvious difference between asset 1 and asset2.

Code

Question 1

```
//Assignment 6 Project 1
//Xin Zeng

#include <iostream>
#include <fstream>
#include <vector>
#include <time.h>
#include <random>

using namespace std;

class Portfolio {
public:
	double percentEquity; // equity percentage of portfolio
	double percentBond; // bond percentage of portfolio
	double ret; // portfolio annual expected return
	double vol; // portfolio annual expected volatility
	double percentCash;
	Portfolio(double percentEquity1, double percentBond1, double ret1, double vol1) {
		percentEquity = percentEquity1;
		percentBond = percentBond1;
		ret = ret1;
		vol = vol1;
		percentCash = 1 - percentEquity1 - percentBond1;
	};
};


class Market {
public:
	double corrEquityBond; // bond-equity correlation
	double equityReturn; // equity annual return
	double equityVol; // equity annual volatility
	double bondReturn; // bond annual return
	double bondVol; // bond annual volatility
	double riskFreeRate; // risk free rate
	int numSimulations; // number of simulations to run
	vector<double> equitySimReturns;
	vector<double> bondSimReturns;

	Market(double corrEquityBond1, double equityReturn1, double equityVol1, double bondReturn1,
double bondVol1,
		double riskFreeRate1, int numSimulations1) {
		corrEquityBond = corrEquityBond1;
		equityReturn = equityReturn1;
		equityVol = equityVol1;
		bondReturn = bondReturn1;
		bondVol = bondVol1;
		riskFreeRate = riskFreeRate1;
```

```
        numSimulations = numSimulations1;

        mt19937 generator1((long unsigned int)time(0));
        mt19937 generator2((long unsigned int)time(0) + 100);
        normal_distribution<double> distribution(0.0, 1.0);
        double dt = 1.0;
        double X1, X2, Z1, Z2;
        for (int i = 0; i < numSimulations; i++) {
            X1 = distribution(generator1);
            X2 = distribution(generator2);
            Z1 = X1;
            Z2 = corrEquityBond1 * X1 + sqrt(1 - corrEquityBond1 * corrEquityBond1) * X2;
            bondSimReturns.push_back((bondReturn1 - 0.5 * bondVol1 * bondVol1) * dt + bondVol1
* sqrt(dt) * Z1);
            equitySimReturns.push_back((equityReturn1 - 0.5 * equityVol1 * equityVol1) * dt +
equityVol1 * sqrt(dt) * Z2);
        }
    }

    Portfolio analyzePortfolio(double percentEquity, double percentBond) {
        double return_sum = 0.0;
        double return_variance = 0.0;
        vector<double> return_portfolio(numSimulations);
        for (int i = 0; i < numSimulations; i++) {
            double return_sigle = -1.0 + percentEquity * exp(equitySimReturns[i]) + percentBond *
exp(bondSimReturns[i]) +
                (1.0 - percentEquity - percentBond) * exp(riskFreeRate);
            return_portfolio[i] = return_sigle;
            return_sum += return_sigle;
        }
        double mean = return_sum / numSimulations;
        for (int i = 0; i < numSimulations; i++) {
            return_variance += pow(return_portfolio[i] - mean, 2);
        }
        double sd = sqrt(return_variance / numSimulations);
        return      Portfolio(percentEquity, percentBond, mean, sd);
    }

    double calculateTangent(double percentEquity) {
        Portfolio equity_up = analyzePortfolio(percentEquity + 0.01, 0.99 - percentEquity);
        Portfolio equity_down = analyzePortfolio(percentEquity - 0.01, 1.01 - percentEquity);
        return (equity_up.ret - equity_down.ret) / (equity_up.vol - equity_down.vol);
    }

    Portfolio findTangencyPortfolio() {
        double max_slope, slope_temp;
        Portfolio TangencyPortfolio(0, 0, 0, 0);
        max_slope = 0.0;
        for (double i = 0.01; i <= 0.99; i += 0.0025) {
            Portfolio P_temp = analyzePortfolio(i, 1 - i);
            slope_temp = (P_temp.ret - riskFreeRate) / P_temp.vol;
            if (slope_temp > max_slope) {
                max_slope = slope_temp;
                TangencyPortfolio = P_temp;
            }

        }
        return TangencyPortfolio;
    }

    const double ep = 1e-10;
    bool iszero(double x) {
        return fabs(x) < ep;
```

```
    }

    Portfolio findPortfolioForVol(double vol) {
        Portfolio TangencyPortfolio = findTangencyPortfolio();
        if (vol < TangencyPortfolio.vol) {
            double CashPercent = 1 - vol / TangencyPortfolio.vol;
            double EquityPercent = (1 - CashPercent) * TangencyPortfolio.percentEquity;
            double BondPercent = (1 - CashPercent) * TangencyPortfolio.percentBond;
            return analyzePortfolio(EquityPercent, BondPercent);
        }
        else {
            double x0 = TangencyPortfolio.percentEquity;
            double x1 = 1.0;
            double x2 = (x0 + x1) / 2;
            Portfolio P_mid = analyzePortfolio(x2, 1 - x2);
            while (!iszero(P_mid.vol - vol)) {
                if (P_mid.vol - vol > 0) {
                    x1 = x2;
                }
                else {
                    x0 = x2;
                }
                x2 = (x0 + x1) / 2;
                P_mid = analyzePortfolio(x2, 1 - x2);
            }
            return analyzePortfolio(x2, 1 - x2);
        }
    }
};


int main() {
    Market Market1(-0.2, 0.088, 0.15, 0.04, 0.06, 0.02, 100000);
    ofstream outfile;
    outfile.open("C:\\Users\\Sandraw Zeng\\Desktop\\Berkeley PreMFE
Courses\\C++\\Assignment6\\Efficient_Frontier.csv");
    outfile << "percentEquity, percentBond, portfolioReturn,portfolioVol" << endl;
    for (double i = 0; i <= 1; i += 0.01) {
        Portfolio P = Market1.analyzePortfolio(i, 1.0 - i);
        outfile << i << ',' << 1 - i << ',' << P.ret << ',' << P.vol << endl;
    }
    outfile.close();

    Market Market2(-0.1, 0.09, 0.20, 0.03, 0.02, 0.02, 100000);
    ofstream outfile2;
    double maxvol = Market2.analyzePortfolio(1, 0).vol;
    outfile2.open("C:\\Users\\Sandraw Zeng\\Desktop\\Berkeley PreMFE
Courses\\C++\\Assignment6\\Tangency_Portfolio.csv");
    outfile2 << "percentEquity, percentBond, percentCash, portfolioReturn, portfolioVol" << endl;
    for (double i = 0; i <= maxvol; i += 0.0025) {
        Portfolio P = Market2.findPortfolioForVol(i);
        outfile2 << P.percentEquity << ',' << P.percentBond << ',' << P.percentCash << ',' << P.ret << ','
<< P.vol << endl;
    }
    outfile2.close();
    return 0;
}
```

Question 2

agents.h

```cpp
#pragma once

#include <vector>
#include <random>
#include <chrono>
using namespace std;

class Dealer; // we need to let the compiler know that Dealer is a class

class Agent {
public:
	// the probability the agent trades during a period
	double tradeProb;
	// a scaling factor for the size of the trade
	double tradeScale;
	// how much an agent trades of each security
	double securityScale1;
	double securityScale2;
public:
	Agent(double tradeProb, double tradeScale, double securityScale1, double securityScale2):
		tradeProb(tradeProb), tradeScale(tradeScale), securityScale1(securityScale1),
securityScale2(securityScale2) {};
	virtual ~Agent();
	virtual void tick(double price1, double price2, Dealer* dealer) = 0;
	virtual void reset() {};
	virtual double getIndexValue(double price1, double price2) {
		return (price1 - 1) * securityScale1 + (price2 - 1) * securityScale2;
	};
};

class Result {
public:
	double Price1;
	double Price2;
public:
	Result() = default;
	Result(double p1, double p2) : Price1(p1), Price2(p2) {}
};

class Dealer {
public:
	vector<Agent*> agents;
	double net1;
	double net2;
	double priceScale1;
	double priceScale2;
private:
	double periodNet1;
	double periodNet2;
public:
	Dealer(double priceScale1, double priceScale2) {
		this -> priceScale1 = priceScale1;
		this -> priceScale2 = priceScale2;
		this -> net1 = 0;
		this -> net2 = 0;
	};
	virtual ~Dealer() {
		agents.clear();
	};
	void addAgent(Agent* a) {
		agents.push_back(a);
	};
	void runSimulation(int numSimulations, int numPeriods, vector< Result >& simResults) {
```

```
            simResults.clear();
            double price1, price2;
            default_random_engine generator;
            uniform_real_distribution<double> uniform(0.0, 1.0);
            for (int i = 0; i < numSimulations; i++) {
                for (int j = 0; j < agents.size(); j++) {
                    agents[j]->reset();
                }
                net1 = 0;
                net2 = 0;
                for (int n = 0; n < numPeriods; n++) {
                    getPrice(price1, price2);
                    periodNet1 = 0;
                    periodNet2 = 0;
                    for (int m = 0; m < agents.size(); m++) {
                        double dis = uniform(generator);
                        if (dis < agents[m]->tradeProb) {
                            agents[m]->tick(price1, price2, this);
                        }
                    }
                    net1 += periodNet1;
                    net2 += periodNet2;
                }
                getPrice(price1, price2);
                Result result(price1, price2);
                simResults.push_back(result);
            };
        }
        void adjustNet(double adjust1, double adjust2) {
            periodNet1 += adjust1;
            periodNet2 += adjust2;
        };
        void getPrice(double& price1, double& price2) {
            price1 = exp(priceScale1 * net1);
            price2 = exp(priceScale2 * net2);
        };
};

class ValueAgent : public Agent {
public:
        ValueAgent(double tradeProb,
            double tradeScale,
            double scale1,
            double scale2
        ) : Agent(tradeProb, tradeScale, scale1, scale2) {}
        void tick(double price1, double price2, Dealer* dealer) {
            double index = getIndexValue(price1, price2);
            double netToTrade = -0.5 + 1 / (1 + exp(tradeScale * index));
            dealer->adjustNet(netToTrade * securityScale1, netToTrade * securityScale2);
        };
};

class MomentumAgent : public Agent {
protected:
        double prevIndex = 0;
public:
        MomentumAgent(double tradeProb,
            double tradeScale,
            double scale1,
            double scale2
        ) : Agent(tradeProb, tradeScale, scale1, scale2) {}
        void tick(double price1, double price2, Dealer* dealer) {
            double index = getIndexValue(price1, price2);
```

```
            double netIndex = index - prevIndex;
            prevIndex = index;
            double netToTrade = -0.5 + 1 / (1 + exp(-tradeScale * netIndex));
            dealer->adjustNet(netToTrade * securityScale1, netToTrade * securityScale2);
        };
        void reset() {
            Agent::reset();
            prevIndex = 0;
        };
};

class NoiseAgent : public Agent {
public:
        NoiseAgent(double tradeScale, double scale1, double scale2) :
            Agent(1.0, tradeScale, scale2, scale1) {}
        void tick(double price1, double price2, Dealer* dealer);
};

Agent::~Agent() {};

default_random_engine generator;
void NoiseAgent::tick(double price1, double price2, Dealer* dealer) {
        normal_distribution<double> normal(0.0, 1.0);
        double netToTrade = normal(generator) * tradeScale;
        dealer->adjustNet(netToTrade * securityScale1, netToTrade * securityScale2);
};
```