Running your first multithread simulation using C++11

Document name: "BDiF – TN0003.pdf"
Date/revision: Sat 14th Dec 2013 / Rev A.
Author: Andrew Sheppard, © 2013.

quantnet.com/bigdata

---

1. Introduction

This technical note is part of the prerequisite learning materials for the "Big Data in Finance" MOOC (BDiF). The prerequisite materials are meant to help get new students up to speed with skills that are necessary to do well in the course. For people new to Big Data the technical notes form a foundation on which to build their skills, while for people already somewhat familiar with the material they are a refresher; in either case, each technical note addresses some basic skill that a successful student needs to know.

The BDiF course puts great emphasis on programs that execute in parallel. Many forms of data analysis follow a divide-and-conquer methodology and the only reasonable way to tackle Big Data is often by working on it in parallel in one way or another.

There are many ways in which to implement parallelism and the BDiF course covers most of them. This technical note covers multithreading on a single processor using the language features available in C++11. Support for parallelism in C++11 (hereafter just C++) is quite strong and that is one of the reasons that you will be using C++ throughout the BDiF course. Another reason for using C++ is because it is very common in quantitative finance and being a good C++ programmer is a valuable skill. This is another area of emphasis in the course. Namely, building practical skills that you can start applying today to Big Data problems (other types of problems too!).

2. Calculating π Using the Monte-Carlo Method

Consider a unit diameter circle inside a unit square, Figure 1.

If we randomly throw darts at the unit square, some will fall inside the circle, and some will fall outside the circle; all will fall within the unit square. The proportion that falls within the circle are related to π by a simple formula. So, if we write a program that simulates throwing darts at a unit square inside which sits an imaginary circle, we can obtain a numerical estimate of π. (Note: there are much better and quicker ways to estimate π than using the Monte-Carlo method. But that need not concern us here as this is more of a programming exercise.)

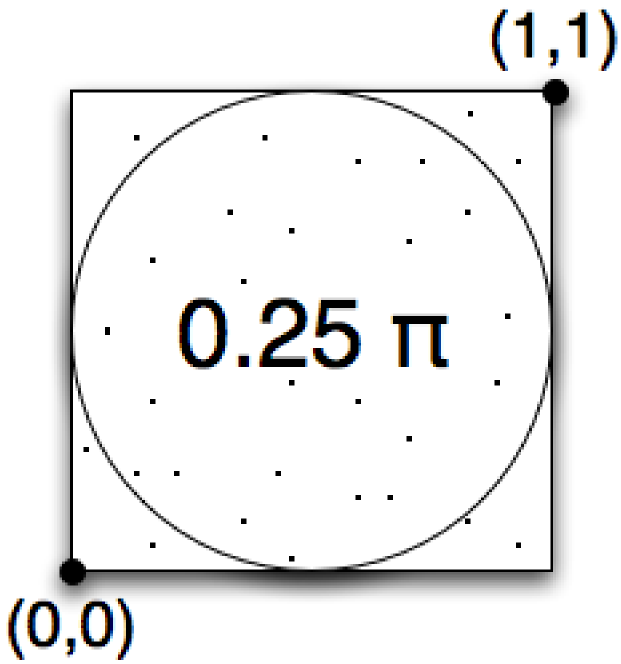

Figure 1. Unit circle inside a unit square.

3. **Running a Serial Simulation**

Here are step-by-step instructions to compiling, running and looking at the output of your first simulation program:

1. Open a web browser and type in the address http://coliru.stacked-crooked.com/ . This will open the window shown in Figure 2.
2. Cut & paste the code from **Appendix I** into the main panel.
3. Then click on the button labeled "Compile, link and run …". This will run the program and the output will appear in the panel directly below the code.
4. That's it! You've run your first simulation in C++.

Alternatively, follow these steps:

1. Click on this link: load code in browser.
2. Click on the button labeled "Edit".
3. Then click on the button labeled "Compile, link and run …".
4. That's it! You've run your first simulation in C++.

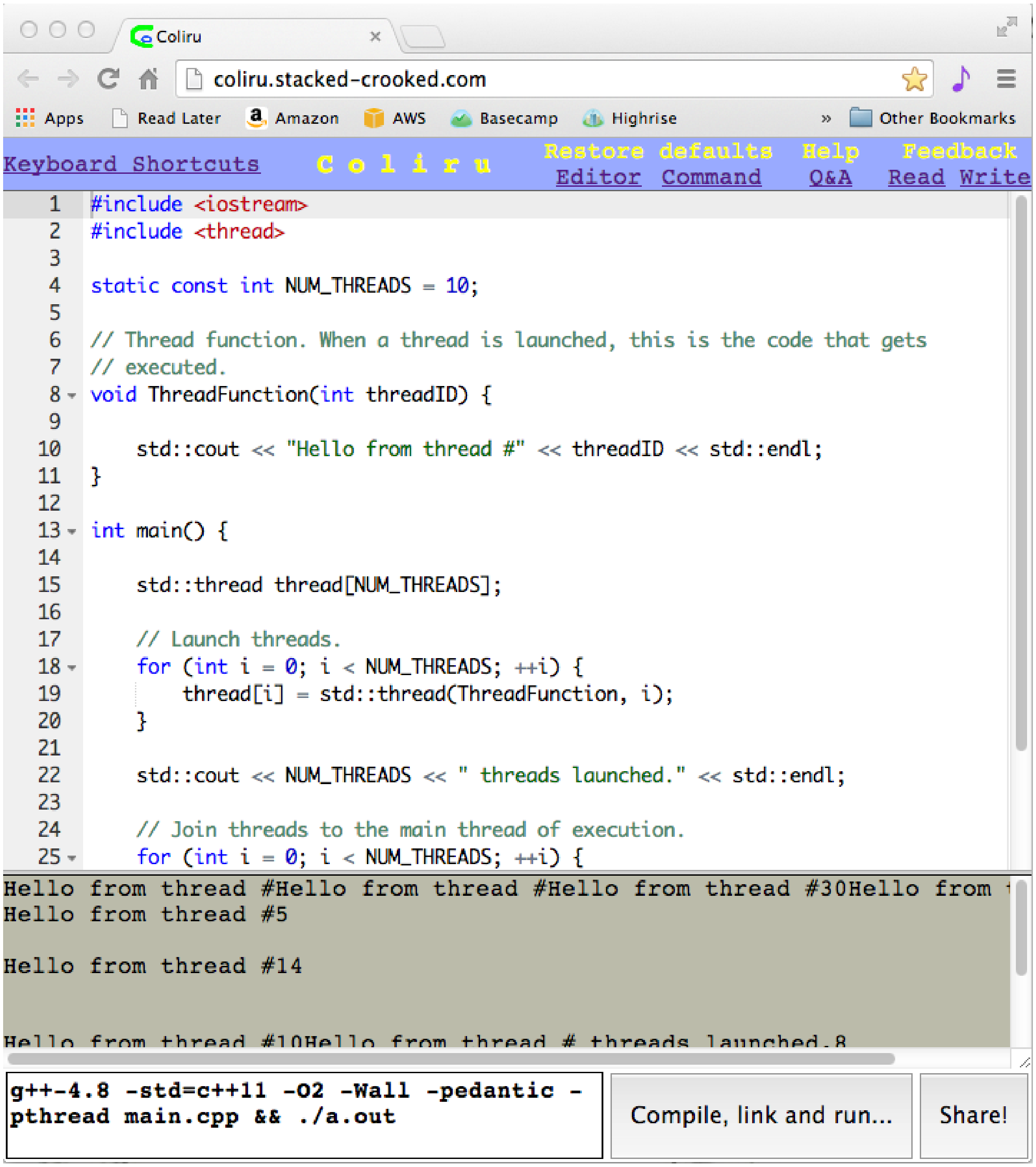

Figure 2. Online C++ compiler "Coliru".

Here's the output that I got from the program (you're output will likely be different because the program only approximates π):

```
Estimated value of PI (using 100000000 random samples): 3.14192
calculated in 2279 ms
```

4. **Running a Parallel Multithread Simulation**

Here are step-by-step instructions to compiling, running and looking at the output of the second simulation program:

1. As before, open a web browser and type in the address http://coliru.stacked-crooked.com/ .
2. Cut & paste the code from **Appendix II** into the main panel.
3. Then cut & paste the code from **Appendix III** directly below the previous code from step 2.
4. Then click on the button labeled "Compile, link and run …".
5. That's it! You've run your first parallel multithread simulation in C++.

Alternatively, follow these steps:

1. Click on this link: load code in browser.
2. Click on the button labeled "Edit".
3. Then click on the button labeled "Compile, link and run …".
4. That's it! You've run your first simulation in C++.

Let's again look at the output:

```
10 threads launched.
Estimated value of PI (using 100000000 random samples): 3.14164
calculated in 2529 ms
```

Note: The online compiler limits the number of threads a program can launch to about 10 threads. If you specify more threads the program won't run. However, the multithread version (with 10 threads) is at least twice as fast as the single thread version.

Also, you may be wondering why you had to include the code from **Appendix II**. The reason is that most standard random number generators (RNGs) are neither thread safe or designed to produce random numbers in parallel (PRNG). The code in Appendix II is an implementation of a PRNG. This is an important point and leads to Rule 3 of parallel programming (Rule 1 and Rule 2 were given in technical note "BDiF – TN0001 - Running your first multithread C++11 program.pdf").

Rule 3: When writing multithread and parallel programs you must be sure that any libraries you use are thread-safe, multithreaded and parallel.

5. **Summary**

The two code examples—serial and parallel estimation of π—in this technical note are small and largely self-explanatory. Even so, you would do well to study the code in detail because they provide two important lessons:

1. How to take a serial program and parallelize it.
2. That care must be taken when generating random numbers in parallel. Even experienced C++ programmers get this wrong.

The parallel version of the program can easily be adapted to less trivial tasks than calculating π, and in that sense is a good starting point for running your own simulations.

Appendix I

Source code for a C++11 program that calculates the value of π in a serial fashion:

_____ SNIP ~ Cut below this line when cutting & pasting _____

```
#include <climits>
#include <iostream>
#include <chrono>
#include <random>

// Needed to time things.
#define START_TIMER  std::chrono::system_clock::time_point t0 = \
std::chrono::system_clock::now();
#define END_TIMER    std::chrono::system_clock::time_point t1 = \
std::chrono::system_clock::now();
#define ELAPSED_TIME \
std::chrono::duration_cast<std::chrono::milliseconds>(t1 - t0).count()

const int NUM_SAMPLES = 100000000;

struct Point {

    private:
        std::default_random_engine rng;
        std::uniform_real_distribution<double> uniform;

    public:
        double x;
        double y;

    void next() {
        x = double(rng())/UINT32_MAX;
        y = double(rng())/UINT32_MAX;
    }

    int inside_circle() {
        return (((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5))<0.25) ? 1 : 0;
    }
};

int main()
{
    double pi = 0.0;

      START_TIMER

    Point p;
    int count = 0; // Count of how many darts fall inside/outside.
      for (int n = 0; n<NUM_SAMPLES; n++) {

        // Throw a dart at the unit square!
        p.next();
        count += p.inside_circle();
      }
```

```
      // Calculating pi
      pi = 4.0 * count / NUM_SAMPLES;

    END_TIMER

    std::cout << "Estimated value of PI (using " << NUM_SAMPLES;
    std::cout << " random samples): "<< pi;
    std::cout << " calculated in " << ELAPSED_TIME << " ms";
    std::cout << std::endl;

      return 0;
}
```

_____ SNIP ~ Cut above this line when cutting & pasting _____

Appendix II

Source code for portable parallel random number generator (PRNG):

_____ SNIP ~ Cut below this line when cutting & pasting _____

```
//  Copyright (c) 2012 M.A. (Thijs) van den Berg, http://sitmo.com/
//
//  Use, modification and distribution are subject to the MIT Software
License.
// (See accompanying file LICENSE.txt or copy at
http://www.stdfin.org/LICENSE.txt)

#ifndef STDFIN_RANDOM_THREEFRY_ENGINE_HPP
#define STDFIN_RANDOM_THREEFRY_ENGINE_HPP

#include <stdexcept>

#ifdef __GNUC__
    #include <stdint.h>                  // respecting the C99 standard.
#endif
#ifdef _MSC_VER
    typedef unsigned __int64 uint64_t;  // Visual Studio 6.0(VC6) and
newer..
    typedef unsigned __int32 uint32_t;
#endif

// Double mixing function
#define MIX2(x0,x1,rx,z0,z1,rz) \
    x0 += x1; \
    z0 += z1; \
    x1 = (x1 << rx) | (x1 >> (64-rx)); \
    z1 = (z1 << rz) | (z1 >> (64-rz)); \
    x1 ^= x0; \
    z1 ^= z0;

// Double mixing function with key adition
#define MIXK(x0,x1,rx,z0,z1,rz,k0,k1,l0,l1) \
    x1 += k1; \
    z1 += l1; \
    x0 += x1+k0; \
    z0 += z1+l0; \
    x1 = (x1 << rx) | (x1 >> (64-rx)); \
    z1 = (z1 << rz) | (z1 >> (64-rz)); \
    x1 ^= x0; \
    z1 ^= z0; \

namespace stdfin {

// "req" are requirements as stated in the C++ 11 draft n3242=11-0012

class threefry_engine
{
public:
    // req: 26.5.1.3 Uniform random number generator requirements,
p.906, table 116, row 1
```

```
    typedef uint32_t result_type;

    // req: 26.5.1.3 Uniform random number generator requirements,
p.906, table 116, row 3
    static result_type (min)() { return 0; }

    // req: 26.5.1.3 Uniform random number generator requirements,
p.906, table 116, row 4
    static result_type (max)() { return 0xFFFFFFFF; }

    // -------------------------------------------------
    // Constructors
    // -------------------------------------------------

    // req: 26.5.1.4 Random number engine requirements, p.907 table
117, row 1
    // Creates an engine with the same initial state as all other
    // default-constructed engines of type E.
    threefry_engine() { seed(); }

    // req: 26.5.1.4 Random number engine requirements, p.907 table
117, row 2
    // Creates an engine that compares equal to x.
    explicit threefry_engine(const threefry_engine& x)
    {
        for (int i=0; i<4; ++i) _s[i] = x._s[i];
        for (int i=0; i<4; ++i) _k[i] = x._k[i];
        for (int i=0; i<4; ++i) _o[i] = x._o[i];
        _o_counter = x._o_counter;
    }

    // req: 26.5.1.4 Random number engine requirements, p.907 table
117, row 3
    // Creates an engine with initial O(size of state) state determined
by s.
    explicit threefry_engine(const result_type& s) { seed(s); }


    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 4
    // Creates an engine with an initial state that depends on a
sequence
    // produced by one call to q.generate.
    template<class Seq>
    explicit threefry_engine(Seq& q) { seed(q); }

    // -------------------------------------------------
    // Seeding
    // -------------------------------------------------

    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 5
    void seed()
    {
        _k[0] = 0; _k[1] = 0; _k[2] = 0; _k[3] = 0;
        _s[0] = 0; _s[1] = 0; _s[2] = 0; _s[3] = 0;
        _o_counter = 0;
```

```
        _o[0] = 0x09218ebde6c85537;
        _o[1] = 0x55941f5266d86105;
        _o[2] = 0x4bd25e16282434dc;
        _o[3] = 0xee29ec846bd2e40b;
    }

    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 6
    void seed(const result_type& s)
    {
        _k[0] = s; _k[1] = 0; _k[2] = 0; _k[3] = 0;
        _s[0] = 0; _s[1] = 0; _s[2] = 0; _s[3] = 0;
        _o_counter = 0;
        encrypt_counter();

    }

    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 7
    template<class Sseq>
    void seed(Sseq& q)
    {
        typename Sseq::result_type w[8];
        q.genarate(&w[0], &w[8]);
        for (int i=0; i<4; ++i)
            _k[i] = (w[2*i] << 32) | w[2*i+1];
        _o_counter = 0;
        encrypt_counter();
    }

    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 8
    // Advances e’s state ei to ei+1 = TA(ei) and returns GA(ei).
    uint32_t operator()()
    {
        if (_o_counter < 8) {
            short _o_index = _o_counter >> 1;
            _o_counter++;
            if (_o_counter&1)
                return _o[_o_index];
            else
                return _o[_o_index] >> 32;
        }
        inc_counter();
        encrypt_counter();
        _o_counter = 1; // the next call
        return _o[0];   // this call
    }

    // -------------------------------------------------
    // misc
    // -------------------------------------------------

    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 9
    // Advances e’s state ei to ei+z by any means equivalent to z
```

```
    // consecutive calls e().
    void discard(uint64_t z)
    {
        // detect bit overflow, and process those
        if (z >= (0xFFFFFFFFFFFFFFFF - _s[0]) ) {
            ++_s[1];
            if (_s[1] == 0) {
                ++_s[2];
                if (_s[2] == 0) {
                    ++_s[3];
                }
            }
        }

        // add it
        _s[0] += z;

        _o_counter = 0;
        encrypt_counter();
    }

    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 10
    // This operator is an equivalence relation. With Sx and Sy as the
infinite
    // sequences of values that would be generated by repeated future
calls to
    // x() and y(), respectively, returns true if Sx = Sy; else returns
false.
    bool operator==(const threefry_engine& y)
    {
        if (_o_counter != y._o_counter) return false;
        for (int i=0; i<4; ++i) {
            if (_s[i] != y._s[i]) return false;
            if (_k[i] != y._k[i]) return false;
            if (_o[i] != y._o[i]) return false;
        }
        return true;
    }

    // req: 26.5.1.4 Random number engine requirements, p.908 table
117, row 11
    bool operator!=(const threefry_engine& y) { return !(*this == y); }


    // Extra function to set the key
    void set_key(uint64_t k0=0, uint64_t k1=0, uint64_t k2=0, uint64_t
k3=0)
    {
        _k[0] = k0; _k[1] = k1; _k[2] = k2; _k[3] = k3;
        _o_counter = 0;
        encrypt_counter();
    }

    // set the counter
    void set_counter(uint64_t s0=0, uint64_t s1=0, uint64_t s2=0,
uint64_t s3=0)
```

```
    {
        _s[0] = s0; _s[1] = s1; _s[2] = s2; _s[3] = s3;
        _o_counter = 0;
        encrypt_counter();
    }

private:
    void encrypt_counter()
    {
        uint64_t b[4];
        uint64_t k[5];

        for (int i=0; i<4; ++i) b[i] = _s[i];
        for (int i=0; i<4; ++i) k[i] = _k[i];

        k[4] = 0x1BD11BDAA9FC1A22 ^ k[0] ^ k[1] ^ k[2] ^ k[3];

        MIXK(b[0], b[1], 14,   b[2], b[3], 16,   k[0], k[1], k[2],
k[3]);
        MIX2(b[0], b[3], 52,   b[2], b[1], 57);
        MIX2(b[0], b[1], 23,   b[2], b[3], 40);
        MIX2(b[0], b[3],  5,   b[2], b[1], 37);
        MIXK(b[0], b[1], 25,   b[2], b[3], 33,   k[1], k[2], k[3],
k[4]+1);
        MIX2(b[0], b[3], 46,   b[2], b[1], 12);
        MIX2(b[0], b[1], 58,   b[2], b[3], 22);
        MIX2(b[0], b[3], 32,   b[2], b[1], 32);

        MIXK(b[0], b[1], 14,   b[2], b[3], 16,   k[2], k[3], k[4],
k[0]+2);
        MIX2(b[0], b[3], 52,   b[2], b[1], 57);
        MIX2(b[0], b[1], 23,   b[2], b[3], 40);
        MIX2(b[0], b[3],  5,   b[2], b[1], 37);
        MIXK(b[0], b[1], 25,   b[2], b[3], 33,   k[3], k[4], k[0],
k[1]+3);

        MIX2(b[0], b[3], 46,   b[2], b[1], 12);
        MIX2(b[0], b[1], 58,   b[2], b[3], 22);
        MIX2(b[0], b[3], 32,   b[2], b[1], 32);

        MIXK(b[0], b[1], 14,   b[2], b[3], 16,   k[4], k[0], k[1],
k[2]+4);
        MIX2(b[0], b[3], 52,   b[2], b[1], 57);
        MIX2(b[0], b[1], 23,   b[2], b[3], 40);
        MIX2(b[0], b[3],  5,   b[2], b[1], 37);

        for (int i=0; i<4; ++i) _o[i] = b[i] + k[i];
        _o[3] += 5;
    }

    void inc_counter()
    {
        ++_s[0];
        if (_s[0] == 0) {
            ++_s[1];
            if (_s[1] == 0) {
                ++_s[2];
```

```
                if (_s[2] == 0) {
                    ++_s[3];
                }
            }
        }
    }

private:

    uint64_t _k[4]; // key
    uint64_t _s[4]; // state (counter)
    uint64_t _o[4]; // cipher output

    short _o_counter; //
};

} // namespace stdfin

#undef MIXK
#undef MIX2

#endif
```

_____ SNIP ~ Cut above this line when cutting & pasting _____

Appendix III

Source code for a C++11 program that calculates the value of π in a parallel multithread fashion:

_____ SNIP ~ Cut below this line when cutting & pasting _____

```
#include <climits>
#include <iostream>
#include <chrono>
#include <random>
#include <thread>

// Needed to time things.
#define START_TIMER  std::chrono::system_clock::time_point t0 = \
std::chrono::system_clock::now();
#define END_TIMER    std::chrono::system_clock::time_point t1 = \
std::chrono::system_clock::now();
#define ELAPSED_TIME \
std::chrono::duration_cast<std::chrono::milliseconds>(t1 - t0).count()

static const int NUM_SAMPLES = 100000000;
static const int NUM_THREADS = 10;

struct Point {

private:
    stdfin::threefry_engine rng;

public:
    double x;
    double y;

    Point() {
        rng.seed();
    }

    Point(uint32_t tid) {
        rng.seed((int)tid);
    }

    void next() {
        x = double(rng())/UINT32_MAX;
        y = double(rng())/UINT32_MAX;
    }

    int inside_circle() {
        return (((x-0.5)*(x-0.5)+(y-0.5)*(y-0.5))<0.25) ? 1 : 0;
    }
};

static int count[NUM_THREADS];

// Thread function. When a thread is launched, this is the code
// that gets executed.
void ThreadFunction(int threadID, int num) {
```

```
    Point p(threadID);

      for (int i = 0; i<num; i++) {

        // Throw a dart at the unit square!
        p.next();
        count[threadID] += p.inside_circle();
      }
}

int main()
{
    double pi = 0.0;

      START_TIMER

    std::thread thread[NUM_THREADS];

      // Launch threads.
      for (int i = 0; i < NUM_THREADS; ++i) {
            thread[i] = std::thread(ThreadFunction, i,
                                    NUM_SAMPLES/NUM_THREADS);
      }

      std::cout << NUM_THREADS << " threads launched." << std::endl;

      // Join threads to the main thread of execution.
    int total = 0;
      for (int i = 0; i < NUM_THREADS; ++i) {
            thread[i].join();
        total += count[i];
      }

      // Calculating pi
      pi = 4.0 * total / NUM_SAMPLES;

    END_TIMER

    std::cout << "Estimated value of PI (using " << NUM_SAMPLES;
    std::cout << " random samples): "<< pi;
    std::cout << " calculated in " << ELAPSED_TIME << " ms";
    std::cout << std::endl << std::endl;

      return 0;
}
```

_____ SNIP ~ Cut above this line when cutting & pasting _____

Baruch College, City University of New York, is home to one of the top Financial Engineering master programs in the world.

QuantNet is the leading online educational resource for quantitative finance.

Fountainhead is an educational and consulting company specializing in financial Big Data.