Oregon State University

CS 475/575 -- Spring Quarter 2022

Project #1

OpenMP: Monte Carlo Simulation

100 Points

Due: April 15

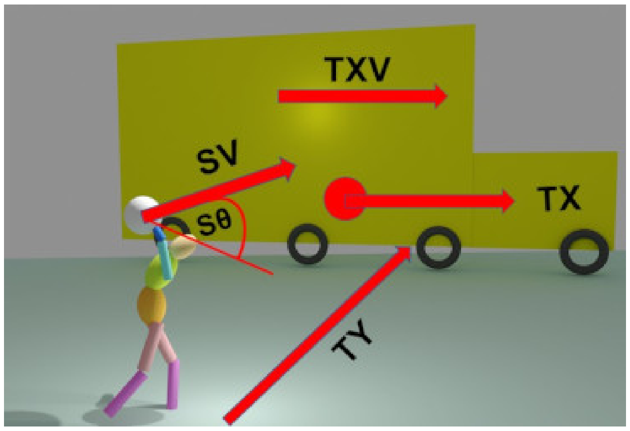
This page was last updated: March 24, 2022

Introduction

Monte Carlo simulation is used to determine the range of outcomes for a series of parameters, each of which has a probability distribution showing how likely each option is to happen. In this project, you will take a scenario and develop a Monte Carlo simulation of it, determining how likely a particular output is to happen.

Clearly, this is very parallelizable -- it is the same computation being run on many permutations of the input parameters. You will run this with OpenMP, testing it on different numbers of threads (1, 2, and 4, but more are OK).

The Scenario



A kid with a snowball is waiting for something to throw it at. (Hey you, get off my lawn!) Suddenly, a truck drives by. When the truck is even with the kid, the kid throws the snowball.

The origin for this coordinate system is at the kid's feet. The X direction goes to the right. The Y directiomn goes deeper into the scene.

But, it is hard to estimate when the truck is exactly even with the thrower. And, even though the a truck driving through snow should be going about 20 feet per second, it is hard to estimate the speed of the truck. And, it is hard for a thrower to get the throw velocity and the throw angle consistent from one time to the next.

Ignore the height of the snowball. It's a really tall truck. We are assuming that the only thing that matters are the horizontal positions of the truck and the snowball.

Use these as the ranges of the input parameters when you choose random parameter values:

Variable	Description	Minimum	Maximum
tx	Truck X starting location in feet	-10.	10.

txv	Truck X velocity in feet/second	10.	30.
ty	Truck Y location in feet	45.	55.
sv	Snowball overall velocity in feet/second	10.	30.
theta	Snowball horizontal launch angle in degrees	10.	90.

I recommend #define'ing or const float'ing all of these minima and maxima at the top of your program.

And, the half-length of the truck is 20. feet.

Given all this uncertainty, what is the probability that the snowball hits the truck?

Requirements:

Run this for some combinations of trials and threads. Do timing for each combination. Like we talked about in the **Project Notes**, run each experiment some number of tries, NUMTIMES, and record just the peak performance.

Do a table and two graphs. The two graphs need to be:

1. Performance versus the number of Monte Carlo trials, with the colored lines being the number of OpenMP threads.
2. Performance versus the number OpenMP threads, with the colored lines being the number of Monte Carlo trials..

(See the **Project Notes** to see an example of this and how to get Excel to do most of the work for you.)

Chosing one of the runs (the one with the maximum number of trials would be good), tell me what you think the actual probability is.

Compute Fp, the Parallel Fraction, for this computation.

Equations

The x and y components of the thrown snowball:

svx = sv * cos(theta)
svy = sv * sin(theta)
Note that **theta** must be converted from degrees to radians for this to work.

To find out at what time the snowball reaches y = ty in depth, solve this for time, t:

t = ty / svy;

To see where the snowball ended up in X in that time:

sbx = svx * t;

To see where the truck ended up in X in that time:

truckx = tx + txv * t;

If the absolute value of the difference between these last two values is less than the truck half-length, then the snowball hit the truck.

The Program Flow

The code below is printed in the handout to make it easy to look at and discuss. You can get this code in the file proj01.cpp.

```
#include <stdio.h>
#define _USE_MATH_DEFINES
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>

// print debugging messages?
#ifndef DEBUG
#define DEBUG   false
#endif

// setting the number of threads:
```

```
#ifndef NUMT
#define NUMT                2
#endif

// setting the number of trials in the monte carlo simulation:
#ifndef NUMTRIALS
#define NUMTRIALS       50000
#endif

// how many tries to discover the maximum performance:
#ifndef NUMTIMES
#define NUMTIMES        20
#endif

// ranges for the random numbers:
const float ????? =     ?????;
        . . .

// degrees-to-radians:
inline
float Radians( float d )
{
        return (M_PI/180.f) * d;
}




int
main( int argc, char *argv[ ] )
{
#ifndef _OPENMP
        fprintf( stderr, "No OpenMP support!\n" );
        return 1;
#endif

        TimeOfDaySeed( );               // seed the random number generator

        omp_set_num_threads( NUMT );    // set the number of threads to use in parallelizing the for-loop:`

        // better to define these here so that the rand() calls don't get into the thread timing:
        float *txs  = new float [NUMTRIALS];
        float *tys  = new float [NUMTRIALS];
        float *txvs = new float [NUMTRIALS];
        float *svs  = new float [NUMTRIALS];
        float *sths = new float [NUMTRIALS];

        // fill the random-value arrays:
        for( int n = 0; n < NUMTRIALS; n++ )
        {
                txs[n]  = Ranf(  TXMIN,  TXMAX );
                tys[n]  = Ranf(  TYMIN,  TYMAX );
                txvs[n] = Ranf(  TXVMIN, TXVMAX );
                svs[n]  = Ranf(  SVMIN,  SVMAX );
                sths[n] = Ranf(  STHMIN, STHMAX );
        }

        // get ready to record the maximum performance and the probability:
        double  maxPerformance = 0.;    // must be declared outside the NUMTIMES loop
        int     numHits;                // must be declared outside the NUMTIMES loop

        // looking for the maximum performance:
        for( int times = 0; times < NUMTIMES; times++ )
        {
                double time0 = omp_get_wtime( );

                numHits = 0;

                #pragma omp parallel for default(none) shared(txs,tys,txvs,svs,sths,stderr) reduction(+:numHits)
                for( int n = 0; n < NUMTRIALS; n++ )
                {
                        // randomize everything:
                        float tx   = txs[n];
                        float ty   = tys[n];
                        float txv  = txvs[n];
                        float sv   = svs[n];
                        float sthd = sths[n];
                        float sthr = ?????( sthd );
                        float svx  = ?????
                        float svy  = ?????

                        // how long until the snowball reaches the y depth:
                        float t = ?????

                        // how far the truck has moved in x in that amount of time:
                        float truckx = ?????

                        // how far the snowball has moved in x in that amount of time:
                        float sbx = ?????

                        // does the snowball hit the truck (just check x distances, not height):
                        if( fabs(?????) < ????? )
                        {
                                numHits++;
                                if( DEBUG )  fprintf( stderr, "Hits the truck at time = %8.3f\n", t );
                        }
                } // for( # of  monte carlo trials )

                double time1 = omp_get_wtime( );
                double megaTrialsPerSecond = (double)NUMTRIALS / ( time1 - time0 ) / 1000000.;
                if( megaTrialsPerSecond > maxPerformance )
                        maxPerformance = megaTrialsPerSecond;

        } // for ( # of timing tries )

        float probability = (float)numHits/(float)( NUMTRIALS );        // just get for last NUMTIMES run
        fprintf(stderr, "%2d threads : %8d trials ; probability = %6.2f%% ; megatrials/sec = %6.2lf\n",
                NUMT, NUMTRIALS, 100.*probability, maxPerformance);
```

Print out: (1) the number of threads, (2) the number of trials, (3) the probability of hitting the truck and (4) the MegaTrialsPerSecond. Printing this as a single line with commas between the numbers but no text is nice so that you can import these lines right into Excel as a CSV file.

Helper Functions:

To choose a random number between two floats or two ints, use:

```
#include <stdlib.h>

float
Ranf( float low, float high )
{
        float r = (float) rand();               // 0 - RAND_MAX
        float t = r  /  (float) RAND_MAX;       // 0. - 1.

        return   low  +  t * ( high - low );
}

int
Ranf( int ilow, int ihigh )
{
        float low = (float)ilow;
        float high = ceil( (float)ihigh );

        return (int) Ranf(low,high);
}

// call this if you want to force your program to use
// a different random number sequence every time you run it:
void
TimeOfDaySeed( )
{
        struct tm y2k = { 0 };
        y2k.tm_hour = 0;   y2k.tm_min = 0; y2k.tm_sec = 0;
        y2k.tm_year = 100; y2k.tm_mon = 0; y2k.tm_mday = 1;

        time_t  timer;
        time( &timer );
        double seconds = difftime( timer, mktime(&y2k) );
        unsigned int seed = (unsigned int)( 1000.*seconds );    // milliseconds
        srand( seed );
}
```

Setting Up To Compile This From a Makefile

```
montecarlo:     montecarlo.cpp
                g++ -O3     montecarlo.cpp   -o montecarlo  -lm  -fopenmp
```

Run it as:

```
make  montecarlo
./montecarlo
```

Setting Up To Compile and Run This From a Script

You can save yourself a *ton* of time by setting this up to run from a script. Check the **Project Notes** to see how to do that in C-shell, bash, or Python. If you've never done this before, learn it now! You will be surprised how much time this saves you throughout this class. Here it is as a C-shell script:

```
#!/bin/csh

foreach t ( 1 2 4 8 12 16 20 24 32 )
  foreach n ( 1 10 100 1000 10000 100000 500000 1000000 )
        g++ -O3   montecarlo.cpp  -DNUMT=$t -DNUMTRIALS=$n  -o montecarlo  -lm  -fopenmp
    ./montecarlo
  end
end
```

Run it as:

```
loop.csh >& proj1.csv
```

Diverting it to a CSV file sets you up to import it right into Excel.

Grading:

Feature	Points
Correct probability	20
Good graph of performance vs. number of trials	30
Good graph of performance vs. number of threads	30
Compute Fp, the Parallel Fraction (*show your work*)	20
Potential Total	**100**