CS 475/575 -- Spring Quarter 2021

Project #1

OpenMP: Monte Carlo Simulation

100 Points

Due: April 16

Lecture video: https://media.oregonstate.edu/media/t/1_5mzv15fl

---

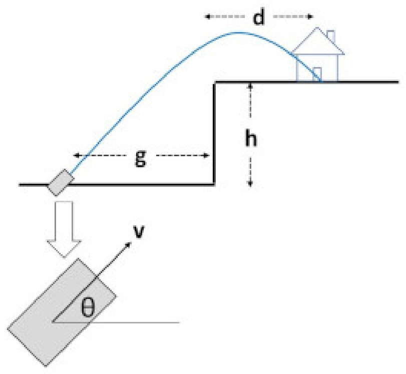
This page was last updated: April 3, 2021

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 castle sits on top of a cliff. An amateur band of merceneries is attempting to destroy it.

Normally this would be a pretty straightforward geometric calculation, but these are amateurs. What makes them amateurs you ask? It's because they are not very good at estimating distances, and not very good at aiming their cannon. They can only determine the 5 input parameters within certain ranges.

Your job is to figure out the probability that these doofuses will actually hit the castle. This is a job for multicore Monte Carlo simulation!

Requirements:

1. The ranges are:

Variable	Meaning	Range
g	Ground distance to the cliff face	20. - 30.
h	Height of the cliff face	10. - 20.
d	Upper deck distance to the castle	10. - 20.
v	Cannonball initial velocity	10. - 30.
θ	Cannon firing angle	30. - 70.

In addition you have:
vx = v*cos(θ)
vy = v*sin(θ)
TOL = 5.0
GRAVITY = -9.8

2. 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, NUMTRIES, 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 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

To find out at what time the ball hits y = 0. (returns to the ground level), solve this for t (time):
$y = 0. = 0. + vy*t + 0.5*GRAVITY*t^2$
Ignore the t=0. solution -- that's when it started.
To see how far the ball went horizontally in that time: x = 0. + vx*t
If this is less than **g**, then the ball never even reached the cliff face -- they missed the castle.

To find out what time the ball gets even with x = g (the cliff face), solve this for t (time):
x = g = 0. + vx*t
To see how far the ball went vertically in that time:
$y = 0. + vy*t + 0.5*GRAVITY*t^2$
If this is less than **h**, then the ball was unable to clear the cliff face -- they missed the castle.

To find out at what time the ball hits y = h (the upper deck), solve this for t (time):
$y = h = 0. + vy*t + 0.5*GRAVITY*t^2$
using the quadratic formula. (Here you thought you were done with this in high school.)

If you think of this equation as being in the form: $at^2 + bt + c = 0.$, then the value of t is:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

This will give you two solutions for time. Choose the larger of the two. The larger of the two gives you the time the cannonball reaches y=h on the way down. (The smaller of the two gives you the time the cannonball reaches y=h on the way up, which doesn't help.)
To see how far the ball went horizontally in that time use:
x = 0. + vx*t
If fabs(x-g - d) <= TOL, then the cannonball actually destroyed the castle. Even a blind squirrel finds an acorn every so often...

The Algorithm

```
#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 NUMTRIES
```

```
#define NUMTRIES        10
#endif

// ranges for the random numbers:
const float GMIN =      20.0;   // ground distance in meters
const float GMAX =      30.0;   // ground distance in meters
const float HMIN =      10.0;   // cliff height in meters
const float HMAX =      20.0;   // cliff height in meters
const float DMIN  =     10.0;   // distance to castle in meters
const float DMAX  =     20.0;   // distance to castle in meters
const float VMIN  =     10.0;   // intial cnnonball velocity in meters / sec
const float VMAX  =     30.0;   // intial cnnonball velocity in meters / sec
const float THMIN =     30.0;   // cannonball launch angle in degrees
const float THMAX =     70.0;   // cannonball launch angle in degrees

const float GRAVITY =   -9.8;   // acceleraion due to gravity in meters / sec^2
const float TOL = 5.0;          // tolerance in cannonball hitting the castle in meters
                                // castle is destroyed if cannonball lands between d-TOL and d+TOL

// function prototypes:
float           Ranf( float, float );
int             Ranf( int, int );
void            TimeOfDaySeed( );

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


// main program:
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 *vs  = new float [NUMTRIALS];
        float *ths = new float [NUMTRIALS];
        float * gs = new float [NUMTRIALS];
        float * hs = new float [NUMTRIALS];
        float * ds = new float [NUMTRIALS];

        // fill the random-value arrays:
        for( int n = 0; n < NUMTRIALS; n++ )
        {
                vs[n]  = Ranf(  VMIN,  VMAX );
                ths[n] = Ranf( THMIN, THMAX );
                gs[n]  = Ranf(  GMIN,  GMAX );
                hs[n]  = Ranf(  HMIN,  HMAX );
                ds[n]  = Ranf(  DMIN,  DMAX );
        }

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

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

                numHits = 0;

                #pragma omp parallel for ?????
                for( int n = 0; n < NUMTRIALS; n++ )
                {
                        // randomize everything:
                        float v   = vs[n];
                        float thr = Radians( ths[n] );
                        float vx  = v * cos(thr);
                        float vy  = v * sin(thr);
                        float  g  =  gs[n];
                        float  h  =  hs[n];
                        float  d  =  ds[n];
                        // see if the ball doesn't even reach the cliff:`
                        float t = ?????
                        float x = ?????
                        if( x <= g )
                        {
                                if( DEBUG )     fprintf( stderr, "Ball doesn't even reach the cliff\n" );
                        }
                        else
                        {
                                // see if the ball hits the vertical cliff face:
                                t = ?????
                                float y = ?????
                                if( y <= h )
                                {
                                        if( DEBUG )     fprintf( stderr, "Ball hits the cliff face\n" );
                                }
                                else
                                {
                                        // the ball hits the upper deck:
                                        // the time solution for this is a quadratic equation of the form:
                                        // at^2 + bt + c = 0.
                                        // where 'a' multiplies time^2
                                        //       'b' multiplies time
                                        //       'c' is a constant
                                        float a = ?????
                                        float b = ?????
                                        float c = ?????
                                        float disc = b*b - 4.f*a*c;     // quadratic formula discriminant

                                        // ball doesn't go as high as the upper deck:
                                        // this should "never happen" ... :-)
                                        if( disc < 0. )
                                        {
                                                if( DEBUG )     fprintf( stderr, "Ball doesn't reach the upper deck.\n" );
                                                exit( 1 );      // something is wrong...
                                        }

                                        // successfully hits the ground above the cliff:
                                        // get the intersection:
                                        disc = sqrtf( disc );
                                        float t1 = (-b + disc ) / ( 2.f*a );    // time to intersect high ground
                                        float t2 = (-b - disc ) / ( 2.f*a );    // time to intersect high ground

                                        // only care about the second intersection
                                        float tmax = t1;
                                        if( t2 > t1 )
                                                tmax = t2;

                                        // how far does the ball land horizontlly from the edge of the cliff?
                                        float upperDist = vx * tmax  -  g;

                                        // see if the ball hits the castle:
                                        if(  fabs( upperDist - d ) > TOL )
                                        {
                                                if( DEBUG )  fprintf( stderr, "Misses the castle at upperDist = %8.3f\n", upperDist );
                                        }
                                        else
                                        {
                                                if( DEBUG )  fprintf( stderr, "Hits the castle at upperDist = %8.3f\n", upperDist );
                                                ?????
                                        }
                                } // if ball clears the cliff face
                        } // if ball gets as far as the cliff face
                } // 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 NUMTRIES run
        fprintf(stderr, "%2d threads : %8d trials ; probability = %6.2f%% ; megatrials/sec = %6.2lf\n",
                NUMT, NUMTRIALS, 100.*probability, maxPerformance);

        return 0;
}
```

Print out: (1) the number of threads, (2) the number of trials, (3) the probability of destroying the castle, and (4) the MegaTrialsPerSecond. Printing this as a single line with tabs or commas between the numbers is nice so that you can import these lines right into Excel.

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 );
}
```

Grading:

Feature	Points
Close estimate of the actual 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**