CSCI 455: Project 2 — Particle Simulation

Darwin Jacob Groskleg

Winter 2020

Contents

Questions **1**
partsim.c 2
physics.h 7
physics.c 8
definitions.h 11
coordinate.h 14
random.h 15
random.c 17

Questions

You should measure and report the following interesting quantities:

1. Explain your choice of distribution of the particles over the processors. Is there an optimal relation between the distribution and the geometry of the domain? Measure this by counting particles passed between processors in each time step.
2. What is the speedup with 1, 2, 4, 16, 32,… processors.
3. What is the scaled speedup (let the number of particles grow proportional to the number of processors).
4. Verify the gas law pV = nRT by changing the number of particles (n) and size of the box (V) and then measure the pressure.
5. Bonus Project: Implement a big and heavy particle and plot the trajectory of it. This kind of motion is called Brownian motion and was found by a biologist studying the motion of pollen on the water surface in a bucket. This is also a example of a random walk and fractal by nature.

partsim.c

```c
/* partsim.c
 * ---------
 * Authors: Darwin Jacob Groskleg
 *
 *  Part I:  Ideal Gas Law Particle Simulation
 *  Part II: Brownian Motion Particle Simulation
 */
#include <mpi.h>
#include <stdlib.h>
#include <math.h>

#include "coordinate.h"
#include "definitions.h"
#include "physics.h"
#include "random.h"

// Developer's Config
#define SIM_TOTAL_TIME 100

#define VDIM  0   // VERTICAL_DIM
#define HDIM  1   // HORIZONTAL_DIM
enum Directions { Above, Below, Left, Right };
enum TaskRanks {
    SendTaskRank = 0
};

typedef struct Grid { int rows; int cols; } grid_t;
grid_t OptimalGrid(int subunits) {
    int k=sqrt(subunits);
    while (k>0 && subunits%k != 0)
        k--;
    return (grid_t) { .rows = k, .cols = subunits / k };
}

// 2D is implied by Box, ndims is not needed.
box_t BoxSubSection(box_t box, grid_t grid, int coords[2]) {
    float section_width  = box.width  / grid.cols;
    float section_height = box.height / grid.rows;
    return (box_t) {
        .width  = section_width,
        .height = section_height,
        .coord  = {
            .x0 = coords[HDIM] * section_width,
            .y0 = coords[VDIM] * section_height,
            .x1 = (coords[HDIM] == grid.cols-1)
                    ? box.width
                    : (coords[HDIM]+1) * section_width,
            .y1 = (coords[VDIM] == grid.rows-1)
                    ? box.height
                    : (coords[VDIM]+1) * section_height,
        }
    };
}

// Global temperature of the box
```

```
56 double GetChamberTemperature(double section_temperature, MPI_Comm comm) {
57     double temp_sum = 0;
58     MPI_Reduce(&section_temperature, &temp_sum, 1, MPI_DOUBLE, MPI_SUM, 0,comm);
59     int cluster_size;
60     MPI_Comm_size(comm, &cluster_size);
61     return temp_sum / TotalParticles(cluster_size);
62 }
63 
64 // knows size, knows values, is monadic
65 // returns temperature
66 double GetRandomParticles(box_t box, particle_t **prtcl_arr) {
67     *prtcl_arr = (particle_t*)malloc(ParticlesForSection()*sizeof(particle_t));
68     double temperature=0;
69     if (*prtcl_arr != NULL) {
70         for (int i=0; i<ParticlesForSection(); i++) {
71             *(*prtcl_arr+i) = ConstructRandomParticle(box);
72             temperature += (*prtcl_arr+i)->temp_0;
73         }
74     }
75     return temperature;
76 }
77 void SafeFree(void **pp){
78     if (pp != NULL && *pp != NULL) {
79         free(*pp);
80         *pp = NULL;
81     }
82 }
83 
84 typedef struct NeighborBuffer {
85     int sendb_size[4];
86     int recvb_size[4];
87     particle_t sendb[4][PARTICLE_BUFFER_SIZE];
88     particle_t recvb[4][PARTICLE_BUFFER_SIZE];
89 } neighbor_buff_t;
90 
91 
92 int main(int argc, char *argv[]) {
93 
94     MPI_Init(&argc, &argv);
95     int rank;
96     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
97     int cluster_size;
98     MPI_Comm_size(MPI_COMM_WORLD, &cluster_size);
99 
100    /// WHAT IS THE PARALLEL PATTERN?
101    ///     - Divide and Conquer: divide 2D map into particle sub-systems,
102    ///         one for each process.
103    ///     - See Lecture 10: Partitioning, suggest D&C for N-body problems.
104    ///     - Thus computation follows static load balancing, imbalance in the
105    ///         distribution of particles more likely as time goes on.
106    /// IS THERE A COMMUNICATION TOPOLOGY?
107    ///     - Each process is a sub-section of the chamber, either think of this
108    ///         recursively or have back and forth between a process and its 8
109    ///         neighboring sections.
110    /// HOW AND WHEN DO PROCESSES COMMUNICATE WITH ONE ANOTHER?
111    ///     - Communication may either be coordinatedj
```

```
    /// WHO INITIALIZES WHAT PARTICLES?
    /// IS THERE AN MPI AWARE APPROACH TO RANDOM SEEDING REQUIRED?
    ///     - No but randomization library has been modified to allow seed
    ///         offsets.
    /// HOW MANY ACTUAL PARTICLES IF WE PLAN TO SCALE PROCESSOR COUNT?

// BUILD GLOBAL BOX
// Depends on:  cluster_size,

    // Greater system: chamber construction
    box_t chamber = {
        .width  = BOX_HORIZ_SIZE,
        .height = BOX_VERT_SIZE,
        .coord  = {
            .x0 = 0,    // left wall
            .y0 = 0,    // bottom wall
            .x1 = BOX_HORIZ_SIZE,   // right wall
            .y1 = BOX_VERT_SIZE
        }
    };

    // Setup Cartesian grid that the discretized particles will exist on.
    // Assuming the grid is square but subsections might be rectangular???
    //
    // We want the smallest grid size that is
    // divisible by the number of processors in our cluster. This optimizes for
    // 2 things: floating point number numerical limits and workload
    // divisibility, allowing the simulation's gas chamber to be easily divided
    // into subsections accross the cluster.
    grid_t grid = OptimalGrid(cluster_size);
    int dims[2];
    dims[VDIM] = grid.rows;
    dims[HDIM] = grid.cols;
    MPI_Dims_create(cluster_size, 2, dims);
    int periods[2] = {1, 1}; // both dims are periodic, world wrapping allowed
    int reorder = 1; // true
    MPI_Comm grid_comm;
    MPI_Cart_create(MPI_COMM_WORLD, 2, dims, periods, reorder, &grid_comm);


// BUILD LOCAL BOX
//  depends on:     grid, grid_comm

    // Get coordinates on grid assigned for current process rank
    int grid_section_coords[2];
    MPI_Cart_get(grid_comm, 2, dims, periods, grid_section_coords);

    // Chamber subsection for current rank
    //  require: chamber, grid, coords[2],
    box_t section = BoxSubSection(chamber, grid, grid_section_coords);

    // Get neighboring processor ranks
    int neighbors[4];
    MPI_Cart_shift(grid_comm, VDIM, 1, &neighbors[Below], &neighbors[Above]);
    MPI_Cart_shift(grid_comm, HDIM, 1, &neighbors[Left],  &neighbors[Right]);

```

```
168     // Greater system: particle initialization?
169
170
171     // Greater system: initial values for IGL Equation
172     // R = 8.3145 joules/moles/kelvin
173
174     // Seed the entropy server
175     RandomizeWithOffset(rank);
176     // Initiate particles.
177     // TODO change this to a linked list
178     particle_t *local_particles = NULL;
179     typedef struct ParticleList {} ParticleList_t;
180     ParticleList_t plist;
181     double section_temp = GetRandomParticles(section, &local_particles);
182
183     double temperature = GetChamberTemperature(section_temp, grid_comm);
184     double section_pressure = 0.0;
185
186     typedef struct ParticleCollision {
187         particle_t *a;
188         particle_t *b;
189         // temperature???
190         // time???
191     } ParticleCollision_t;
192     typedef struct CollisionList     {} CollisionList_t;
193
194     neighbor_buff_t mailbox;
195
196     // Each time-step do:
197     //      - for all particles:
198     //          - 1. check for collisions
199     //          - 2. move particles not colliding
200     //          - 3. check for wall interaction
201     //               and add the momentum
202     //      - Communicate if needed
203     //
204     // TIME == SPEED OF CAUSALITY
205     for (int t=0; t<SIM_TOTAL_TIME; t++) {
206
207         // Clean the mailbox: send and receivce buffers
208         for (int direction=Above; direction<=Right; direction++) {
209
210         }
211
212         //// Iterating foreach-style over particles guarantees O(N^2)
213         //for (node_t *part_ptr=plist.head; part_ptr!=NULL; part_ptr=part_ptr->next) {
214         // 1. Find collisions in local section of box
215
216         /* FindCollisions
217          * returns
218          *      - A list of collisions between p_i and p_last,
219          *          that can be appended to the greater collision list.
220          *      - Mutates *particle_ptr to be the next particle in list
221          *          whenever a collision is found.
222          *      ASSUMES only 2 particles in a collision.
223          *      - Removes colliding particles from the particle list,
```

```
224             *          safely leaving others as 'non-colliding' particles.
225             */
226            //CollisionList_t collisions = FindCollisions(&part_ptr);
227
228            /* ExtractCollisions
229             *
230             * RETURNS
231             *  A list of collisions (pairs) between particles in the given list.
232             *
233             * MUTATES
234             *  The given particle list, the colliding particles are removed.
235             *
236             * ASSUMES
237             *  A collision only ever happens between 2 particles and no more.
238             *
239             * Complexity: O(N^2)
240             */
241            CollisionList_t collisions = ExtractCollisions(&plist);
242
243            // 2. Move non-colliding particles
244            StepTimeForward(&plist);
245
246            // 3. Simulate collisions and add particles back to list
247            //  MUTATION
248            //      Collision list will be empty.
249            StepCollisionsForward(&collisions, &plist);
250
251            // 4. Check for wall interaction and add the momentum
252            //  MUTATIONS
253            //      Elements in plist will have their vectors changed to account for
254            //      wall rebound.
255            section_pressure += AdjustForWallPressure(chamber, &plist);
256
257            // 5. Gather those particles thath have left this section of the chamber
258            CollectSectionEmigrants(section, &plist, &mailbox);
259
260            // 6. Send particles asynchronously
261            MPI_Request isend_status[4];
262            DispatchEmigrantParticles(&mailbox, &isend_status, grid_comm);
263
264            // 7. Receive particles form neighboring sections, wait until done
265            IntegrateImmigrantParticles(&plist, &mailbox, &isend_status, grid_comm);
266
267        }
268        // free(particleList)
269
270        // Caculate final pressure.
271        double pressure = GetChamberPressure(section_pressure, grid_comm);
272
273        printf("Final pressure: %.3lf\n", pressure);
274
275        MPI_Finalize();
276        return 0;
277    }
```

physics.h

```c
#ifndef _physics_h
#define _physics_h

#include "coordinate.h"

/* the step size use in the integration */
#define STEP_SIZE 1.0

/*
 * Used to move a particle.
 */
int feuler(pcoord_t *a, float time_step);

/*
 * Checks if a particle has exceeded the boundary and returns a momentum.
 *
 * Use this momentum to calculate the pressure.
 */
float wall_collide(pcoord_t *p, lsegment_t wall);

/*
 * Checks if there will be no collision at this time step, returning -1,
 * otherwise it will return when the collision occurs.
 *
 * Can be used as one of input parameter to the routine interact.
 */
float collide(pcoord_t *p1, pcoord_t *p2);

/*
 * The routine interact moves two particles involved in a collision, at a
 * particular time.
 * Do not move these particles again.
 */
void interact(pcoord_t *p1, pcoord_t *p2, float t);

#endif /* _physics_h */
```

physics.c

```
#include "physics.h"
#include "coordinate.h"

#include <stdlib.h>
#include <math.h>

#ifndef sqr
#define sqr(a) ((a)*(a))
#endif

#ifndef sign
#define sign(a) ((a) > 0 ? 1 : -1)
#endif

int feuler(pcoord_t *a, float time_step) {
    a->x = a->x + time_step* a->vx;
    a->y = a->y + time_step* a->vy;
    return 0;
}

float wall_collide(pcoord_t *p, lsegment_t wall) {
    float gPressure = 0.0;

    if (p->x < wall.x0) {
        p->vx = -p->vx;
        p->x  = wall.x0 + (wall.x0-p->x);
        gPressure += 2.0*fabs(p->vx);
    }
    if (p->x > wall.x1) {
        p->vx = -p->vx;
        p->x  = wall.x1 - (p->x-wall.x1);
        gPressure += 2.0*fabs(p->vx);
    }
    if (p->y < wall.y0) {
        p->vy = -p->vy;
        p->y  = wall.y0 + (wall.y0-p->y);
        gPressure += 2.0*fabs(p->vy);
    }
    if (p->y > wall.y1) {
        p->vy = -(p->vy);
        p->y  = wall.y1 - (p->y-wall.y1);
        gPressure += 2.0*fabs(p->vy);
    }
    return gPressure;
}



float collide(pcoord_t *p1, pcoord_t *p2) {
    double a,b,c;
    double temp,t1,t2;

    a =  sqr(p1->vx-p2->vx)
        +sqr(p1->vy-p2->vy);
    b = 2*(  (p1->x - p2->x)*(p1->vx - p2->vx)
```

```
            +(p1->y - p2->y)*(p1->vy - p2->vy));
    c =  sqr(p1->x-p2->x)
        +sqr(p1->y-p2->y)
        -4*1*1;

    if (a!=0.0) {
        temp = sqr(b)-4*a*c;
        if (temp>=0) {
            temp=sqrt(temp);
            t1=(-b+temp)/(2*a);
            t2=(-b-temp)/(2*a);

            if (t1>t2) {
                temp=t1;
                t1=t2;
                t2=temp;
            }
            if ((t1>=0)&(t1<=1))
                return t1;
            else if ((t2>=0)&(t2<=1))
                return t2;
        }
    }
    return -1;
}



void interact(pcoord_t *p1, pcoord_t *p2, float t){
    float c,s,a,b,tao;
    pcoord_t p1temp,p2temp;

    if (t>=0) {
        /* Move to impact point */
        (void)feuler(p1,t);
        (void)feuler(p2,t);

        /* Rotate the coordinate system around p1*/
        p2temp.x = p2->x-p1->x;
        p2temp.y = p2->y-p1->y;

        /* Givens plane rotation, Golub, van Loan p. 216 */
        a = p2temp.x;
        b = p2temp.y;
        if (p2->y==0) {
            c=1;s=0;
        }
        else {
            if (fabs(b)>fabs(a)) {
                tao=-a/b;
                s=1/(sqrt(1+sqr(tao)));
                c=s*tao;
            }
            else {
                tao=-b/a;
                c=1/(sqrt(1+sqr(tao)));
```

```
                s=c*tao;
            }
        }

        p2temp.x=c * p2temp.x+s * p2temp.y; /* This should be equal to 2r */
        p2temp.y=0.0;

        p2temp.vx= c* p2->vx + s* p2->vy;
        p2temp.vy=-s* p2->vx + c* p2->vy;
        p1temp.vx= c* p1->vx + s* p1->vy;
        p1temp.vy=-s* p1->vx + c* p1->vy;

        /* Assume the balls has the same mass... */
        p1temp.vx=-p1temp.vx;
        p2temp.vx=-p2temp.vx;

        p1->vx = c * p1temp.vx - s * p1temp.vy;
        p1->vy = s * p1temp.vx + c * p1temp.vy;
        p2->vx = c * p2temp.vx - s * p2temp.vy;
        p2->vy = s * p2temp.vx + c * p2temp.vy;

        /* Move the balls the remaining time. */
        c=1.0-t;
        (void)feuler(p1,c);
        (void)feuler(p2,c);
    }
}
```

definitions.h

```
/* definitions.h
 * -------------
 * Purpose:
 *  For keeping all simulation assumptions and initial values given by the
 *  instructor in a single place. Any other assumptions not given by
 *  instruction will be listed elsewhere.
 */
//#include <stdlib.h>
//#include <math.h>
#include "coordinate.h"

#ifndef _definitions_h
#define _definitions_h

#ifndef PI
#define PI 3.141592653
#endif // PI

/// REQUIRED INITIAL VALUES (Section 2.3)

/// - Each time-step must be 1 time unit long.
///  1 time-step == 1 time unit long (seconds?)
#define TIME_STEP 1

#include "random.h"
#include <math.h>
/// - Particle vector initialization:
///   initial_velocity < 50 (m/s)
///   absolute_velocity = rand()
///     r = rand() * MAC_INITIAL_VELOCITY
///     theta_0 = rand() * 2*PI
///     vx_0 = r*cos(theta_0)
///     vy_0 = r*sin(theta_0)
#define MAX_INITIAL_VELOCITY  50
inline particle_t ConstructRandomParticle(box_t box){
    float r = RandomReal(0, MAX_INITIAL_VELOCITY);
    float theta = RandomReal(0, 2*PI);
    float vx_0 = r * cos(theta);
    float vy_0 = r * sin(theta);
    return (particle_t) {
        .pcoord.x  = RandomReal(box.coord.x0, box.coord.x1),
        .pcoord.y  = RandomReal(box.coord.y0, box.coord.y1),
        .pcoord.vx = vx_0,
        .pcoord.vy = vy_0,
        .temp_0 = (r*r) / 2 // from Maxwell's equations
    };
}
//extern pcoord_t ConstructRandomParticle(int seed);

/// - Particle count
///     Suggested 10,000 x processing_cluster_size in assignment.
///     Below sets Initial value.
///     So which is it.
///   Simulation must end if 30x initial particles end up in any processor's
///   sub-section.
```

```
56 #define MAX_NO_PARTICLES   15000  /* Maximum number of particles/processor */
57 #define INIT_NO_PARTICLES    500  /* Initial number of particles/processor */
58 inline int TotalParticles(int processor_count) {
59     return processor_count * INIT_NO_PARTICLES;
60 }
61 inline int ParticlesForSection() {
62     return INIT_NO_PARTICLES;
63 }
64 //inline int ParticlesForSection(int p_rank, int processor_count) {
65 //    p_rank++;   // dummy
66 //    processor_count++;  // dummy
67 //    return INIT_NO_PARTICLES;
68 //}
69
70 // Worst case Receiving, how many particles per buffer?
71 //  initial worst case: all in section,
72 //                      init_no_particles == 500
73 //  overall worst case: all particles in all sections end up in one, then move.
74 //                      NOT POSSIBLE.
75 //  actual worst case:  maxed particles from all 4 neighborsj,
76 //
77 #define PARTICLE_BUFFER_SIZE        MAX_NO_PARTICLES/5
78 // Worst case Sending: the max number of particles a process can hold.
79 //                     MAX_NO_PARTICLES
80 //                     == COMM_BUFFER_SIZE
81 // Worst case Recving: the max number of particles
82 // IGNORE COMM_BUFFER_SIZE for our implementation
83 #define COMM_BUFFER_SIZE            5*PARTICLE_BUFFER_SIZE
84 inline int MoveBufferToList(particle_t *arr, int arr_size, list_t *ll) {
85     // ASSERT
86     // arr <= PARTICLE_BUffER_SIZE
87     // ll->size  <= MAX_NO_PARTICLES
88     while (arr_size--) {
89         ParticleListAppend(ll, arr[arr_size] );
90     }
91     return arr_size;
92 }
93 inline int MoveParticleToBuffer(node_t *pnp, particle_t *arr, int *arr_size) {
94     // ASSERT
95     //      (*arr_size) < PARTICLE_BUFFER_SIZE
96     if (pnp != NULL && arr != NULL && arr_size != NULL) {
97         arr[*arr_size] = pnp->element;
98         (*arr_size)++;
99         ParticleListDeleteNode(pnp);
100    }
101 }
102
103
104 /// - Area of Box: width x height = 10^4 x 10^4
105 #define BOX_HORIZ_SIZE 10000.0
106 #define BOX_VERT_SIZE  10000.0
107 #define WALL_LENGTH (2.0*BOX_HORIZ_SIZE+2.0*BOX_VERT_SIZE)
108
109
110 /// - The Big Particle (for optional Brownian motion simulation)
111
```

```
112 #endif  /* _definitions_h */
```

coordinate.h

```c
#ifndef _coordinate_h
#define _coordinate_h

/* Line Segment
 *
 * This struct has to be some of the worst semantics I've ever seen.
 * It was originally defined as a coordinate (mispelled I might add).
 * WTF is this? A coordinate, a vector, a box, a segment?
 * Sure it can be all these things but that does not make it clever.
 *
 * This is dirty code.
 * Code is not just meant for the compiler but for the reader.
 * C can still be a beautiful language with proper semantics and just as fast.
 * Do better than this.
 */
typedef struct line_segment {
    float x0;
    float x1;
    float y0;
    float y1;
} lsegment_t;

/* A struct useful for the global simulation box
 * and to each process managing a subsection of the simulation box.
 */
typedef struct CoordinateBox {
    float width;
    float height;
    // x0 is the left side of the box, x1 is the right side
    // y0 is the bottom side of the box, y1 is the top side
    lsegment_t coord;
} box_t;

/* Particle Coordinate
 *  has a location in the plane (x, y)
 *  has a velocity as two vectors (vx, vy)
 */
typedef struct particle_coordinate {
    float x;
    float y;
    float vx;
    float vy;
} pcoord_t;


/* Particle: particle are defined by physics */
typedef struct particle {
    pcoord_t  pcoord;
    /* Used to simulate mixing of gases */
    int ptype;
    double temp_0;
} particle_t;

#endif  /* _coordinate_h */
```

random.h

```c
/* random.h
 * --------
 * Credit:  Dr. Martin van Bommel
 * Authors: Dr. Martin van Bommel, Darwin Jacob Groskleg
 * Modified from CSCI 162 / Assig 3
 */
#ifndef RANDOM_H_INCLUDED
#define RANDOM_H_INCLUDED

#include <stdbool.h>

/*
 * Function: Randomize
 * Usage: Randomize();
 * -------------------
 * This function sets the random seed so that the random sequence
 * is unpredictable.  During the debugging phase, it is best not
 * to call this function, so that program behavior is repeatable.
 * Otherwise only call it once at the start of use of random values.
 *
 * Only needs to be called once per process.
 */
extern void Randomize();
extern void RandomizeWithOffset(int seed_offset);
extern void RandomizeSeedWithOffset(int seed_offset, unsigned int seed);

/*
 * Function: RandomInteger
 * Usage: i = RandomInteger(low, high);
 * ------------------------------------
 * This function returns an integer in the closed interval [low..high],
 * meaning that the result is greater than or equal to low and less than
 * or equal to high, with each value having equal probability.
 */
extern int RandomInteger(int low, int high);

/*
 * Function: RandomReal
 * Usage: d = RandomReal(low, high);
 * ---------------------------------
 * This function returns a random real number in the half-open
 * interval [low .. high), meaning that the result is always
 * greater than or equal to low but strictly less than high.
 */
extern double RandomReal(double low, double high);

/*
 * Function: RandomChance
 * Usage: if (RandomChance(p)) . . .
 * ---------------------------------
 * The RandomChance function returns true with the probability
 * indicated by p, which should be a floating-point number between
 * 0 (meaning never) and 1 (meaning always).  For example, calling
 * RandomChance(.30) returns true 30 percent of the time.
 */
```

```
56 extern bool RandomChance(double probability);
57
58 /*
59  * Function: RandomNormal
60  * Usage: n = RandomNormal(mean, std);
61  * ----------------------------------
62  * This function returns a random real number following the
63  * normal distribution with given mean and standard deviation (std).
64  */
65 double RandomNormal(double mean, double std);
66
67 #endif /* RANDOM_H_INCLUDED */
```

random.c

```c
/* File: random.c
 * --------------
 * Credit:  Dr. Martin van Bommel
 * Authors: Dr. Martin van Bommel, Darwin Jacob Groskleg
 * Modified from CSCI 162 / Assig 3
 *
 * This file implements the random.h interface.
 */
#include "random.h"

#include <stdlib.h>
#include <time.h>
#include <math.h>

#ifdef __cplusplus
//using namespace std;
#endif

static bool isRandomized = false;

/*
 * Function: Randomize
 * Usage: Randomize();
 * -------------------
 * This function sets the random seed using the system time
 * so that the random sequence is unpredictable.
 *
 * Need to be able to add an offset for randomization accross processes that
 * are started at the same time. The square of the offset is used to ensure it
 * is a positive value and to exagerrate the offset further where similar offset
 * in the result from time() may negate it's effect.
 */
void Randomize()
{
    RandomizeSeedWithOffset(0, time(NULL));
}

void RandomizeWithOffset(int seed_offset)
{
    RandomizeSeedWithOffset(seed_offset, time(NULL));
}

void RandomizeSeedWithOffset(int seed_offset, unsigned int seed)
{
    if (!isRandomized)
        srand(seed + pow(seed_offset, 2));
    isRandomized = true;
}

/*
 * Function: RandomInteger
 * -----------------------
 * This function first obtains a random integer in
 * the range [0..RAND_MAX] and converts it into an
 * integer in the range [low..high] by applying the
```

```
 * four steps:
 * (1) Generate a real number between 0 and 1.
 * (2) Scale it to the appropriate range size.
 * (3) Truncate the value to an integer.
 * (4) Translate it to the appropriate range.
 */
int RandomInteger(int low, int high)
{
    int k;
    double d;

    d = (double) rand() / ( (double) RAND_MAX + 1);
    k = (int) (d * (high - low + 1));
    return (low + k);
}

/*
 * Function: RandomReal
 * Usage: d = RandomReal(low, high);
 * ---------------------------------
 * This function returns a random real number in the half-open
 * interval [low .. high).  The function first obtains a random
 * integer in the range [0..RAND_MAX] and converts it
 * by applying the steps:
 * (1) Generate a real number between 0 and 1.
 * (2) Scale it to the appropriate range size.
 * (4) Translate it to the appropriate range.
 */
double RandomReal(double low, double high)
{
    double d;

    d = (double) rand() / ( (double) RAND_MAX + 1);
    return (low + d * (high - low));
}

/*
 * Function: RandomChance
 * Usage: if (RandomChance(p)) . . .
 * ---------------------------------
 * The RandomChance function returns true with the probability
 * indicated by p, which should be a floating-point number between
 * 0 (meaning never) and 1 (meaning always).
 */
bool RandomChance(double probability)
{
    return (RandomReal(0,1) < probability);
}

/*
 * Function: RandomNormal
 * Usage: n = RandomNormal(mean, std);
 * -----------------------------------
 * This function returns a random real number following the
 * normal distribution with given mean and standard deviation (std)
 * using the Box-Muller Method.
```

```
112  */
113 double RandomNormal(double mean, double std)
114 {
115     double u1 = RandomReal(0,1);
116     double u2 = RandomReal(0,1);
117 
118     double z = sqrt(-2 * log(u1)) * sin(2 * M_PI * u2);
119 
120     return mean + std*z;
121 }
```