Simulación de Dinámica de Partículas con Colisiones Elásticas

Documentación Técnica de Simulación de Billar 2D

29 de octubre de 2025

Objetivos

El objetivo principal de este proyecto es simular la dinámica de un sistema de partículas circulares 2D (bolas de billar o esferas duras) confinadas en una caja rectangular, utilizando diferentes métodos numéricos de integración y modelando las colisiones elásticas entre partículas y con las paredes.

- Implementar la lógica de movimiento y colisión para una partícula individual (`Bola`).
- Desarrollar una clase que gestione el sistema completo (`Sistema`), permitiendo la elección entre los integradores de Euler y Velocity-Verlet.
- Generar datos de trayectoria para su posterior análisis y visualización (animación, histogramas de velocidad, etc.).

Método Numérico

La simulación se basa en la integración temporal de las ecuaciones de movimiento, donde la fuerza neta sobre las partículas es cero ($\mathbf{F} = 0$), y la dinámica del sistema es determinada por la conservación del momento y la energía durante los eventos de colisión.

Integradores Implementados

El sistema permite seleccionar entre dos métodos de integración para avanzar la posición de las partículas en el tiempo dt:

Método de Euler

Este es el método más simple. Para una fuerza neta nula, solo actualiza la posición:

$$\mathbf{r}(t + dt) = \mathbf{r}(t) + \mathbf{v}(t) \cdot dt$$

Este método se combina con una colisión contra paredes **simple** que solo invierte la velocidad.

Método de Velocity-Verlet

Aunque para $\mathbf{F} = 0$ la ecuación de avance de posición es la misma que la de Euler, este integrador se combina con un manejo de colisión contra paredes **robusto**. El método robusto corrige la posición de la partícula al momento de la colisión para asegurar que nunca penetre la pared, mejorando la estabilidad y la conservación de la energía del sistema.

Análisis Físico y Conservación de Magnitudes

Colisión Elástica entre Partículas

El método clave es `Bola::ChoqueElastico`, el cual modela una colisión elástica. En estas colisiones se cumplen dos principios fundamentales:

1. Conservación del Momento Lineal Total: El momento lineal del sistema de las dos partículas ($\mathbf{p}_1 + \mathbf{p}_2$) se conserva.
2. Conservación de la Energía Cinética Total (E_k): La energía cinética total del sistema se mantiene constante, ya que no hay pérdidas por fricción u otros mecanismos disipativos.

El algoritmo implementado también incluye una corrección de solapamiento (overlap correction) para ajustar las posiciones de las bolas que se hayan penetrado ligeramente debido al paso de tiempo Δt, garantizando la estabilidad de la simulación.

Colisión contra Paredes Robusta

El método `ResuelvaColisionParedesRobusto` implementado para el integrador Verlet es crucial para la estabilidad:

- No solo invierte la componente de la velocidad perpendicular a la pared ($\mathbf{v} \to -\mathbf{v}$), sino que también refleja la posición para evitar que la partícula quede atrapada dentro de la pared, corrigiendo el error de truncamiento del paso temporal.

Validación Básica y Convergencia

La validación de esta simulación se centra en la estabilidad del método y la fidelidad física:

Convergencia Cualitativa

- Se espera que el método **Velocity-Verlet** (`PasoVerlet`) muestre un movimiento más suave y estable, especialmente para valores de Δt más grandes, en comparación con el método de Euler (`PasoEuler`) que puede mostrar inestabilidades.

Conservación de Magnitudes

- Energía Cinética Total: Para validar la simulación, se debe calcular la energía cinética total del sistema ($\sum \frac{1}{2} m_i v_i^2$) a lo largo del tiempo. En una simulación elástica ideal, esta debe ser **constante**. Se espera que el integrador Verlet conserve la energía mucho mejor que el de Euler.
- Momento Lineal Total: El momento lineal total del sistema de partículas también debe conservarse (aunque las colisiones contra las paredes externas introducen un momento impulsivo).

Resultados y Postprocesado

Gráficas y Tablas

La simulación genera el archivo `trayectorias.dat` con los datos de tiempo (`t`), posición (`x`, `y`) y velocidad (`vx`, `vy`) para cada partícula.

- Gráfica de Posiciones (`x` vs. `y`): Se utiliza para visualizar la trayectoria de las partículas y, en el postprocesado, para generar la animación de la simulación.
- Histograma de Velocidades: Para un análisis a largo plazo, el histograma de las magnitudes de velocidad de las partículas debe converger a la distribución de Maxwell-Boltzmann, un resultado fundamental de la física estadística.

Estructura de Carpetas del Proyecto

El proyecto está organizado de manera modular para separar la lógica de simulación, la configuración y el post-procesado (visualización).

- **Carpeta Raíz:** Contiene el archivo de compilación principal (e.g., `Makefile`) y los directorios principales.
- `src/` (Source): Contiene todo el código fuente en C++ (clases y programa principal).
- `scripts/` (Scripts): Contiene archivos para el post-procesado, como scripts de Python o Gnuplot.
- `results/` (Resultados): Directorio donde se almacenan los datos de salida (`trayectorias.dat`).

```
/billar
|-- build/
|-- include/
|   |-- Bola.h
|   |-- Caja.h
|   |-- Sistema.h
|-- scripts/
|   |-- graficar.py
|   |-- graficar.gnuplot
|-- results/
|   |-- trayectorias.dat
```

```
|   |-- gráficas
|-- src/
|   |-- Bola.cpp
|   |-- Caja.cpp
|   |-- Sistema.cpp
|-- CMakeLists.txt
|-- main.cpp
```

Diagrama de Clases

La simulación se basa en un diseño modular con tres clases principales: `Caja`, `Bola` y `Sistema`, las cuales interactúan para gestionar la dinámica.

Relaciones de Clases

- **Composición (`Sistema` a `Caja` y `Bola`):** El objeto `Sistema` contiene (gestiona) la única `Caja` y el conjunto (`std::vector`) de objetos `Bola`.
- **Dependencia (`Bola` a `Caja`):** La clase `Bola` depende de las dimensiones de `Caja` para sus métodos de colisión con las paredes.

Listing 1: Código de Diagrama de Clases (UML Simplificado)

```
classDiagram
    direction LR

    class Caja {
        -W : double (Ancho)
        -H : double (Alto)
        +Defina(W, H)
        +GetW() : double
    }

    class Bola {
        -x, y, vx, vy, m, r : double
        +Muevase(dt)
        +ChoqueElastico(Bola& otra)
        +ResuelvaColisionParedesRobusto(Caja)
    }

    class Sistema {
        -caja : Caja
        -bolas : vector<Bola>
        -integrador_actual : Enum
        +SeleccioneIntegrador(nombre)
        +Paso(dt)
        -PasoEuler(dt)
        -PasoVerlet(dt)
    }

    Caja "1" <-- "1" Bola : Usa en Colisiones
    Sistema "1" --> "1" Caja : Contiene
    Sistema "1" --> "N" Bola : Contiene
    main --|> Sistema : Usa para simular
```

Figura 1: Diagrama de Clases de la Simulación.

Código

A continuación, se presenta el código fuente de las clases principales del sistema.

Archivo `main.cpp`

El programa principal gestiona la interacción con el usuario, configura el sistema, ejecuta el bucle de simulación y coordina el postprocesado.

```cpp
#include <iostream>
#include <iomanip>
#include <filesystem>
#include <string>
#include <stdexcept>
#include "Sistema.h"

int main() {
    Sistema sim;
    const double dt_sim = 0.001, dt_frame = 0.01;
    const double m = 1.0, r = 0.2, vmax = 1.0;
    double tf, W, H;
    int N;
    std::string integrador_nombre;

    // --- User Input ---
    std::cout << "Ingrese el numero de particulas (N): ";
    std::cin >> N;
    std::cout << "Ingrese el tiempo total de simulacion (s): ";
    std::cin >> tf;
    std::cout << "Ingrese el ancho de la caja:";
    std::cin >> W;
    std::cout << "Ingrese el alto de la caja: ";
    std::cin >> H;
    std::cout << "Elija el integrador (euler/verlet): ";
    std::cin >> integrador_nombre;

    // --- Setup Simulation ---
    try {
        sim.SeleccioneIntegrador(integrador_nombre);
    } catch (const std::invalid_argument& e) {
        std::cerr << "Error: " << e.what() << std::endl;
        return 1;
    }

    sim.DefinaCaja(W, H);
    sim.Reserve(N);
    sim.InicialiceRejilla(m, r, vmax);

    // --- Prepare Output File ---
std::filesystem::create_directories("../results");
std::ofstream archivo("../results/trayectorias.dat");
// ... (Verificaci n de apertura)

// **A adir estas l neas para Gnuplot y Python**
archivo << "# W: " << W << "\n";
archivo << "# H: " << H << "\n";
archivo << "# R_BOLA: " << r << "\n";

// --- Simulation Loop ---
std::cout << "Iniciando simulacion con el integrador '" << integrador_nombre << "'..." <<
    std::endl;
double t = 0;
long pasos_por_frame = static_cast<long>(dt_frame / dt_sim);

    sim.Encabezado(archivo); // Write header once

    while (t <= tf) {
        sim.Guarde(archivo, t); // Save current state

        // Advance simulation until the next frame time
```

```
        for(long i = 0; i < pasos_por_frame; ++i) {
            sim.Paso(dt_sim);
        }
        t += dt_frame;
        // Simple progress indicator
        std::cout << "\rProgreso: " << std::fixed << std::setprecision(1) << (t / tf) * 100.0
            << "%" << std::flush;
    }
    std::cout << "\n Simulacion completada. Datos guardados en ../results/trayectorias.dat\n";
    archivo.close();

    // --- Plotting Choice ---
    std::cout << "Generar animacion con (p)ython o (g)nuplot? ";
    char op;
    std::cin >> op;

    if (op == 'p' || op == 'P') {
        std::cout << "Ejecutando script de Python..." << std::endl;
        system("python3 ../scripts/graficar.py");
    } else if (op == 'g' || op == 'G') {
        std::cout << "Ejecutando script de Gnuplot..." << std::endl;
        system("gnuplot -p ../scripts/graficar.gnuplot");
    }

    return 0;
}
```

Archivo `Sistema.cpp`

Implementa la lógica del sistema, gestionando las partículas y despachando la integración al método Euler o Verlet según la selección.

```
#include "Sistema.h"
#include <cstdlib>
#include <cmath>
#include <iostream>
#include <iomanip>
#include <stdexcept> // For std::invalid_argument

void Sistema::SeleccioneIntegrador(const std::string& nombre) {
    if (nombre == "euler") {
        integrador_actual = Integrador::Euler;
    } else if (nombre == "verlet") {
        integrador_actual = Integrador::Verlet;
    } else {
        throw std::invalid_argument("Integrador no valido. Elija 'euler' o 'verlet'.");
    }
}

void Sistema::Paso(double dt) {
    // Dispatch to the correct integration method
    if (integrador_actual == Integrador::Euler) {
        PasoEuler(dt);
    } else {
        PasoVerlet(dt);
    }
}

// --- EULER METHOD (PROTOTYPE) ---
// Simple but physically inaccurate wall collisions.
void Sistema::PasoEuler(double dt) {
    // 1. Move all particles (Euler step)
    for (auto& b : bolas) {
        b.Muevase(dt);
    }
```

```
34
35      // 2. Handle collisions
36      // Walls (simple, inaccurate reflection)
37      for (auto& b : bolas) {
38          b.ResuelvaColisionParedesSimple(caja);
39      }
40      // Between particles
41      for (size_t i = 0; i < bolas.size(); ++i) {
42          for (size_t j = i + 1; j < bolas.size(); ++j) {
43              bolas[i].ChoqueElastico(bolas[j]);
44          }
45      }
46  }
47
48  // --- VELOCITY-VERLET METHOD (STABLE & RECOMMENDED) ---
49  // More robust wall collision handling that prevents sticking.
50  void Sistema::PasoVerlet(double dt) {
51      // 1. Move all particles (same as Euler for F=0)
52      for (auto& b : bolas) {
53          b.Muevase(dt);
54      }
55
56      // 2. Handle collisions
57      // Walls (robust, position-correcting reflection)
58      for (auto& b : bolas) {
59          b.ResuelvaColisionParedesRobusto(caja);
60      }
61      // Between particles
62      for (size_t i = 0; i < bolas.size(); ++i) {
63          for (size_t j = i + 1; j < bolas.size(); ++j) {
64              bolas[i].ChoqueElastico(bolas[j]);
65          }
66      }
67  }
68
69  // --- Other System Methods (Setup and I/O) ---
70  // ... (c digo de setup e I/O)
71  // Note: InitialiceRejilla sets up particles in a grid with random velocities.
72  void Sistema::DefinaCaja(double W, double H) {
73      caja.Defina(W, H);
74  }
75
76  void Sistema::Reserve(int N) {
77      bolas.resize(N);
78  }
79
80  void Sistema::InicialiceRejilla(double m, double r, double vmax, bool alterna) {
81      int N = bolas.size();
82      if (N == 0) return;
83
84      srand(time(nullptr));
85      double W = caja.GetW();
86      double H = caja.GetH();
87
88      int cols = static_cast<int>(std::sqrt(N * W / H));
89      int rows = (N > 0) ? (N + cols - 1) / cols : 0;
90
91      double dx = (cols > 1) ? (W - 2*r) / (cols - 1) : W / 2.0;
92      double dy = (rows > 1) ? (H - 2*r) / (rows - 1) : H / 2.0;
93
94      for (int i = 0; i < N; ++i) {
95          int row = i / cols;
96          int col = i % cols;
97
98          double x0 = (cols > 1) ? r + col * dx : W / 2.0;
```

```
        double y0 = (rows > 1) ? r + row * dy : H / 2.0;

        double ang = 2 * M_PI * ((double)rand() / RAND_MAX);
        double v = vmax * ((double)rand() / RAND_MAX);
        double vx = alterna && (i % 2 != 0) ? -v * cos(ang) : v * cos(ang);
        double vy = alterna && (i % 2 != 0) ? -v * sin(ang) : v * sin(ang);

        bolas[i].Inicie(x0, y0, vx, vy, m, r);
    }
    std::cout << "Inicializacion en rejilla completada con " << N << " bolas.\n";
}

void Sistema::Encabezado(std::ofstream& f) {
    f << "# " << std::setw(9) << "t";
    for (size_t i = 0; i < bolas.size(); i++) {
        f << std::setw(15) << "x" + std::to_string(i)
          << std::setw(15) << "y" + std::to_string(i)
          << std::setw(15) << "vx" + std::to_string(i)
          << std::setw(15) << "vy" + std::to_string(i);
    }
    f << "\n";
}

void Sistema::Guarde(std::ofstream& f, double t) {
    f << std::setw(10) << std::fixed << std::setprecision(4) << t;
    for (const auto& b : bolas) {
        f << std::setw(15) << std::fixed << std::setprecision(6) << b.Getx()
          << std::setw(15) << std::fixed << std::setprecision(6) << b.Gety()
          << std::setw(15) << std::fixed << std::setprecision(6) << b.Getvx()
          << std::setw(15) << std::fixed << std::setprecision(6) << b.Getvy();
    }
    f << "\n";
}
```

Archivo `Bola.cpp`

Contiene la implementación del movimiento de una partícula individual y, crucialmente, los dos métodos de colisión contra paredes y la colisión elástica entre partículas.

```
#include "Bola.h"
#include <cmath>

Bola::Bola() : x(0), y(0), vx(0), vy(0), m(1.0), r(0.1) {}

void Bola::Inicie(double x0, double y0, double vx0, double vy0, double m0, double r0) {
    x = x0; y = y0;
    vx = vx0; vy = vy0;
    m = m0; r = r0;
}

// Just advances position based on velocity. Collisions handled by Sistema.
void Bola::Muevase(double dt) {
    x += vx * dt;
    y += vy * dt;
}

// Simple collision: just inverts velocity. Can cause sticking.
void Bola::ResuelvaColisionParedesSimple(const Caja& C) {
    if (x - r < 0 && vx < 0) vx *= -1;
    if (x + r > C.GetW() && vx > 0) vx *= -1;
    if (y - r < 0 && vy < 0) vy *= -1;
    if (y + r > C.GetH() && vy > 0) vy *= -1;
}

// Robust collision: corrects position to avoid passing through the wall.
```

```cpp
void Bola::ResuelvaColisionParedesRobusto(const Caja& C) {
    if (x - r < 0 && vx < 0) {
        x = r + (r - x); // Reflect position
        vx *= -1;
    }
    if (x + r > C.GetW() && vx > 0) {
        x = C.GetW() - r - (x + r - C.GetW()); // Reflect position
        vx *= -1;
    }
    if (y - r < 0 && vy < 0) {
        y = r + (r - y); // Reflect position
        vy *= -1;
    }
    if (y + r > C.GetH() && vy > 0) {
        y = C.GetH() - r - (y + r - C.GetH()); // Reflect position
        vy *= -1;
    }
}

// Elastic collision between two balls (unchanged from your original)
void Bola::ChoqueElastico(Bola& otra) {
    double dx = otra.x - x;
    double dy = otra.y - y;
    double dist_sq = dx * dx + dy * dy;
    double minDist = r + otra.r;

    if (dist_sq < minDist * minDist) {
        double dist = std::sqrt(dist_sq);
        // Evita divisi n por cero si est n exactamente en el mismo punto
        if (dist == 0.0) return;

        double nx = dx / dist;
        double ny = dy / dist;

        double dvx = otra.vx - vx;
        double dvy = otra.vy - vy;
        double vn = dvx * nx + dvy * ny;

        // Only apply collision if they are moving towards each other
        if (vn < 0) {
            double J = (-2 * vn) / (1/m + 1/otra.m);
            vx -= (J / m) * nx;
            vy -= (J / m) * ny;
            otra.vx += (J / otra.m) * nx;
            otra.vy += (J / otra.m) * ny;
        }

        // Overlap correction
        double overlap = 0.51 * (minDist - dist);
        x -= overlap * nx;
        y -= overlap * ny;
        otra.x += overlap * nx;
        otra.y += overlap * ny;
    }
}
```

Discusión y Conclusiones

Esta sección se completará una vez se hayan obtenido los resultados de la simulación, pero se deben abordar los siguientes puntos:

- Comparación de Integradores: Discutir la diferencia en la conservación de energía entre Euler y Velocity-Verlet, y concluir por qué el método robusto de Verlet es preferible para sistemas con colisiones de esferas duras.
- Análisis de Resultados Físicos:

- Evaluar si la distribución de velocidades obtenida (histograma) se aproxima a la distribución de Maxwell-Boltzmann, validando así que el sistema alcanza el equilibrio termodinámico.
- Comentar sobre la estabilidad del sistema a largo plazo y la efectividad de la corrección de solapamiento en las colisiones entre partículas.