**Implementation of Coordinate System Transformations**

**Objective:** To familiarize oneself with different coordinate systems (Cartesian, polar, and spherical) and gain practical skills in transitioning between them. To determine the computational efficiency of distance calculations in these coordinate systems through benchmarking.

**Solution:**

**Tasks**

**1) Transition Between Coordinate Systems:**

Two-Dimensional Space: Cartesian and Polar Coordinate Systems

* Define the coordinates of several points in the polar coordinate system.
* Convert these coordinates to the Cartesian coordinate system.
* Perform the reverse conversion from the Cartesian coordinate system to the polar coordinate system.
* Verify the correctness of the calculations by ensuring that the initial coordinates match the ones obtained after the reverse conversion.

Three-Dimensional Space: Cartesian and Spherical Coordinate Systems

* Define the coordinates of several points in the spherical coordinate system.
* Convert these coordinates to the Cartesian coordinate system.
* Perform the reverse conversion from the Cartesian coordinate system to the spherical coordinate system.
* Verify the correctness of the calculations by ensuring that the initial coordinates match the ones obtained after the reverse conversion.

**Code listing and result:**

**2D**

#include <iostream>

#include <cmath>

#include <vector>

#include <tuple>

#include <utility>

#define PI 3.14159265358979323846

// Coordinate conversion functions

std::pair<double, double> polar\_to\_cartesian(double r, double theta) {

double x = r \* cos(theta);

double y = r \* sin(theta);

return { x, y };

}

std::pair<double, double> cartesian\_to\_polar(double x, double y) {

double r = sqrt(x \* x + y \* y);

double theta = atan2(y, x);

return { r, theta };

}

int main() {

// Sample points for conversion verification

std::vector<std::pair<double, double>> polar\_points = { {5, PI / 4}, {10, PI / 3}, {7, PI / 6} };

// 2D Conversion Verification

for (const auto& p : polar\_points) {

double r = p.first;

double theta = p.second;

// Use std::pair to hold the returned values

std::pair<double, double> cartesian\_coords = polar\_to\_cartesian(r, theta);

std::pair<double, double> polar\_coords = cartesian\_to\_polar(cartesian\_coords.first, cartesian\_coords.second);

// Access elements through .first and .second

std::cout << "Original (r, theta): (" << r << ", " << theta << ") -> Cartesian (x, y): ("

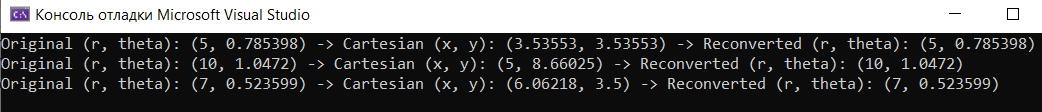
<< cartesian\_coords.first << ", " << cartesian\_coords.second << ") -> Reconverted (r, theta): ("

<< polar\_coords.first << ", " << polar\_coords.second << ")\n";

}

return 0;

}



**3D**

#include <iostream>

#include <cmath> // For math functions like sin, cos, sqrt, atan2, acos

#include <vector> // For std::vector

#include <tuple> // For std::tuple

#include <utility> // For std::pair

#include <iomanip> // For std::setprecision

#define PI 3.14159265358979323846

// Coordinate conversion functions

std::tuple<double, double, double> spherical\_to\_cartesian(double rho, double theta, double phi) {

double x = rho \* sin(phi) \* cos(theta);

double y = rho \* sin(phi) \* sin(theta);

double z = rho \* cos(phi);

return { x, y, z };

}

std::tuple<double, double, double> cartesian\_to\_spherical(double x, double y, double z) {

double rho = sqrt(x \* x + y \* y + z \* z);

double theta = atan2(y, x);

double phi = acos(z / rho);

return { rho, theta, phi };

}

int main() {

// Sample points for conversion verification

std::vector<std::tuple<double, double, double>> spherical\_points = {

{5, PI / 4, PI / 3},

{10, PI / 3, PI / 6},

{7, PI / 6, PI / 4}

};

// 3D Conversion Verification

for (const auto& p : spherical\_points) {

double rho, theta, phi;

std::tie(rho, theta, phi) = p; // Unpack spherical coordinates

// Convert spherical to Cartesian

double x, y, z; // Declare Cartesian coordinates

std::tie(x, y, z) = spherical\_to\_cartesian(rho, theta, phi);

// Convert back from Cartesian to spherical

double rho\_new, theta\_new, phi\_new; // Declare new spherical coordinates

std::tie(rho\_new, theta\_new, phi\_new) = cartesian\_to\_spherical(x, y, z);

// Output the results

std::cout << std::fixed << std::setprecision(4); // Format output to 4 decimal places

std::cout << "Original (rho, theta, phi): (" << rho << ", " << theta << ", " << phi << ") -> Cartesian (x, y, z): ("

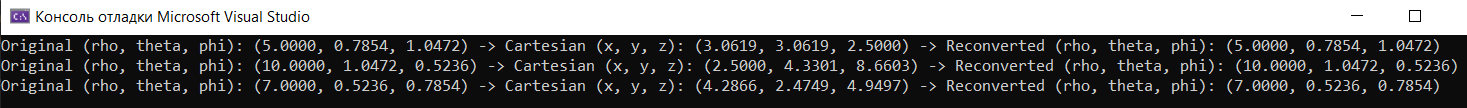
<< x << ", " << y << ", " << z << ") -> Reconverted (rho, theta, phi): ("

<< rho\_new << ", " << theta\_new << ", " << phi\_new << ")\n";

}

return 0;

}



**2) Distance Calculation in the Spherical Coordinate System:**

Compute the distance between points in the spherical coordinate system using two methods:

* Cartesian Coordinate System: Use the standard formula for calculating straight-line distance in two-dimensional and three-dimensional spaces.
* Polar Coordinate System: Use the formula for calculating the distance between points in a two-dimensional space.
* Spherical Coordinate System: Compute the distance between points using two methods:

1. Through the volume of the sphere: Use the formula for straight-line distance in three-dimensional space.
2. Along the surface of the sphere: Use the formula for great-circle distance.

**Code listing and result:**

#include <iostream>

#include <cmath> // Required for mathematical functions

#include <vector> // For std::vector

#include <tuple> // For std::tuple

#include <iomanip> // For std::setprecision

#define PI 3.14159265358979323846

// Function to convert spherical coordinates to Cartesian coordinates

std::tuple<double, double, double> spherical\_to\_cartesian(double rho, double theta, double phi) {

double x = rho \* sin(phi) \* cos(theta);

double y = rho \* sin(phi) \* sin(theta);

double z = rho \* cos(phi);

return { x, y, z };

}

// Function to calculate the straight-line distance in Cartesian coordinates

double cartesian\_distance(double x1, double y1, double z1, double x2, double y2, double z2) {

return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2) + pow(z2 - z1, 2));

}

// Function to compute the distance between two points in spherical coordinates

double spherical\_distance(double rho1, double theta1, double phi1, double rho2, double theta2, double phi2) {

// Convert spherical to Cartesian

double x1, y1, z1; // Declare Cartesian coordinates for first point

std::tie(x1, y1, z1) = spherical\_to\_cartesian(rho1, theta1, phi1);

double x2, y2, z2; // Declare Cartesian coordinates for second point

std::tie(x2, y2, z2) = spherical\_to\_cartesian(rho2, theta2, phi2);

// Calculate and return the distance

return cartesian\_distance(x1, y1, z1, x2, y2, z2);

}

int main() {

// Example points in spherical coordinates

double rho1 = 5.0, theta1 = PI / 4, phi1 = PI / 3; // Point 1

double rho2 = 10.0, theta2 = PI / 3, phi2 = PI / 6; // Point 2

// Calculate distance between the two points in spherical coordinates

double distance = spherical\_distance(rho1, theta1, phi1, rho2, theta2, phi2);

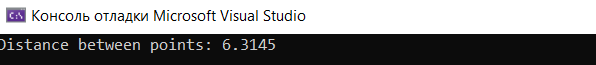
// Output the result

std::cout << std::fixed << std::setprecision(4); // Format output to 4 decimal places

std::cout << "Distance between points: " << distance << "\n";

return 0;

}



3) Performance Benchmarks:

- Generate an array of coordinate pairs in each coordinate system (Cartesian, polar, spherical).

- Compute the distances between these points for each coordinate system.

- Measure the computation time for each coordinate system.

- Choose an array size that results in minimal variability in benchmarking results between runs (a recommended array size is 10,000 to 100,000 points).

**Code listing and result:**

#include <iostream>

#include <cmath>

#include <vector>

#include <tuple>

#include <iomanip>

#include <chrono> // For timing

#include <cstdlib> // For random number generation

#define PI 3.14159265358979323846

// Function prototypes

std::tuple<double, double, double> spherical\_to\_cartesian(double rho, double theta, double phi);

double cartesian\_distance\_3d(double x1, double y1, double z1, double x2, double y2, double z2);

double great\_circle\_distance(double rho1, double theta1, double phi1, double rho2, double theta2, double phi2);

double spherical\_distance(double rho1, double theta1, double phi1, double rho2, double theta2, double phi2);

void generate\_random\_points(int n, std::vector<std::tuple<double, double, double>>& points\_spherical);

// Function to convert spherical coordinates to Cartesian coordinates

std::tuple<double, double, double> spherical\_to\_cartesian(double rho, double theta, double phi) {

double x = rho \* sin(phi) \* cos(theta);

double y = rho \* sin(phi) \* sin(theta);

double z = rho \* cos(phi);

return { x, y, z };

}

// Function to calculate the straight-line distance in Cartesian coordinates

double cartesian\_distance\_3d(double x1, double y1, double z1, double x2, double y2, double z2) {

return sqrt(pow(x2 - x1, 2) + pow(y2 - y1, 2) + pow(z2 - z1, 2));

}

// Function to compute the distance between two points in spherical coordinates

double spherical\_distance(double rho1, double theta1, double phi1, double rho2, double theta2, double phi2) {

// Convert spherical to Cartesian

double x1, y1, z1;

std::tie(x1, y1, z1) = spherical\_to\_cartesian(rho1, theta1, phi1);

double x2, y2, z2;

std::tie(x2, y2, z2) = spherical\_to\_cartesian(rho2, theta2, phi2);

// Calculate and return the distance

return cartesian\_distance\_3d(x1, y1, z1, x2, y2, z2);

}

// Function to calculate the great-circle distance

double great\_circle\_distance(double rho1, double theta1, double phi1, double rho2, double theta2, double phi2) {

double delta\_theta = theta2 - theta1;

double delta\_phi = phi2 - phi1;

// Using the haversine formula

double a = sin(delta\_phi / 2) \* sin(delta\_phi / 2) +

cos(phi1) \* cos(phi2) \*

sin(delta\_theta / 2) \* sin(delta\_theta / 2);

double c = 2 \* atan2(sqrt(a), sqrt(1 - a));

return rho1 \* c; // Assuming both points are on the same sphere

}

// Function to generate random spherical points

void generate\_random\_points(int n, std::vector<std::tuple<double, double, double>>& points\_spherical) {

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

double rho = static\_cast<double>(rand() % 100 + 1); // Random radius between 1 and 100

double theta = static\_cast<double>(rand()) / RAND\_MAX \* 2 \* PI; // Random theta between 0 and 2π

double phi = static\_cast<double>(rand()) / RAND\_MAX \* PI; // Random phi between 0 and π

points\_spherical.emplace\_back(rho, theta, phi);

}

}

int main() {

int num\_points = 10000; // Number of random points

std::vector<std::tuple<double, double, double>> points\_spherical;

generate\_random\_points(num\_points, points\_spherical);

// Benchmarking distance calculations

auto start = std::chrono::high\_resolution\_clock::now();

for (size\_t i = 0; i < points\_spherical.size() - 1; ++i) {

double rho1, theta1, phi1;

std::tie(rho1, theta1, phi1) = points\_spherical[i];

double rho2, theta2, phi2;

std::tie(rho2, theta2, phi2) = points\_spherical[i + 1];

double dis = spherical\_distance(rho1, theta1, phi1, rho2, theta2, phi2);

}

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

std::cout << std::fixed << std::setprecision(4);

std::cout << "Time taken for spherical distance calculation: " << elapsed.count() << " seconds\n";

start = std::chrono::high\_resolution\_clock::now();

for (size\_t i = 0; i < points\_spherical.size() - 1; ++i) {

double rho1, theta1, phi1;

std::tie(rho1, theta1, phi1) = points\_spherical[i];

double rho2, theta2, phi2;

std::tie(rho2, theta2, phi2) = points\_spherical[i + 1];

double dis = great\_circle\_distance(rho1, theta1, phi1, rho2, theta2, phi2);

}

end = std::chrono::high\_resolution\_clock::now();

elapsed = end - start;

std::cout << std::fixed << std::setprecision(4);

std::cout << "Time taken for great-circle distance calculation: " << elapsed.count() << " seconds\n";

return 0;

}

