**Skyhook**

#include <iostream>

#include <cmath>

#include <memory>

#include <vector>

#include <chrono>

#include <thread>

#include <queue>

#include <mutex>

#include <random>

#include <shared\_mutex>

#include <onnxruntime\_cxx\_api.h>

#include <algorithm> // 添加缺失的头文件

// 统一物理常数命名空间

namespace AstroConstants {

// 基础轨道参数

constexpr double G = 6.67430e-11; // 万有引力常数 [m³/kg/s²]

constexpr double EarthRadius = 6371e3; // 地球半径 [m]

constexpr double EarthMass = 5.972e24; // 地球质量 [kg]

constexpr double J2 = 1.08262668e-3; // 地球扁率摄动项

// 增强参数

constexpr double c = 299792458; // 光速 [m/s]

constexpr double EarthGeoLatency = 0.12; // 地空单程时延 [s]

constexpr double QubitErrorRate = 1e-5; // 量子比特错误率

constexpr int QuantumRedundancy = 3; // 量子冗余度

}

// 增强型三维矢量（带相对论时间戳）

struct Vector3D {

double x, y, z;

std::chrono::system\_clock::time\_point rel\_time;

Vector3D operator+(const Vector3D& other) const {

return {

x + other.x,

y + other.y,

z + other.z,

std::max(rel\_time, other.rel\_time)

};

}

Vector3D operator-(const Vector3D& other) const {

return {

x - other.x,

y - other.y,

z - other.z,

std::max(rel\_time, other.rel\_time)

};

}

Vector3D operator\*(double scalar) const {

return {x\*scalar, y\*scalar, z\*scalar, rel\_time};

}

double magnitude() const {

return std::hypot(x, y, z);

}

};

// 量子增强轨道状态

struct OrbitalState {

Vector3D position; // 地心惯性坐标系 [m]

Vector3D velocity; // 地心惯性坐标系 [m/s]

double mass; // 航天器质量 [kg]

double fuel; // 剩余燃料 [kg]

double quantum\_fidelity; // 量子态保真度 [0-1]

uint32\_t error\_correction; // 纠错码状态

};

// 高精度轨道力学计算器

class OrbitalCalculator {

public:

// 含J2摄动和相对论修正的引力计算

static Vector3D calculate\_gravity(const Vector3D& pos) {

const double r = std::max(pos.magnitude(), 1e3); // 防止极小值

const double k = AstroConstants::G \* AstroConstants::EarthMass / std::pow(r, 3);

// J2摄动项

const double z2 = pos.z \* pos.z;

const double J2\_term = 1.5 \* AstroConstants::J2 \*

std::pow(AstroConstants::EarthRadius/r, 2) \*

(1 - 5\*z2/(r\*r));

// 一阶后牛顿修正

const double rel\_term = 1 + (AstroConstants::G \* AstroConstants::EarthMass) /

(AstroConstants::c \* AstroConstants::c \* r);

return {

-pos.x \* k \* (1 + J2\_term) \* rel\_term,

-pos.y \* k \* (1 + J2\_term) \* rel\_term,

-pos.z \* k \* (1 + 3\*J2\_term) \* rel\_term,

pos.rel\_time

};

}

// 四阶Runge-Kutta轨道传播

static void propagate\_orbit(OrbitalState& state, double dt) {

const auto& pos = state.position;

const auto& vel = state.velocity;

const Vector3D k1\_v = calculate\_gravity(pos);

const Vector3D k1\_p = vel;

const Vector3D k2\_v = calculate\_gravity(pos + k1\_p\*(dt/2));

const Vector3D k2\_p = vel + k1\_v\*(dt/2);

const Vector3D k3\_v = calculate\_gravity(pos + k2\_p\*(dt/2));

const Vector3D k3\_p = vel + k2\_v\*(dt/2);

const Vector3D k4\_v = calculate\_gravity(pos + k3\_p\*dt);

const Vector3D k4\_p = vel + k3\_v\*dt;

state.velocity = vel + (k1\_v + k2\_v\*2 + k3\_v\*2 + k4\_v)\*(dt/6);

state.position = pos + (k1\_p + k2\_p\*2 + k3\_p\*2 + k4\_p)\*(dt/6);

}

};

// 量子容错导航系统

class QuantumNavSystem {

mutable std::shared\_mutex data\_mutex;

std::array<OrbitalState, AstroConstants::QuantumRedundancy> qubit\_states;

public:

void update\_state(const OrbitalState& raw\_state) {

std::unique\_lock lock(data\_mutex);

for(auto& s : qubit\_states) {

s = apply\_quantum\_correction(raw\_state);

}

}

OrbitalState get\_best\_estimate() {

std::shared\_lock lock(data\_mutex);

OrbitalState result = qubit\_states[0];

for(size\_t i=1; i<qubit\_states.size(); ++i) {

result.position = result.position + qubit\_states[i].position;

result.velocity = result.velocity + qubit\_states[i].velocity;

}

result.position = result.position \* (1.0/qubit\_states.size());

result.velocity = result.velocity \* (1.0/qubit\_states.size());

result.quantum\_fidelity = calculate\_fidelity();

return result;

}

private:

OrbitalState apply\_quantum\_correction(const OrbitalState& state) {

OrbitalState corrected = state;

// 表面-17量子纠错码模拟 (17-qubit surface code)

corrected.error\_correction = (state.error\_correction & 0x1FF) | 0xE00;

corrected.quantum\_fidelity \*= 0.98;

return corrected;

}

double calculate\_fidelity() const {

double sum = 0;

for(const auto& s : qubit\_states) {

sum += s.quantum\_fidelity;

}

return sum / qubit\_states.size();

}

};

// AI-经典混合制导控制器

class HybridGuidance {

Ort::Env onnx\_env;

Ort::Session ai\_session;

// ONNX模型参数

const std::vector<int64\_t> input\_shape = {1, 12};

const char\* input\_names[1] = {"input"};

const char\* output\_names[1] = {"output"};

public:

HybridGuidance()

: onnx\_env(ORT\_LOGGING\_LEVEL\_WARNING, "SpaceDockingAI"),

ai\_session(onnx\_env, "docking\_model.onnx", Ort::SessionOptions{}) {}

~HybridGuidance() {

Ort::GetApi().ReleaseSession(ai\_session); // 显式释放资源

}

Vector3D calculate\_maneuver(const OrbitalState& current,

const OrbitalState& target) {

if(current.quantum\_fidelity > 0.95) {

return ai\_guidance(current, target);

}

return classical\_guidance(current, target);

}

private:

Vector3D ai\_guidance(const OrbitalState& current, const OrbitalState& target) {

std::vector<float> input\_tensor = prepare\_input(current, target);

Ort::MemoryInfo memory\_info = Ort::MemoryInfo::CreateCpu(

OrtAllocatorType::OrtArenaAllocator, OrtMemType::OrtMemTypeDefault);

std::vector<Ort::Value> inputs;

inputs.emplace\_back(Ort::Value::CreateTensor<float>(

memory\_info, input\_tensor.data(), input\_tensor.size(),

input\_shape.data(), input\_shape.size()));

auto outputs = ai\_session.Run(Ort::RunOptions{nullptr},

input\_names, inputs.data(),

inputs.size(), output\_names, 1);

float\* data = outputs[0].GetTensorMutableData<float>();

return {data[0], data[1], data[2]};

}

Vector3D classical\_guidance(const OrbitalState& current,

const OrbitalState& target) {

Vector3D pos\_err = target.position - current.position;

Vector3D vel\_err = target.velocity - current.velocity;

return (pos\_err\*0.3 + vel\_err\*0.7).magnitude() > 1e-3 ?

(pos\_err\*0.3 + vel\_err\*0.7) : Vector3D{0,0,0};

}

std::vector<float> prepare\_input(const OrbitalState& curr,

const OrbitalState& tgt) {

return {

static\_cast<float>(curr.position.x),

static\_cast<float>(curr.position.y),

static\_cast<float>(curr.position.z),

static\_cast<float>(curr.velocity.x),

static\_cast<float>(curr.velocity.y),

static\_cast<float>(curr.velocity.z),

static\_cast<float>(tgt.position.x - curr.position.x),

static\_cast<float>(tgt.position.y - curr.position.y),

static\_cast<float>(tgt.position.z - curr.position.z),

static\_cast<float>(curr.mass),

static\_cast<float>(curr.fuel),

static\_cast<float>(curr.quantum\_fidelity)

};

}

};

// 天地协同云控制器

class CloudControl {

std::atomic\_bool cloud\_enabled{true};

std::queue<Vector3D> cmd\_queue;

mutable std::shared\_mutex queue\_mutex;

public:

void enable(bool flag) { cloud\_enabled.store(flag); }

void upload\_telemetry(const OrbitalState& state) {

if(!cloud\_enabled) return;

std::this\_thread::sleep\_for(

std::chrono::milliseconds(

static\_cast<int>(AstroConstants::EarthGeoLatency \* 1000)));

}

void push\_command(const Vector3D& cmd) {

std::unique\_lock lock(queue\_mutex);

cmd\_queue.push(cmd);

}

std::optional<Vector3D> fetch\_command() {

std::shared\_lock lock(queue\_mutex);

if(cmd\_queue.empty()) return std::nullopt;

auto cmd = cmd\_queue.front();

cmd\_queue.pop();

return cmd\_time\_compensation(cmd);

}

private:

Vector3D cmd\_time\_compensation(const Vector3D& cmd) {

auto now = std::chrono::system\_clock::now();

auto delay = std::chrono::duration\_cast<std::chrono::milliseconds>(

now - cmd.rel\_time).count();

double factor = std::clamp(1.0 + delay\*0.0005, 0.95, 1.05);

return cmd \* factor;

}

};

// 主控制系统

class DockingMaster {

QuantumNavSystem nav\_system;

HybridGuidance guidance;

CloudControl cloud\_ctl;

OrbitalState spacecraft;

OrbitalState skyhook;

std::atomic\_bool running{false}; // 控制线程运行状态

constexpr static double CONTROL\_DT = 0.1; // 100ms控制周期

constexpr static double DOCK\_TOL = 0.5; // 0.5米对接精度

public:

DockingMaster(const OrbitalState& sc, const OrbitalState& sh)

: spacecraft(sc), skyhook(sh) {}

void run\_docking\_sequence() {

running = true;

auto next\_cycle = std::chrono::steady\_clock::now();

while(running && !check\_docking()) {

// 量子导航更新

nav\_system.update\_state(spacecraft);

OrbitalState est\_state = nav\_system.get\_best\_estimate();

// 获取控制指令

std::optional<Vector3D> cmd = cloud\_ctl.fetch\_command();

if(!cmd.has\_value()) {

cmd = guidance.calculate\_maneuver(est\_state, skyhook);

}

// 执行轨道机动

if(cmd.has\_value()) {

execute\_maneuver(cmd.value());

}

// 轨道状态传播

OrbitalCalculator::propagate\_orbit(spacecraft, CONTROL\_DT);

// 天地数据同步

cloud\_ctl.upload\_telemetry(spacecraft);

// 严格周期控制

next\_cycle += std::chrono::milliseconds(100);

std::this\_thread::sleep\_until(next\_cycle);

}

running = false;

std::cout << "量子增强对接完成！位置误差: "

}

void stop() { running = false; }

private:

void execute\_maneuver(const Vector3D& dir) {

if(spacecraft.fuel <= 0) return;

const double isp = 300.0; // 比冲300秒

const double thrust = 1000.0; // 推力1000N

const double delta\_v = dir.magnitude();

const double burn\_time = (delta\_v \* spacecraft.mass) / (isp \* AstroConstants::G);

const double fuel\_flow = thrust / (isp \* AstroConstants::G);

if(burn\_time \* fuel\_flow > spacecraft.fuel) {

std::cerr << "燃料不足，无法完成机动！" << std::endl;

return;

}

spacecraft.velocity = spacecraft.velocity + dir;

spacecraft.fuel -= fuel\_flow \* burn\_time;

spacecraft.mass -= fuel\_flow \* burn\_time;

}

bool check\_docking() const {

return position\_error() < DOCK\_TOL &&

velocity\_difference() < 0.1;

}

double position\_error() const {

return (spacecraft.position - skyhook.position).magnitude();

}

double velocity\_difference() const {

return (spacecraft.velocity - skyhook.velocity).magnitude();

}

};

// 模拟地面站

void ground\_station\_sim(DockingMaster& ctrl) {

std::random\_device rd;

std::mt19937 gen(rd());

std::normal\_distribution<> dist(0.0, 0.05);

while(ctrl.running) { // 使用原子标志控制循环

Vector3D cmd {

dist(gen),

dist(gen),

dist(gen),

std::chrono::system\_clock::now()

};

ctrl.cloud\_ctl.push\_command(cmd);

std::this\_thread::sleep\_for(std::chrono::seconds(1));

}

}

int main() {

// 初始化轨道参数

OrbitalState skyhook = {

{{AstroConstants::EarthRadius + 400e3, 0, 0}, std::chrono::system\_clock::now()},

{{0, 7670, 0}, std::chrono::system\_clock::now()},

20000, 0, 1.0, 0xE1F

};

OrbitalState spacecraft = {

{{AstroConstants::EarthRadius + 350e3, 1000, 0}, std::chrono::system\_clock::now()},

{{0, 7600, 0}, std::chrono::system\_clock::now()},

5000, 150, 0.97, 0xE1F

};

DockingMaster controller(spacecraft, skyhook);

std::thread gs\_thread(ground\_station\_sim, std::ref(controller));

controller.run\_docking\_sequence();

gs\_thread.join();

return 0;

}