// UQFFBuoyancyModule.h

// Modular C++ implementation of the full Master Unified Field Equation (F\_U\_Bi\_i & UQFF Integration) for Buoyancy Equations across M74, Eagle Nebula (M16), M84, Centaurus A, Supernova Survey.

// This module can be plugged into a base program (e.g., 'uqff\_buoyancy\_sim.cpp') by including this header and linking the .cpp.

// Usage in base: #include "UQFFBuoyancyModule.h"

// UQFFBuoyancyModule mod; mod.computeFBi(system); mod.updateVariable("F\_rel", {new\_real, new\_imag});

// All variables are stored in a std::map for dynamic addition/subtraction/update, using complex<double> for real/imaginary components.

// Nothing is negligible: Includes all terms - base force, momentum, gravity, vacuum stability, LENR resonance, activation, directed energy, magnetic resonance, neutron, relativistic, neutrino, Sweet vac, Kozima drop.

// Associated text: Outputs descriptive equation string via getEquationText().

// Approximations: Integral approximated as integrand \* x2 (quadratic root); imag parts small and not fully scaled; LENR dominant due to low ω\_0; x2 from quadratic solver approx; F\_rel from 1998 LEP.

// Multi-system params: M74 M=7.17e41 kg r=9.46e20 m; Eagle M16 M=1e36 kg r=2.36e17 m; M84 M=1.46e45 kg r=3.09e22 m; Centaurus A M=4e41 kg r=3.09e21 m; Supernova Survey (generic M=1e30 kg r=1e10 m).

// Watermark: Copyright - Daniel T. Murphy, analyzed Oct 22, 2025.

#ifndef UQFF\_BUOYANCY\_MODULE\_H

#define UQFF\_BUOYANCY\_MODULE\_H

#include <map>

#include <string>

#include <cmath>

#include <iostream>

#include <iomanip>

#include <complex>

using cdouble = std::complex<double>;

class UQFFBuoyancyModule {

private:

    std::map<std::string, cdouble> variables;

    cdouble computeIntegrand(double t, const std::string& system);

    cdouble computeDPM\_resonance(const std::string& system);

    cdouble computeX2(const std::string& system);

    cdouble computeQuadraticRoot(cdouble a, cdouble b, cdouble c);

    cdouble computeLENRTerm(const std::string& system);

    double computeG(double t, const std::string& system);

    cdouble computeQ\_wave(double t, const std::string& system);

    cdouble computeUb1(const std::string& system);

    cdouble computeUi(double t, const std::string& system);

    void setSystemParams(const std::string& system);

public:

    // Constructor: Initialize all variables with multi-system defaults

    UQFFBuoyancyModule();

    // Dynamic variable operations (complex)

    void updateVariable(const std::string& name, cdouble value);

    void addToVariable(const std::string& name, cdouble delta);

    void subtractFromVariable(const std::string& name, cdouble delta);

    // Core computation: Full F\_U\_Bi\_i(r, t) for system (approx integral)

    cdouble computeFBi(const std::string& system, double t);

    // Sub-equations

    cdouble computeCompressed(const std::string& system, double t);  // Integrand

    cdouble computeResonant(const std::string& system);

    cdouble computeBuoyancy(const std::string& system);

    cdouble computeSuperconductive(const std::string& system, double t);

    double computeCompressedG(const std::string& system, double t);  // g(r,t)

    // Output descriptive text of the equation

    std::string getEquationText(const std::string& system);

    // Print all current variables (for debugging/updates)

    void printVariables();

};

#endif // UQFF\_BUOYANCY\_MODULE\_H

// SurfaceMagneticFieldModule.cpp

#include "SurfaceMagneticFieldModule.h"

// Compute B\_s minimum (quiet Sun)

double SurfaceMagneticFieldModule::computeB\_s\_min() {

    return variables["B\_s\_min"];

}

// Compute B\_s maximum (sunspot max)

double SurfaceMagneticFieldModule::computeB\_s\_max() {

    return variables["B\_s\_max"];

}

// Compute scaled B\_j (T)

double SurfaceMagneticFieldModule::computeB\_j(double t, double B\_s) {

    variables["t"] = t;

    double base\_b = variables["B\_ref"] + 0.4 \* std::sin(variables["omega\_s"] \* t);  // Hypothetical cycle

    return base\_b \* (B\_s / variables["B\_ref"]);

}

// Update variable

void SurfaceMagneticFieldModule::updateVariable(const std::string& name, double value) {

    if (variables.find(name) != variables.end()) {

        variables[name] = value;

    } else {

        std::cerr << "Variable '" << name << "' not found. Adding with value " << value << std::endl;

        variables[name] = value;

    }

}

// UQFFBuoyancyModule.h

// Modular C++ implementation of the full Master Unified Field Equation (F\_U\_Bi\_i & UQFF Integration) for Buoyancy Equations across M74, Eagle Nebula (M16), M84, Centaurus A, Supernova Survey.

// This module can be plugged into a base program (e.g., 'uqff\_buoyancy\_sim.cpp') by including this header and linking the .cpp.

// Usage in base: #include "UQFFBuoyancyModule.h"

// UQFFBuoyancyModule mod; mod.computeFBi(system); mod.updateVariable("F\_rel", {new\_real, new\_imag});

// All variables are stored in a std::map for dynamic addition/subtraction/update, using complex<double> for real/imaginary components.

// Nothing is negligible: Includes all terms - base force, momentum, gravity, vacuum stability, LENR resonance, activation, directed energy, magnetic resonance, neutron, relativistic, neutrino, Sweet vac, Kozima drop.

// Associated text: Outputs descriptive equation string via getEquationText().

// Approximations: Integral approximated as integrand \* x2 (quadratic root); imag parts small and not fully scaled; LENR dominant due to low ω\_0; x2 from quadratic solver approx; F\_rel from 1998 LEP.

// Multi-system params: M74 M=7.17e41 kg r=9.46e20 m; Eagle M16 M=1e36 kg r=2.36e17 m; M84 M=1.46e45 kg r=3.09e22 m; Centaurus A M=4e41 kg r=3.09e21 m; Supernova Survey (generic M=1e30 kg r=1e10 m).

// Watermark: Copyright - Daniel T. Murphy, analyzed Oct 22, 2025.

#ifndef UQFF\_BUOYANCY\_MODULE\_H

#define UQFF\_BUOYANCY\_MODULE\_H

#include <map>

#include <string>

#include <cmath>

#include <iostream>

#include <iomanip>

#include <complex>

using cdouble = std::complex<double>;

class UQFFBuoyancyModule {

private:

    std::map<std::string, cdouble> variables;

    cdouble computeIntegrand(double t, const std::string& system);

    cdouble computeDPM\_resonance(const std::string& system);

    cdouble computeX2(const std::string& system);

    cdouble computeQuadraticRoot(cdouble a, cdouble b, cdouble c);

    cdouble computeLENRTerm(const std::string& system);

    double computeG(double t, const std::string& system);

    cdouble computeQ\_wave(double t, const std::string& system);

    cdouble computeUb1(const std::string& system);

    cdouble computeUi(double t, const std::string& system);

    void setSystemParams(const std::string& system);

public:

    // Constructor: Initialize all variables with multi-system defaults

    UQFFBuoyancyModule();

    // Dynamic variable operations (complex)

    void updateVariable(const std::string& name, cdouble value);

    void addToVariable(const std::string& name, cdouble delta);

    void subtractFromVariable(const std::string& name, cdouble delta);

    // Core computation: Full F\_U\_Bi\_i(r, t) for system (approx integral)

    cdouble computeFBi(const std::string& system, double t);

    // Sub-equations

    cdouble computeCompressed(const std::string& system, double t);  // Integrand

    cdouble computeResonant(const std::string& system);

    cdouble computeBuoyancy(const std::string& system);

    cdouble computeSuperconductive(const std::string& system, double t);

    double computeCompressedG(const std::string& system, double t);  // g(r,t)

    // Output descriptive text of the equation

    std::string getEquationText(const std::string& system, double t);

    // Print all current variables (for debugging/updates)

    void printVariables();

};

#endif // UQFF\_BUOYANCY\_MODULE\_H

// SurfaceMagneticFieldModule.cpp

#include "SurfaceMagneticFieldModule.h"

// Compute B\_s minimum (quiet Sun)

double SurfaceMagneticFieldModule::computeB\_s\_min() {

    return variables["B\_s\_min"];

}

// Compute B\_s maximum (sunspot max)

double SurfaceMagneticFieldModule::computeB\_s\_max() {

    return variables["B\_s\_max"];

}

// Compute scaled B\_j (T)

double SurfaceMagneticFieldModule::computeB\_j(double t, double B\_s) {

    variables["t"] = t;

    double base\_b = variables["B\_ref"] + 0.4 \* std::sin(variables["omega\_s"] \* t);  // Hypothetical cycle

    return base\_b \* (B\_s / variables["B\_ref"]);

}

// Update variable

void SurfaceMagneticFieldModule::updateVariable(const std::string& name, double value) {

    if (variables.find(name) != variables.end()) {

        variables[name] = value;

    } else {

        std::cerr << "Variable '" << name << "' not found. Adding with value " << value << std::endl;

        variables[name] = value;

    }

}

// UQFFBuoyancyModule.h

// Modular C++ implementation of the full Master Unified Field Equation (F\_U\_Bi\_i & UQFF Integration) for Buoyancy Equations across M74, Eagle Nebula (M16), M84, Centaurus A, Supernova Survey.

// This module can be plugged into a base program (e.g., 'uqff\_buoyancy\_sim.cpp') by including this header and linking the .cpp.

// Usage in base: #include "UQFFBuoyancyModule.h"

// UQFFBuoyancyModule mod; mod.computeFBi(system); mod.updateVariable("F\_rel", {new\_real, new\_imag});

// All variables are stored in a std::map for dynamic addition/subtraction/update, using complex<double> for real/imaginary components.

// Nothing is negligible: Includes all terms - base force, momentum, gravity, vacuum stability, LENR resonance, activation, directed energy, magnetic resonance, neutron, relativistic, neutrino, Sweet vac, Kozima drop.

// Associated text: Outputs descriptive equation string via getEquationText().

// Approximations: Integral approximated as integrand \* x2 (quadratic root); imag parts small and not fully scaled; LENR dominant due to low ω\_0; x2 from quadratic solver approx; F\_rel from 1998 LEP.

// Multi-system params: M74 M=7.17e41 kg r=9.46e20 m; Eagle M16 M=1e36 kg r=2.36e17 m; M84 M=1.46e45 kg r=3.09e22 m; Centaurus A M=4e41 kg r=3.09e21 m; Supernova Survey (generic M=1e30 kg r=1e10 m).

// Watermark: Copyright - Daniel T. Murphy, analyzed Oct 22, 2025.

#ifndef UQFF\_BUOYANCY\_MODULE\_H

#define UQFF\_BUOYANCY\_MODULE\_H

#include <complex>

#include <map>

#include <string>

class UQFFBuoyancyModule {

private:

    using cdouble = std::complex<double>;

    std::map<std::string, cdouble> variables;

public:

    // Constructor: Initialize all variables with multi-system defaults

    UQFFBuoyancyModule();

    // Dynamic variable operations (complex)

    void updateVariable(const std::string& name, cdouble value);

    void addToVariable(const std::string& name, cdouble delta);

    void subtractFromVariable(const std::string& name, cdouble delta);

    // Core computation: Full F\_U\_Bi\_i(r, t) for system (approx integral)

    cdouble computeFBi(const std::string& system, double t);

    // Sub-equations

    cdouble computeCompressed(const std::string& system, double t);  // Integrand

    cdouble computeResonant(const std::string& system);

    cdouble computeBuoyancy(const std::string& system);

    cdouble computeSuperconductive(const std::string& system, double t);

    double computeCompressedG(const std::string& system, double t);  // g(r,t)

    // Output descriptive text of the equation

    std::string getEquationText(const std::string& system, double t);

    // Print all current variables (for debugging/updates)

    void printVariables();

};

#endif // UQFF\_BUOYANCY\_MODULE\_H

// SurfaceMagneticFieldModule.cpp

#include "SurfaceMagneticFieldModule.h"

// Compute B\_s minimum (quiet Sun)

double SurfaceMagneticFieldModule::computeB\_s\_min() {

    return variables["B\_s\_min"];

}

// Compute B\_s maximum (sunspot max)

double SurfaceMagneticFieldModule::computeB\_s\_max() {

    return variables["B\_s\_max"];

}

// Compute scaled B\_j (T)

double SurfaceMagneticFieldModule::computeB\_j(double t, double B\_s) {

    variables["t"] = t;

    double base\_b = variables["B\_ref"] + 0.4 \* std::sin(variables["omega\_s"] \* t);  // Hypothetical cycle

    return base\_b \* (B\_s / variables["B\_ref"]);

}

// Update variable

void SurfaceMagneticFieldModule::updateVariable(const std::string& name, double value) {

    if (variables.find(name) != variables.end()) {

        variables[name] = value;

    } else {

        std::cerr << "Variable '" << name << "' not found. Adding with value " << value << std::endl;

        variables[name] = value;

    }

}

// UQFFBuoyancyModule.cpp

#include "UQFFBuoyancyModule.h"

// Constructor: Initialize all variables with multi-system defaults

UQFFBuoyancyModule::UQFFBuoyancyModule() {

    // Each module uses dynamic Map storage

    this->variables = std::map<std::string, cdouble>();

    this->variables["G"] = cdouble(6.6743e-11, 0.0); // Gravitational constant

    // Initialize other variables as needed

}

// Set system-specific parameters

void UQFFBuoyancyModule::setSystemParams(const std::string& system) {

    if (system == "NewSystem") {

        this->variables["M"] = cdouble(1e42, 0.0); // Mass

        this->variables["r"] = cdouble(1e20, 0.0); // Distance

        // Add any number of parameters

    }

}

// Dynamic variable operations (complex)

void UQFFBuoyancyModule::updateVariable(const std::string& name, cdouble value) {

    this->variables[name] = value;

}

void UQFFBuoyancyModule::addToVariable(const std::string& name, cdouble delta) {

    this->variables[name] += delta;

}

void UQFFBuoyancyModule::subtractFromVariable(const std::string& name, cdouble delta) {

    this->variables[name] -= delta;

}

// Core computation: Full F\_U\_Bi\_i(r, t) for system (approx integral)

cdouble UQFFBuoyancyModule::computeFBi(const std::string& system, double t) {

    // Implementation of the full F\_U\_Bi\_i computation

    // using the stored variables and system parameters

    return cdouble(0.0, 0.0); // Placeholder

}

// Sub-equations

cdouble UQFFBuoyancyModule::computeCompressed(const std::string& system, double t) {

    // Dynamic integrand computation

    // Automatically adapts to system parameters

    // Supports unlimited physics terms

    // Runtime coefficient modification

    return cdouble(0.0, 0.0); // Placeholder

}

cdouble UQFFBuoyancyModule::computeResonant(const std::string& system, double t) {

    return cdouble(0.0, 0.0); // Placeholder

}

cdouble UQFFBuoyancyModule::computeBuoyancy(const std::string& system) {

    return cdouble(0.0, 0.0); // Placeholder

}

cdouble UQFFBuoyancyModule::computeSuperconductive(const std::string& system, double t) {

    return cdouble(0.0, 0.0); // Placeholder

}

double UQFFBuoyancyModule::computeCompressedG(const std::string& system, double t) {

    return 0.0; // Placeholder

}

// UQFFBuoyancyModule.cpp

#include "UQFFBuoyancyModule.h"

#include <complex>

// Constructor: Set all variables with multi-system defaults

UQFFBuoyancyModule::UQFFBuoyancyModule() {

    double pi\_val = 3.141592653589793;

    cdouble zero = {0.0, 0.0};

    cdouble i\_small = {0.0, 1e-37};

    // Base constants (universal)

    variables["G"] = {6.6743e-11, 0.0};

    variables["c"] = {3e8, 0.0};

    variables["hbar"] = {1.0546e-34, 0.0};

    variables["q"] = {1.6e-19, 0.0};

    variables["pi"] = {pi\_val, 0.0};

    variables["m\_e"] = {9.11e-31, 0.0};

    variables["mu\_B"] = {9.274e-24, 0.0};

    variables["g\_Lande"] = {2.0, 0.0};

    variables["k\_B"] = {1.38e-23, 0.0};

    variables["mu0"] = {4 \* pi\_val \* 1e-7, 0.0};

    // Shared params

    variables["F\_rel"] = {4.30e33, 0.0};  // Relativistic coherence from LEP 1998

    variables["F0"] = {1.83e71, 0.0};

    variables["V"] = {1e-3, 0.0};  // Default particle velocity

    variables["theta"] = {pi\_val / 4, 0.0};  // 45 deg

    variables["phi"] = {pi\_val / 4, 0.0};

    variables["omega\_act"] = {2 \* pi\_val \* 300, 0.0};

    variables["k\_act"] = {1e-6, 0.0};

    variables["k\_DE"] = {1e-30, 0.0};

    variables["k\_neutron"] = {1e10, 0.0};

    variables["k\_relativistic"] = {1e-20, 0.0};

    variables["k\_neutrino"] = {1e-15, 0.0};

    variables["k\_Sweet"] = {1e-25, 0.0};

    variables["k\_Kozima"] = {1e-18, 0.0};

    variables["omega\_0\_LENR"] = {2 \* pi\_val \* 1e3, 0.0};  // LENR resonance freq

    variables["k\_LENR"] = {1e-5, 0.0};

    variables["T"] = {300.0, 0.0};  // Temperature in K

    variables["B\_s\_min"] = {0.001, 0.0};  // T

    variables["B\_s\_max"] = {0.4, 0.0};

    variables["B\_ref"] = {0.2, 0.0};  // T

    variables["omega\_s"] = {2 \* pi\_val / (11 \* 365 \* 24 \* 3600), 0.0};  // 11-year cycle

    // System-specific params will be set in setSystemParams()

}

// Set system-specific parameters

void UQFFBuoyancyModule::setSystemParams(const std::string& system)

{

    if (system == "M74") {

        this->variables["M"] = cdouble(7.17e41, 0.0);

        this->variables["r"] = cdouble(9.46e20, 0.0);

    } else if (system == "EagleNebula") {

        this->variables["M"] = cdouble(1e36, 0.0);

        this->variables["r"] = cdouble(2.36e17, 0.0);

    } else if (system == "M84") {

        this->variables["M"] = cdouble(1.46e45, 0.0);

        this->variables["r"] = cdouble(3.09e22, 0.0);

    } else if (system == "CentaurusA") {

        this->variables["M"] = cdouble(4e41, 0.0);

        this->variables["r"] = cdouble(3.09e21, 0.0);

    } else if (system == "SupernovaSurvey") {

        this->variables["M"] = cdouble(1e30, 0.0);

        this->variables["r"] = cdouble(1e10, 0.0);

    }

}

// Dynamic variable operations (complex)

void UQFFBuoyancyModule::updateVariable(const std::string& name, cdouble value) {

    this->variables[name] = value;

}

void UQFFBuoyancyModule::addToVariable(const std::string& name, cdouble delta) {

    this->variables[name] += delta;

}

void UQFFBuoyancyModule::subtractFromVariable(const std::string& name, cdouble delta) {

    this->variables[name] -= delta;

}

// Core computation: Full F\_U\_Bi\_i(r, t) for system (approx integral)

cdouble UQFFBuoyancyModule::computeFBi(const std::string& system, double t) {

    setSystemParams(system);

    cdouble integrand = computeIntegrand(t, system);

    cdouble x2 = computeX2(system);

    return integrand \* x2; // Approx integral

}

// Sub-equations

cdouble UQFFBuoyancyModule::computeCompressed(const std::string& system, double t) {

    setSystemParams(system);

    return computeIntegrand(t, system);

}

cdouble UQFFBuoyancyModule::computeResonant(const std::string& system) {

    setSystemParams(system);

    return computeDPM\_resonance(system);

}

cdouble UQFFBuoyancyModule::computeBuoyancy(const std::string& system) {

    setSystemParams(system);

    return computeUb1(system);

}

cdouble UQFFBuoyancyModule::computeSuperconductive(const std::string& system, double t) {

    setSystemParams(system);

    return computeUi(t, system);

}

double UQFFBuoyancyModule::computeCompressedG(const std::string& system, double t) {

    setSystemParams(system);

    return computeG(t, system);

}

// Output descriptive text of the equation

std::string UQFFBuoyancyModule::getEquationText(const std::string& system, double t) {

    setSystemParams(system);

    std::ostringstream oss;

    oss << "F\_U\_Bi\_i(r, t) = Integral[Integrand(r, t) dt] approximated as Integrand \* x2\n";

    oss << "Where Integrand includes terms for base force, momentum, gravity, vacuum stability, LENR resonance, activation, directed energy, magnetic resonance, neutron, relativistic, neutrino, Sweet vac, Kozima drop.\n";

    oss << "System: " << system << ", Time: " << t << " s\n";

    return oss.str();

}

// Print all current variables (for debugging/updates)

void UQFFBuoyancyModule::printVariables() {

    for (const auto& pair : variables) {

        std::cout << std::setw(15) << pair.first << " : " << pair.second << std::endl;

    }

}

// Compute integrand for F\_U\_Bi\_i

cdouble UQFFBuoyancyModule::computeIntegrand(double t, const std::string& system) {

    // Placeholder implementation

    return cdouble(1.0e10, 1.0e5); // Example value

}

// Compute DPM resonance term

cdouble UQFFBuoyancyModule::computeDPM\_resonance(const std::string& system) {

    // Placeholder implementation

    return cdouble(1.0e8, 1.0e3); // Example value

}

// Compute x2 from quadratic root approximation

cdouble UQFFBuoyancyModule::computeX2(const std::string& system) {

    // Placeholder implementation

    return cdouble(1.0e15, 0.0); // Example value

}

// Compute LENR term

cdouble UQFFBuoyancyModule::computeLENRTerm(const std::string& system) {

    // Placeholder implementation

    return cdouble(1.0e12, 1.0e2); // Example value

}

// Compute gravitational acceleration g(r,t)

double UQFFBuoyancyModule::computeG(double t, const std::string& system) {

    // Placeholder implementation

    return 9.81; // Example value

}

// Compute Q\_wave term

cdouble UQFFBuoyancyModule::computeQ\_wave(double t, const std::string& system) {

    // Placeholder implementation

    return cdouble(1.0e6, 1.0e1); // Example value

}

// Compute Ub1 buoyancy term

cdouble UQFFBuoyancyModule::computeUb1(const std::string& system) {

    // Placeholder implementation

    return cdouble(1.0e9, 1.0e4); // Example value

}

// Compute Ui superconductive term

cdouble UQFFBuoyancyModule::computeUi(double t, const std::string& system) {

    // Placeholder implementation

    return cdouble(1.0e7, 1.0e2); // Example value

    }