🔥 SIMULAÇÃO NPE-PSQ PARA REATOR DE FUSÃO NUCLEAR

Vou criar um sistema NPE-PSQ (Nuclear Plasma Equilibrium - Plasma Stability and Quenching) para controle de plasma instável em reator de fusão tokamak. Este é um sistema muito mais complexo que o MPI biomédico.

📁 ESTRUTURA DE ARQUIVOS:

```
npe_psq_simulation/
├── plasma_physics.c       # Modelos físicos MHD do plasma
├── plasma_controller.c    # Controlador MPC para estabilidade
├── plasma_safety.c        # Sistema de segurança (disrupções)
├── coil_control.c         # Controle de bobinas magnéticas
├── heating_control.c      # Controle de sistemas de aquecimento
├── diagnostics.c          # Sistema de diagnóstico (interferometria, etc.)
├── npe_simulation.c       # Simulação principal
├── plasma_visualization.py # Visualização 3D do plasma
└── npe_config.h           # Configurações do tokamak
```

⚛️ 1. ARQUITETURA DO SISTEMA NPE-PSQ:

```c
// npe_config.h - Configurações para reator de fusão tokamak
#ifndef NPE_CONFIG_H
#define NPE_CONFIG_H

#include <stdint.h>
#include <stdbool.h>
#include <math.h>

// ================= PARÂMETROS DO TOKAMAK =================
#define TOKAMAK_MAJOR_RADIUS     1.8f      // Raio maior (m) - tipo ITER
#define TOKAMAK_MINOR_RADIUS     0.6f      // Raio menor (m)
#define TOKAMAK_TOROIDAL_FIELD   5.3f      // Campo toroidal (T)
#define TOKAMAK_PLASMA_CURRENT   15.0f     // Corrente de plasma (MA)

// ================= PARÂMETROS DO PLASMA =================
#define PLASMA_TEMPERATURE_CORE  15.0f     // Temperatura central (keV)
#define PLASMA_DENSITY_CORE      1.0e20f   // Densidade central (m^-3)
#define PLASMA_BETA_TARGET       0.03f     // Beta total alvo (β = 2μ₀<p>/B²)
#define PLASMA_LI_TARGET         1.0f      // Indutância interna alvo

// ================= PARÂMETROS DE ESTABILIDADE =================
#define SAFETY_FACTOR_Q95_MIN    3.0f      // Fator de segurança mínimo (q95)
#define SAFETY_FACTOR_Q95_MAX    5.0f      // Fator de segurança máximo
#define BETA_NORMAL_LIMIT        3.5f      // Limite βN (normalizado)
```

```
#define LOWER_HYBRID_LIMIT      0.8f     // Limite de modo híbrido inferior

// ================ PARÂMETROS DE CONTROLE ================
#define NUM_PF_COILS            10       // Bobinas de campo poloidal
#define NUM_VERTICAL_COILS      4        // Bobinas de controle vertical
#define NUM_HORIZONTAL_COILS    4        // Bobinas de controle horizontal
#define NUM_HEATING_SYSTEMS     3        // Sistemas de aquecimento

// ================ LIMITES DE SEGURANÇA ================
#define DISRUPTION_CURRENT_RAMP  3.0f    // dIp/dt máximo (MA/s)
#define VERTICAL_DISPLACEMENT_MAX 0.15f  // Deslocamento vertical máximo (m)
#define RADIATION_PEAK_LIMIT    10.0f    // Pico de radiação (MW/m²)
#define WALL_LOAD_LIMIT         1.0f     // Carga térmica na parede (MW/m²)

// ================ TEMPOS CARACTERÍSTICOS ================
#define PLASMA_CURRENT_RISE_TIME 30.0f   // Tempo de subida da corrente (s)
#define ENERGY_CONFINEMENT_TIME  5.0f    // Tempo de confinamento (s)
#define DISRUPTION_WARNING_TIME  0.05f   // Tempo de aviso de disrupção (s)
#define MITIGATION_RESPONSE_TIME 0.01f   // Tempo resposta mitigação (s)

// ================ ESTRUTURAS DE DADOS ================

/**
 * @brief Estado do plasma (variáveis principais)
 */
typedef struct {
    // Corrente e campos
    float plasma_current;        // Ip (MA)
    float safety_factor_q95;     // q95
    float beta_normalized;       // βN
    float li_inductance;         // Indutância interna

    // Posição e forma
    float radial_position;       // Posição radial (m)
    float vertical_position;     // Posição vertical (m)
    float elongation;            // Elongação (κ)
    float triangularity;         // Triangularidade (δ)

    // Parâmetros físicos
    float temperature_core;      // Temperatura central (keV)
    float temperature_edge;      // Temperatura da borda (keV)
    float density_core;          // Densidade central (10^19 m^-3)
    float density_edge;          // Densidade da borda (10^19 m^-3)

    // Estabilidade
    float mhd_activity_level;    // Nível de atividade MHD (0-1)
    float ntm_amplitude;         // Amplitude de ilhas NTM (m)
    float elm_frequency;         // Frequência de ELMs (Hz)
```

```
    // Diagnóstico avançado
    float neutron_rate;           // Taxa de nêutrons (Hz)
    float impurity_concentration;  // Concentração de impurezas (%)
    float radiation_power;        // Potência radiada (MW)

} PlasmaState;

/**
 * @brief Sistema de controle de plasma
 */
typedef struct {
    // Estado atual
    PlasmaState current_state;
    PlasmaState target_state;

    // Controle ativo
    float pf_coil_currents[NUM_PF_COILS];      // Correntes nas bobinas PF
    float vertical_coil_currents[NUM_VERTICAL_COILS];
    float horizontal_coil_currents[NUM_HORIZONTAL_COILS];

    // Sistemas de aquecimento
    struct {
        float power;           // Potência (MW)
        float frequency;       // Frequência (GHz) - para ECRH
        bool enabled;
    } heating_systems[NUM_HEATING_SYSTEMS];

    // Injeção de combustível
    float fuel_injection_rate;  // Taxa de injeção (partículas/s)
    float impurity_injection_rate; // Para mitigação de disrupções

    // Estado do controlador
    enum {
        PSQ_STATE_INIT,
        PSQ_STATE_RAMP_UP,
        PSQ_STATE_FLAT_TOP,
        PSQ_STATE_RAMP_DOWN,
        PSQ_STATE_DISRUPTION,
        PSQ_STATE_MITIGATION,
        PSQ_STATE_SAFE_SHUTDOWN
    } controller_state;

    // Tempo e histórico
    float simulation_time;
    uint32_t iteration_count;
    float state_history[1000][10]; // Histórico para análise
```

```
    // Sistema de segurança
    bool disruption_detected;
    bool mitigation_activated;
    float disruption_warning_time;

    // Métricas de desempenho
    float energy_confinement_time;
    float fusion_gain_Q;
    float stored_energy;

} PlasmaControlSystem;

/**
 * @brief Sistema de diagnóstico
 */
typedef struct {
    // Medições reais (com ruído)
    float interferometer_density[32];    // 32 canais de interferometria
    float thomson_scattering_temp[20];   // 20 pontos Thomson scattering
    float bolometer_channels[48];        // 48 canais de bolometria
    float magnetics_probes[64];          // 64 bobinas magnéticas

    // Diagnóstico avançado
    float soft_xray_array[64];           // Array de raios-X
    float neutron_cameras[8];            // Câmeras de nêutrons
    float spectroscopy_lines[16];        // Linhas espectroscópicas

    // Processamento de sinal
    float mhd_spectrum[1024];            // Espectro MHD (0-100 kHz)
    float coherence_analysis[32][32];    // Análise de coerência

    // Status
    bool system_ok;
    float data_acquisition_rate;         // Taxa de aquisição (Hz)

} DiagnosticsSystem;

/**
 * @brief Sistema de segurança e mitigação
 */
typedef struct {
    // Detecção de disrupção
    struct {
        bool locked_mode_detected;
        bool vertical_displacement_event;
        bool density_limit_exceeded;
        bool beta_limit_exceeded;
        bool current_quench_detected;
```

```
        bool thermal_quench_detected;
    } disruption_flags;

    // Sistema de mitigação
    struct {
        bool massive_gas_injection_ready;
        bool pellet_injection_ready;
        bool killer_pulse_ready;
        bool runaway_electron_mitigation;
    } mitigation_systems;

    // Atuação
    float gas_injection_valve_position;  // 0-100%
    float pellet_injection_rate;         // Pellets/s
    float killer_pulse_amplitude;        // Amplitude do pulso (kA)

    // Histórico
    uint32_t disruption_count;
    uint32_t mitigation_success_count;
    float last_disruption_time;

} SafetyMitigationSystem;

#endif // NPE_CONFIG_H
```

🌪️ 2. FÍSICA DO PLASMA: plasma_physics.c

```c
// plasma_physics.c - Modelos físicos MHD do plasma
#include "npe_config.h"
#include <stdlib.h>
#include <stdio.h>
#include <complex.h>

// Constantes físicas
#define MU0            (4.0e-7 * M_PI)    // Permeabilidade magnética
#define ELECTRON_CHARGE 1.602e-19         // Carga do elétron (C)
#define ELECTRON_MASS   9.109e-31         // Massa do elétron (kg)
#define PROTON_MASS     1.673e-27         // Massa do próton (kg)

/**
 * @brief Equilíbrio MHD simplificado (equação de Grad-Shafranov)
 * ∇²ψ = -μ₀R²dp/dψ - FdF/dψ
 *
 * @param psi Função de fluxo poloidal
 * @param R Coordenada radial maior
 * @param Z Coordenada vertical
```

```
 * @param p Pressão do plasma
 * @param F Função de campo toroidal
 */
float grad_shafranov_solution(float R, float Z, float *params) {
    // Solução simplificada para plasma circular
    float a = TOKAMAK_MINOR_RADIUS;
    float R0 = TOKAMAK_MAJOR_RADIUS;

    // Coordenadas normalizadas
    float r = sqrtf((R - R0)*(R - R0) + Z*Z) / a;

    if (r >= 1.0f) return 0.0f;

    // Perfil parabólico simplificado
    float psi = params[0] * (1.0f - r*r);

    return psi;
}

/**
 * @brief Calcula fator de segurança q(r)
 * q(r) = (rB_φ)/(RB_θ) ≈ (2πB_φr²)/(μ₀R₀I_prψ)
 */
float safety_factor_profile(float r_normalized, PlasmaState *state) {
    float R0 = TOKAMAK_MAJOR_RADIUS;
    float B_toroidal = TOKAMAK_TOROIDAL_FIELD;
    float I_p = state->plasma_current * 1e6; // Convert MA to A

    // Aproximação para plasma circular
    float q = (2.0f * M_PI * B_toroidal * r_normalized * r_normalized *
               TOKAMAK_MINOR_RADIUS * TOKAMAK_MINOR_RADIUS) /
              (MU0 * R0 * I_p);

    // Ajuste para perfil não-uniforme
    q *= (1.0f + 0.5f * r_normalized * r_normalized);

    return q;
}

/**
 * @brief Calcula beta do plasma
 * β = 2μ₀<p>/B²
 */
float calculate_beta(PlasmaState *state) {
    // Pressão média aproximada (p ~ nT)
    float pressure_avg = (state->density_core * 1e19 * state->temperature_core *
                          1.602e-16) / 3.0f; // keV to J, divide by 3 for avg
```

```
    float B_total = sqrtf(TOKAMAK_TOROIDAL_FIELD * TOKAMAK_TOROIDAL_FIELD +
                          powf(MU0 * state->plasma_current * 1e6 /
                               (2.0f * M_PI * TOKAMAK_MINOR_RADIUS), 2));

    float beta = 2.0f * MU0 * pressure_avg / (B_total * B_total);

    return beta;
}

/**
 * @brief Calcula βN (beta normalizado)
 * βN = β(%) * a(m) * B(T) / I_p(MA)
 */
float calculate_beta_normalized(PlasmaState *state) {
    float beta = calculate_beta(state) * 100.0f; // Convert to %

    float beta_N = beta * TOKAMAK_MINOR_RADIUS * TOKAMAK_TOROIDAL_FIELD /
                   state->plasma_current;

    return beta_N;
}

/**
 * @brief Modelo de crescimento de ilhas NTM (Neoclassical Tearing Modes)
 * dw/dt = Δ'w + αw/(1 + w³) - βw
 */
float ntm_island_growth(float w, float w_sat, float delta_prime,
                        float alpha, float beta, float dt) {
    // Taxa de crescimento (modelo simplificado)
    float growth_rate = delta_prime * w +
                        alpha * w / (1.0f + w*w*w) -
                        beta * w;

    // Saturação não-linear
    float saturation = 1.0f / (1.0f + expf(10.0f * (w - w_sat)));

    return w + dt * growth_rate * saturation;
}

/**
 * @brief Modelo de ELMs (Edge Localized Modes)
 * Baseado no modelo de pedestal peeling-ballooning
 */
float elm_cycle_model(float time, float pedestal_pressure,
                      float pedestal_current, float *params) {
    // Frequência de ELMs (modelo simplificado)
    float frequency = 0.1f * sqrtf(pedestal_pressure / pedestal_current);
```

```
    // Amplitude dos ELMs (fracionária da energia)
    float amplitude = 0.05f + 0.1f * sinf(2.0f * M_PI * frequency * time);

    return amplitude;
}

/**
 * @brief Modelo de disrupção térmica
 * Baseado no modelo de corrente de halo e radiação
 */
float thermal_quench_model(float time_since_onset, float initial_energy,
                           float impurity_concentration) {
    // Tempo característico da disrupção térmica (rápido!)
    float tau_TQ = 0.001f; // 1 ms

    // Perda de energia exponencial
    float energy_loss = initial_energy * expf(-time_since_onset / tau_TQ);

    // Efeito das impurezas (aumenta a perda)
    energy_loss *= (1.0f - 0.5f * impurity_concentration);

    return energy_loss;
}

/**
 * @brief Modelo de disrupção de corrente
 * Mais lenta que a térmica
 */
float current_quench_model(float time_since_TQ, float initial_current,
                           float plasma_resistance) {
    // Tempo característico mais longo
    float tau_CQ = 0.01f; // 10 ms

    // Decaimento da corrente
    float current = initial_current * expf(-time_since_TQ / tau_CQ);

    // Efeito da resistividade
    current *= (1.0f - 0.1f * plasma_resistance * time_since_TQ);

    return current;
}

/**
 * @brief Calcula forças eletromagnéticas durante disrupção
 * F = J × B + pressão magnética
 */
float calculate_disruption_forces(PlasmaState *state, float *coil_currents) {
    float force_total = 0.0f;
```

```
    // Força devido à queda de corrente
    float dIp_dt = -state->plasma_current / 0.01f; // Aproximação durante CQ

    // Campo magnético das bobinas
    float B_coil = 0.0f;
    for (int i = 0; i < NUM_PF_COILS; i++) {
        B_coil += coil_currents[i] * 1e-6f /  // kA to MA
                  (2.0f * M_PI * TOKAMAK_MAJOR_RADIUS);
    }

    // Força de Lorentz
    float lorentz_force = dIp_dt * B_coil * TOKAMAK_MINOR_RADIUS;

    // Pressão magnética (B²/2μ₀)
    float B_total = sqrtf(TOKAMAK_TOROIDAL_FIELD * TOKAMAK_TOROIDAL_FIELD +
                          (MU0 * state->plasma_current * 1e6) /
                          (2.0f * M_PI * TOKAMAK_MINOR_RADIUS));

    float magnetic_pressure = B_total * B_total / (2.0f * MU0);

    force_total = lorentz_force + magnetic_pressure * TOKAMAK_MINOR_RADIUS;

    return force_total;
}

/**
 * @brief Modelo de aquecimento por ECRH (Electron Cyclotron Resonance Heating)
 */
float ecrh_heating_model(float power, float frequency,
                         PlasmaState *state, float *deposition_profile) {
    // Frequência do ciclotron do elétron
    float f_ce = ELECTRON_CHARGE * TOKAMAK_TOROIDAL_FIELD /
                 (2.0f * M_PI * ELECTRON_MASS);

    // Eficiência de absorção (simplificada)
    float absorption = 0.0f;
    if (fabsf(frequency - f_ce) < 1e9) { // Dentro da ressonância
        absorption = 0.8f;
    } else {
        absorption = 0.3f * expf(-powf(frequency - f_ce, 2) / (2.0f * 1e18f));
    }

    // Potência depositada
    float power_deposited = power * absorption;

    // Perfil de deposição (gaussiano centrado)
    for (int i = 0; i < 10; i++) {
```

```
        float r = i / 10.0f;
        deposition_profile[i] = expf(-powf(r - 0.5f, 2) / 0.1f);
    }

    // Aquecimento do elétron
    float delta_T = power_deposited / (state->density_core * 1e19 *
                                ELECTRON_CHARGE * 1000.0f); // keV

    return delta_T;
}

/**
 * @brief Modelo de transporte de energia (confinamento)
 * Baseado no scaling ITER-98y2
 */
float energy_confinement_time(PlasmaState *state, float heating_power) {
    // Scaling ITER-98(y,2) simplificado
    float Ip = state->plasma_current; // MA
    float Bt = TOKAMAK_TOROIDAL_FIELD; // T
    float n = state->density_core * 0.1f; // 10^20 m^-3
    float R = TOKAMAK_MAJOR_RADIUS; // m
    float a = TOKAMAK_MINOR_RADIUS; // m
    float kappa = state->elongation;

    float tau_E = 0.0562f * powf(Ip, 0.93f) * powf(Bt, 0.15f) *
                  powf(n, 0.41f) * powf(R, 1.97f) *
                  powf(a, 0.58f) * powf(kappa, 0.78f);

    // Efeito do aquecimento (degradação com potência)
    tau_E *= powf(heating_power, -0.69f);

    return tau_E;
}

/**
 * @brief Simula o próximo estado do plasma
 * Integra as equações MHD simplificadas
 */
void advance_plasma_state(PlasmaState *state, PlasmaControlSystem *control,
                          float dt) {
    // ========== EVOLUÇÃO DA CORRENTE ==========
    // Equação da corrente do plasma (circuito equivalente)
    float Lp = 5.0e-7f; // Indutância do plasma (H)
    float Rp = 1.0e-6f; // Resistência do plasma (Ohm)

    float V_loop = control->pf_coil_currents[0] * 0.1f; // Tensão de loop simplificada

    float dIp_dt = (V_loop - Rp * state->plasma_current * 1e6) / Lp;
```

```
state->plasma_current += dIp_dt * dt / 1e6; // Convert back to MA

// ========== EVOLUÇÃO DA TEMPERATURA ==========
// Equação do balanço de energia
float P_heating = 0.0f;
for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
    if (control->heating_systems[i].enabled) {
        P_heating += control->heating_systems[i].power;
    }
}

float P_loss = state->stored_energy / control->energy_confinement_time;

float dW_dt = P_heating - P_loss;
state->stored_energy += dW_dt * dt;

// Atualiza temperatura (W ~ nTV)
float plasma_volume = 2.0f * M_PI * M_PI * TOKAMAK_MAJOR_RADIUS *
                      TOKAMAK_MINOR_RADIUS * TOKAMAK_MINOR_RADIUS *
                      state->elongation;

state->temperature_core = state->stored_energy * 1e6 /
                          (1.5f * state->density_core * 1e19 *
                           plasma_volume * ELECTRON_CHARGE * 1000.0f);

// ========== EVOLUÇÃO DA DENSIDADE ==========
// Balanço de partículas
float S_in = control->fuel_injection_rate;
float tau_p = 10.0f; // Tempo de confinamento de partículas (s)
float S_out = state->density_core * 1e19 * plasma_volume / tau_p;

float dn_dt = (S_in - S_out) / plasma_volume;
state->density_core += dn_dt * dt / 1e19; // Convert back

// ========== EVOLUÇÃO DA POSIÇÃO ==========
// Equação do movimento vertical
float mass_plasma = state->density_core * 1e19 * plasma_volume *
                    (PROTON_MASS + ELECTRON_MASS);

float F_vertical = 0.0f;
for (int i = 0; i < NUM_VERTICAL_COILS; i++) {
    F_vertical += control->vertical_coil_currents[i] *
                  state->plasma_current * 0.1f;
}

float damping = 0.1f;
float dVz_dt = (F_vertical - damping * state->vertical_position) / mass_plasma;
state->vertical_position += state->vertical_position * dt + 0.5f * dVz_dt * dt * dt;
```

```
    // ========== EVOLUÇÃO DA ESTABILIDADE ==========
    // Atualiza q95
    state->safety_factor_q95 = safety_factor_profile(0.95f, state);

    // Atualiza beta
    state->beta_normalized = calculate_beta_normalized(state);

    // Modelo de atividade MHD (ruído)
    state->mhd_activity_level = 0.1f * sinf(control->simulation_time * 100.0f) +
                                0.05f * ((float)rand() / RAND_MAX);

    // ========== DETECÇÃO DE DISRUPÇÃO ==========
    // Condições para instabilidade
    if (state->safety_factor_q95 < SAFETY_FACTOR_Q95_MIN) {
        state->mhd_activity_level += 0.5f; // Aumenta atividade MHD
    }

    if (state->beta_normalized > BETA_NORMAL_LIMIT) {
        state->mhd_activity_level += 0.3f; // Limite beta excedido
    }

    if (fabsf(state->vertical_position) > VERTICAL_DISPLACEMENT_MAX) {
        state->mhd_activity_level += 0.7f; // VDE (Vertical Displacement Event)
    }
}
```

🎮 3. CONTROLADOR MPC: plasma_controller.c

```c
// plasma_controller.c - Controlador MPC para estabilidade do plasma
#include "npe_config.h"
#include <string.h>
#include <stdio.h>

/**
 * @brief Inicializa o sistema de controle de plasma
 */
void plasma_control_init(PlasmaControlSystem *ctrl) {
    memset(ctrl, 0, sizeof(PlasmaControlSystem));

    // Estado inicial (plasma frio)
    ctrl->current_state.plasma_current = 0.0f;
    ctrl->current_state.temperature_core = 0.1f; // 100 eV
    ctrl->current_state.density_core = 0.01f; // 10^17 m^-3
    ctrl->current_state.safety_factor_q95 = 99.0f; // Muito alto inicialmente
    ctrl->current_state.elongation = 1.0f; // Circular inicialmente
```

```
    // Estado alvo (plasma de fusão)
    ctrl->target_state.plasma_current = TOKAMAK_PLASMA_CURRENT;
    ctrl->target_state.temperature_core = PLASMA_TEMPERATURE_CORE;
    ctrl->target_state.density_core = PLASMA_DENSITY_CORE / 1e19f;
    ctrl->target_state.safety_factor_q95 = 3.5f;
    ctrl->target_state.elongation = 1.8f;
    ctrl->target_state.beta_normalized = PLASMA_BETA_TARGET;

    // Inicializa bobinas
    for (int i = 0; i < NUM_PF_COILS; i++) {
        ctrl->pf_coil_currents[i] = 0.0f;
    }

    // Inicializa sistemas de aquecimento
    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        ctrl->heating_systems[i].power = 0.0f;
        ctrl->heating_systems[i].enabled = false;
        ctrl->heating_systems[i].frequency = (i == 0) ? 170.0f : // ECRH
                                             (i == 1) ? 50.0f :  // ICRH
                                             3.0f;               // NBI (keV)
    }

    ctrl->controller_state = PSQ_STATE_INIT;
    ctrl->simulation_time = 0.0f;
    ctrl->iteration_count = 0;

    printf("[NPE-PSQ] Sistema de controle de plasma inicializado\n");
    printf("          Target: Ip=%.1f MA, Te=%.1f keV, n=%.1f 10^19 m^-3\n",
           ctrl->target_state.plasma_current,
           ctrl->target_state.temperature_core,
           ctrl->target_state.density_core);
}

/**
 * @brief Controlador MPC para estabilidade do plasma
 * Minimiza: J = ∑ (x - x_ref)ᵀQ(x - x_ref) + uᵀRu + ΔuᵀSΔu
 */
void mpc_plasma_control(PlasmaControlSystem *ctrl, PlasmaState *current,
                        PlasmaState *target, float dt) {

    // Horizonte de predição
    const int N = 10;

    // Matrizes de custo (pesos)
    float Q[6][6] = {
        {10.0f, 0, 0, 0, 0, 0},    // Corrente
        {0, 5.0f, 0, 0, 0, 0},     // q95
```

```
    {0, 0, 8.0f, 0, 0, 0},     // βN
    {0, 0, 0, 20.0f, 0, 0},    // Posição vertical
    {0, 0, 0, 0, 3.0f, 0},     // Temperatura
    {0, 0, 0, 0, 0, 3.0f}      // Densidade
};

float R[NUM_PF_COILS + NUM_HEATING_SYSTEMS];
for (int i = 0; i < NUM_PF_COILS + NUM_HEATING_SYSTEMS; i++) {
    R[i] = 0.1f; // Peso no esforço de controle
}

// Vetor de estados
float x[6] = {
    current->plasma_current,
    current->safety_factor_q95,
    current->beta_normalized,
    current->vertical_position,
    current->temperature_core,
    current->density_core
};

float x_ref[6] = {
    target->plasma_current,
    target->safety_factor_q95,
    target->beta_normalized,
    0.0f, // Posição vertical alvo = 0
    target->temperature_core,
    target->density_core
};

// Otimização simplificada (gradiente descendente)
float u_opt[NUM_PF_COILS];
float heating_opt[NUM_HEATING_SYSTEMS];

// Para cada bobina PF
for (int i = 0; i < NUM_PF_COILS; i++) {
    // Sensibilidade aproximada
    float sensitivity[6] = {0.1f,  // dIp/dI_coil
                            -0.05f, // dq95/dI_coil
                            0.02f,  // dβN/dI_coil
                            (i < 2) ? 0.5f : -0.5f, // dZ/dI_coil (depende da bobina)
                            0.01f,  // dTe/dI_coil
                            0.0f};  // dn/dI_coil

    // Calcula gradiente do custo
    float gradient = 0.0f;
    for (int j = 0; j < 6; j++) {
        gradient += 2.0f * Q[j][j] * (x[j] - x_ref[j]) * sensitivity[j];
```

```
    }

    // Lei de controle (com limite de taxa)
    float u_new = ctrl->pf_coil_currents[i] - 0.1f * gradient;

    // Limites físicos
    float max_current = (i < 4) ? 50.0f : 20.0f; // kA
    if (u_new > max_current) u_new = max_current;
    if (u_new < -max_current) u_new = -max_current;

    // Limite de taxa (dI/dt < 10 kA/s)
    float max_delta = 10.0f * dt;
    float delta = u_new - ctrl->pf_coil_currents[i];
    if (fabsf(delta) > max_delta) {
        u_new = ctrl->pf_coil_currents[i] + copysignf(max_delta, delta);
    }

    u_opt[i] = u_new;
}

// Controle de aquecimento
for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
    // Erro de temperatura
    float temp_error = target->temperature_core - current->temperature_core;

    // Controlador PI simplificado
    static float integral_error = 0.0f;
    integral_error += temp_error * dt;

    // Limita integral
    if (integral_error > 100.0f) integral_error = 100.0f;
    if (integral_error < -100.0f) integral_error = -100.0f;

    float Kp = 1.0f;
    float Ki = 0.1f;

    float power = Kp * temp_error + Ki * integral_error;

    // Limites de potência
    float max_power = (i == 0) ? 20.0f :  // ECRH
                      (i == 1) ? 30.0f :  // ICRH
                      40.0f;              // NBI

    if (power > max_power) power = max_power;
    if (power < 0.0f) power = 0.0f;

    heating_opt[i] = power;
}
```

```
    // Aplica controles otimizados
    memcpy(ctrl->pf_coil_currents, u_opt, sizeof(u_opt));

    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        ctrl->heating_systems[i].power = heating_opt[i];
        ctrl->heating_systems[i].enabled = (heating_opt[i] > 0.1f);
    }
}

/**
 * @brief Máquina de estados do controle de plasma
 */
void plasma_state_machine(PlasmaControlSystem *ctrl, float dt) {
    static float ramp_up_rate = 0.0f;
    static float flat_top_duration = 0.0f;

    switch (ctrl->controller_state) {
        case PSQ_STATE_INIT:
            // Preparação inicial
            printf("[NPE-PSQ] Estado: INICIALIZAÇÃO\n");

            // Verifica sistemas
            if (ctrl->iteration_count > 10) {
                ctrl->controller_state = PSQ_STATE_RAMP_UP;
                printf("[NPE-PSQ] Transição para RAMP UP\n");
            }
            break;

        case PSQ_STATE_RAMP_UP:
            // Subida da corrente do plasma
            printf("[NPE-PSQ] Estado: RAMP UP (Ip=%.1f MA)\n",
                   ctrl->current_state.plasma_current);

            // Perfil de rampa (linear por enquanto)
            float target_ip = ctrl->target_state.plasma_current;
            float current_ip = ctrl->current_state.plasma_current;

            ramp_up_rate = 0.5f; // MA/s
            float new_ip = current_ip + ramp_up_rate * dt;

            if (new_ip > target_ip) {
                new_ip = target_ip;
                ctrl->controller_state = PSQ_STATE_FLAT_TOP;
                printf("[NPE-PSQ] Transição para FLAT TOP\n");
            }

            ctrl->current_state.plasma_current = new_ip;
```

```
    // Aumenta aquecimento durante rampa
    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        ctrl->heating_systems[i].power = 5.0f * new_ip / target_ip;
        ctrl->heating_systems[i].enabled = true;
    }
    break;

case PSQ_STATE_FLAT_TOP:
    // Plasma em regime estacionário
    flat_top_duration += dt;

    if (flat_top_duration > 100.0f) { // 100 segundos de flat-top
        ctrl->controller_state = PSQ_STATE_RAMP_DOWN;
        printf("[NPE-PSQ] Transição para RAMP DOWN\n");
    }

    // Mantém parâmetros no alvo usando MPC
    mpc_plasma_control(ctrl, &ctrl->current_state,
                       &ctrl->target_state, dt);

    // Verifica estabilidade
    if (ctrl->current_state.mhd_activity_level > 0.8f) {
        printf("[NPE-PSQ] ALERTA: Alta atividade MHD detectada!\n");
        // Poderíamos transicionar para mitigação aqui
    }
    break;

case PSQ_STATE_RAMP_DOWN:
    // Descida controlada da corrente
    printf("[NPE-PSQ] Estado: RAMP DOWN\n");

    ctrl->current_state.plasma_current -= 0.3f * dt; // MA/s

    // Reduz aquecimento
    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        ctrl->heating_systems[i].power *= 0.99f;
        if (ctrl->heating_systems[i].power < 0.1f) {
            ctrl->heating_systems[i].enabled = false;
        }
    }

    if (ctrl->current_state.plasma_current < 0.1f) {
        ctrl->controller_state = PSQ_STATE_SAFE_SHUTDOWN;
        printf("[NPE-PSQ] Transição para SHUTDOWN SEGURO\n");
    }
    break;
```

```
case PSQ_STATE_DISRUPTION:
    // Disrupção detectada - ativa mitigação
    printf("[NPE-PSQ] ESTADO CRÍTICO: DISRUPÇÃO DETECTADA!\n");
    printf("          Ativando sistemas de mitigação...\n");

    // Ações de emergência
    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        ctrl->heating_systems[i].enabled = false;
    }

    // Injeta gás nobre (mitigação)
    ctrl->fuel_injection_rate = 1e23f; // Alta taxa para mitigação

    // Aplica "killer pulse" nas bobinas
    for (int i = 0; i < NUM_PF_COILS; i++) {
        ctrl->pf_coil_currents[i] *= 0.5f; // Reduz corrente rapidamente
    }

    ctrl->controller_state = PSQ_STATE_MITIGATION;
    break;

case PSQ_STATE_MITIGATION:
    // Em mitigação de disrupção
    printf("[NPE-PSQ] Estado: MITIGAÇÃO DE DISRUPÇÃO\n");

    // Monitora se a mitigação foi bem-sucedida
    if (ctrl->current_state.plasma_current < 0.5f) {
        ctrl->controller_state = PSQ_STATE_SAFE_SHUTDOWN;
        printf("[NPE-PSQ] Mitigação bem-sucedida, desligando...\n");
    }
    break;

case PSQ_STATE_SAFE_SHUTDOWN:
    // Desligamento seguro
    printf("[NPE-PSQ] Estado: SHUTDOWN SEGURO\n");

    // Garante que tudo está desligado
    for (int i = 0; i < NUM_PF_COILS; i++) {
        ctrl->pf_coil_currents[i] = 0.0f;
    }

    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        ctrl->heating_systems[i].enabled = false;
        ctrl->heating_systems[i].power = 0.0f;
    }

    ctrl->fuel_injection_rate = 0.0f;
    break;
```

```
    }
}

/**
 * @brief Detecta condições de disrupção
 * Retorna true se disrupção iminente for detectada
 */
bool detect_imminent_disruption(PlasmaState *state, DiagnosticsSystem *diag) {
    bool disruption_detected = false;

    // 1. Modo locked (rotação do plasma próxima de zero)
    if (state->mhd_activity_level > 0.7f && state->beta_normalized > 2.5f) {
        printf("[DISRUPÇÃO] Modo locked detectado! Nível MHD: %.2f\n",
               state->mhd_activity_level);
        disruption_detected = true;
    }

    // 2. VDE (Vertical Displacement Event)
    if (fabsf(state->vertical_position) > VERTICAL_DISPLACEMENT_MAX * 0.8f) {
        printf("[DISRUPÇÃO] VDE iminente! Deslocamento: %.3f m\n",
               state->vertical_position);
        disruption_detected = true;
    }

    // 3. Excesso de densidade (limite de Greenwald)
    float greenwald_limit = state->plasma_current /
                            (M_PI * TOKAMAK_MINOR_RADIUS * TOKAMAK_MINOR_RADIUS);
    if (state->density_core > greenwald_limit * 0.9f) {
        printf("[DISRUPÇÃO] Limite de Greenwald excedido! n/n_GW=%.2f\n",
               state->density_core / greenwald_limit);
        disruption_detected = true;
    }

    // 4. Excesso de beta
    if (state->beta_normalized > BETA_NORMAL_LIMIT * 0.9f) {
        printf("[DISRUPÇÃO] Limite beta excedido! βN=%.2f\n",
               state->beta_normalized);
        disruption_detected = true;
    }

    // 5. q95 muito baixo
    if (state->safety_factor_q95 < SAFETY_FACTOR_Q95_MIN * 1.1f) {
        printf("[DISRUPÇÃO] q95 muito baixo! q95=%.2f\n",
               state->safety_factor_q95);
        disruption_detected = true;
    }

    // 6. Sinais de diagnóstico (se disponível)
```

```
    if (diag != NULL) {
        // Exemplo: rápido aumento de radiação
        static float last_radiation = 0.0f;
        float radiation_rate = (state->radiation_power - last_radiation) / 0.1f; // W/s
        last_radiation = state->radiation_power;

        if (radiation_rate > 1e9f) { // Aumento rápido de radiação
            printf("[DISRUPÇÃO] Aumento rápido de radiação: %.1e W/s\n",
                   radiation_rate);
            disruption_detected = true;
        }
    }

    return disruption_detected;
}

/**
 * @brief Executa um passo completo do controle de plasma
 */
void plasma_control_step(PlasmaControlSystem *ctrl, float dt) {
    // Atualiza tempo
    ctrl->simulation_time += dt;
    ctrl->iteration_count++;

    // Executa máquina de estados
    plasma_state_machine(ctrl, dt);

    // Avança o estado do plasma (física)
    advance_plasma_state(&ctrl->current_state, ctrl, dt);

    // Calcula métricas de desempenho
    ctrl->energy_confinement_time = energy_confinement_time(
        &ctrl->current_state,
        ctrl->heating_systems[0].power +
        ctrl->heating_systems[1].power +
        ctrl->heating_systems[2].power
    );

    // Calcula ganho de fusão Q (simplificado)
    // P_fusão ~ n²T²σv
    float n = ctrl->current_state.density_core * 1e19; // m^-3
    float T = ctrl->current_state.temperature_core * 1000.0f; // eV
    float sigma_v = 1e-22f * powf(T/10000.0f, 2.0f); // m³/s (aproximação)

    float P_fusion = n * n * sigma_v * 3.5e6f * ELECTRON_CHARGE *
                     2.0f * M_PI * M_PI * TOKAMAK_MAJOR_RADIUS *
                     TOKAMAK_MINOR_RADIUS * TOKAMAK_MINOR_RADIUS *
                     ctrl->current_state.elongation;
```

```
    float P_heating = 0.0f;
    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        if (ctrl->heating_systems[i].enabled) {
            P_heating += ctrl->heating_systems[i].power * 1e6f; // MW to W
        }
    }

    if (P_heating > 0.0f) {
        ctrl->fusion_gain_Q = P_fusion / P_heating;
    } else {
        ctrl->fusion_gain_Q = 0.0f;
    }

    // Armazena histórico (para análise)
    int idx = ctrl->iteration_count % 1000;
    ctrl->state_history[idx][0] = ctrl->simulation_time;
    ctrl->state_history[idx][1] = ctrl->current_state.plasma_current;
    ctrl->state_history[idx][2] = ctrl->current_state.temperature_core;
    ctrl->state_history[idx][3] = ctrl->current_state.safety_factor_q95;
    ctrl->state_history[idx][4] = ctrl->current_state.beta_normalized;
    ctrl->state_history[idx][5] = ctrl->current_state.vertical_position;
    ctrl->state_history[idx][6] = ctrl->current_state.mhd_activity_level;
    ctrl->state_history[idx][7] = ctrl->fusion_gain_Q;
    ctrl->state_history[idx][8] = P_heating / 1e6f; // MW
    ctrl->state_history[idx][9] = ctrl->energy_confinement_time;
}
```

⚡ 4. SISTEMA DE SEGURANÇA: plasma_safety.c

```c
// plasma_safety.c - Sistema de segurança para disrupções de plasma
#include "npe_config.h"
#include <stdio.h>
#include <math.h>

/**
 * @brief Sistema de mitigação de disrupção por injeção de gás massivo (MGI)
 */
void massive_gas_injection(MitigationSystem *mit, float gas_quantity,
                           float injection_time) {
    printf("[SEG] Ativando MGI: %.1f Pa.m³ em %.3f s\n",
           gas_quantity, injection_time);

    // Abre válvula de injeção
    mit->gas_injection_valve_position = 100.0f; // 100% aberta
```

```
    // Injetores típicos: Ne, Ar, D₂
    // Quantidade: ~100-1000 Pa.m³

    // Efeitos:
    // 1. Aumenta radiação → resfria plasma
    // 2. Aumenta densidade → aumenta resistividade
    // 3. Dilui plasma → reduz temperatura

    // Tempo de resposta: 1-10 ms
}

/**
 * @brief Sistema de mitigação por injeção de pellets
 */
void pellet_injection(MitigationSystem *mit, float pellet_size,
                      float velocity, int num_pellets) {
    printf("[SEG] Injetando pellets: %d pellets de %.1f mm a %.0f m/s\n",
           num_pellets, pellet_size * 1000.0f, velocity);

    // Pellets de deutério/gás congelado
    // Tamanho: 1-3 mm
    // Velocidade: 100-1000 m/s
    // Penetração: 5-50 cm

    mit->pellet_injection_rate = num_pellets / 0.1f; // pellets/s
}

/**
 * @brief "Killer pulse" - pulso de corrente para matar o plasma
 */
void killer_pulse(MitigationSystem *mit, float coil_current,
                  float duration) {
    printf("[SEG] Aplicando killer pulse: %.0f kA por %.3f s\n",
           coil_current, duration);

    // Aplica corrente reversa nas bobinas
    // Objetivo: destruir simetria do campo magnético
    // Interrompe confinamento → extingue plasma

    mit->killer_pulse_amplitude = coil_current;
}

/**
 * @brief Mitigação de elétrons runaway
 */
void runaway_electron_mitigation(MitigationSystem *mit, float frequency,
                                 float power) {
    printf("[SEG] Mitigação de elétrons runaway: %.1f GHz, %.1f MW\n",
```

```
        frequency, power);

    // Técnicas:
    // 1. Aquecimento por ECRH em ressonância
    // 2. Injeção de poeira/impurezas
    // 3. Modulação do campo magnético

    mit->runaway_electron_mitigation = true;
}

/**
 * @brief Calcula forças eletromagnéticas durante disrupção
 */
float calculate_disruption_forces(PlasmaState *state, float *coil_currents) {
    // Durante uma disrupção, as forças podem chegar a MILHÕES de Newtons!

    // Força devido à mudança de corrente do plasma
    float dIp_dt = -state->plasma_current / 0.01f; // Decaimento em ~10ms

    // Campo das bobinas PF
    float B_coil_total = 0.0f;
    for (int i = 0; i < NUM_PF_COILS; i++) {
        B_coil_total += coil_currents[i] * 1e-3f /  // kA to A
                        (2.0f * M_PI * TOKAMAK_MAJOR_RADIUS);
    }

    // Força de Lorentz: F = I × B
    float lorentz_force = fabsf(dIp_dt * B_coil_total * TOKAMAK_MINOR_RADIUS);

    // Pressão magnética: p_mag = B²/2μ₀
    float B_total = sqrtf(TOKAMAK_TOROIDAL_FIELD * TOKAMAK_TOROIDAL_FIELD +
                          powf(MU0 * state->plasma_current * 1e6 /
                               (2.0f * M_PI * TOKAMAK_MINOR_RADIUS), 2));

    float magnetic_pressure = B_total * B_total / (2.0f * MU0);
    float pressure_force = magnetic_pressure *
                           (2.0f * M_PI * TOKAMAK_MINOR_RADIUS *
                            TOKAMAK_MINOR_RADIUS * state->elongation);

    float total_force = lorentz_force + pressure_force;

    printf("[SEG] Forças durante disrupção: Lorentz=%.1f MN, Pressão=%.1f MN\n",
           lorentz_force / 1e6, pressure_force / 1e6);

    return total_force;
}

/**
```

```
 * @brief Sistema de interlock para proteção do tokamak
 */
bool safety_interlock_check(PlasmaState *state, PlasmaControlSystem *ctrl) {
    bool safe_to_continue = true;

    // 1. Temperatura das bobinas
    static float coil_temperatures[NUM_PF_COILS];
    for (int i = 0; i < NUM_PF_COILS; i++) {
        // Simulação simples de aquecimento
        coil_temperatures[i] += ctrl->pf_coil_currents[i] * ctrl->pf_coil_currents[i] * 0.001f;

        if (coil_temperatures[i] > 100.0f) { // 100°C limite
            printf("[INTERLOCK] Bobina %d superaquecida: %.1f°C\n",
                   i, coil_temperatures[i]);
            safe_to_continue = false;
        }
    }

    // 2. Vacuum integrity
    static float vacuum_pressure = 1e-5f; // Pa
    if (vacuum_pressure > 1e-3f) { // Perda de vácuo
        printf("[INTERLOCK] Perda de vácuo: %.1e Pa\n", vacuum_pressure);
        safe_to_continue = false;
    }

    // 3. Cooling water flow
    static float cooling_flow = 100.0f; // L/s
    if (cooling_flow < 50.0f) {
        printf("[INTERLOCK] Fluxo de resfriamento baixo: %.1f L/s\n", cooling_flow);
        safe_to_continue = false;
    }

    // 4. Magnetic field sensors
    float measured_Bt = TOKAMAK_TOROIDAL_FIELD * (0.95f + 0.1f *
((float)rand()/RAND_MAX));
    if (fabsf(measured_Bt - TOKAMAK_TOROIDAL_FIELD) > 0.5f) {
        printf("[INTERLOCK] Anomalia no campo toroidal: %.2f T\n", measured_Bt);
        safe_to_continue = false;
    }

    // 5. Plasma position limits
    if (fabsf(state->vertical_position) > VERTICAL_DISPLACEMENT_MAX) {
        printf("[INTERLOCK] Deslocamento vertical excessivo: %.3f m\n",
               state->vertical_position);
        safe_to_continue = false;
    }

    if (!safe_to_continue) {
```

```
        printf("[INTERLOCK] DESLIGAMENTO DE SEGURANÇA ATIVADO\n");
    }

    return safe_to_continue;
}

/**
 * @brief Sistema de diagnóstico em tempo real para prevenção de disrupções
 */
DisruptionPrediction realtime_disruption_prediction(PlasmaState *state,
                                                    DiagnosticsSystem *diag,
                                                    float time_window) {
    DisruptionPrediction prediction;
    prediction.disruption_probability = 0.0f;
    prediction.time_to_disruption = 999.0f;
    strcpy(prediction.most_likely_cause, "Nenhuma");

    // Análise baseada em múltiplos indicadores

    // 1. Análise de modo locked
    float locked_mode_amplitude = state->mhd_activity_level;
    if (locked_mode_amplitude > 0.5f) {
        prediction.disruption_probability += 0.4f;
        prediction.time_to_disruption = 0.05f; // 50 ms
        strcpy(prediction.most_likely_cause, "Modo Locked");
    }

    // 2. Análise de crescimento de ilhas NTM
    if (state->ntm_amplitude > 0.05f) { // 5 cm
        prediction.disruption_probability += 0.3f;
        if (prediction.time_to_disruption > 0.5f) {
            prediction.time_to_disruption = 0.5f; // 500 ms
            strcpy(prediction.most_likely_cause, "Ilhas NTM");
        }
    }

    // 3. Análise de limite beta
    float beta_margin = (BETA_NORMAL_LIMIT - state->beta_normalized) /
BETA_NORMAL_LIMIT;
    if (beta_margin < 0.1f) { // Menos de 10% de margem
        prediction.disruption_probability += 0.3f;
        if (prediction.time_to_disruption > 1.0f) {
            prediction.time_to_disruption = 1.0f; // 1 s
            strcpy(prediction.most_likely_cause, "Limite Beta");
        }
    }

    // 4. Análise de limite de densidade
```

```
    float greenwald_limit = state->plasma_current /
                            (M_PI * TOKAMAK_MINOR_RADIUS * TOKAMAK_MINOR_RADIUS);
    float density_margin = (greenwald_limit - state->density_core) / greenwald_limit;
    if (density_margin < 0.1f) {
        prediction.disruption_probability += 0.2f;
        if (prediction.time_to_disruption > 2.0f) {
            prediction.time_to_disruption = 2.0f; // 2 s
            strcpy(prediction.most_likely_cause, "Limite de Densidade");
        }
    }

    // 5. Análise de q95
    float q_margin = (state->safety_factor_q95 - SAFETY_FACTOR_Q95_MIN) /
                     SAFETY_FACTOR_Q95_MIN;
    if (q_margin < 0.1f) {
        prediction.disruption_probability += 0.4f;
        if (prediction.time_to_disruption > 0.1f) {
            prediction.time_to_disruption = 0.1f; // 100 ms
            strcpy(prediction.most_likely_cause, "q95 Baixo");
        }
    }

    // Limita probabilidade a 1.0
    if (prediction.disruption_probability > 1.0f) {
        prediction.disruption_probability = 1.0f;
    }

    return prediction;
}

/**
 * @brief Sistema de tomada de decisão para mitigação
 */
MitigationDecision decide_mitigation_action(DisruptionPrediction prediction,
                                            PlasmaState *state) {
    MitigationDecision decision;
    decision.action = MITIGATION_NONE;
    decision.urgency = 0.0f;

    // Matriz de decisão baseada em probabilidade e tempo
    if (prediction.disruption_probability > 0.8f ||
        prediction.time_to_disruption < 0.01f) {
        // DISRUPÇÃO IMINENTE - AÇÃO IMEDIATA
        decision.action = MITIGATION_MGI_KILLERPULSE;
        decision.urgency = 1.0f;
        printf("[SEG] DECISÃO: Ação imediata - MGI + Killer Pulse\n");
    }
    else if (prediction.disruption_probability > 0.5f ||
```

```
            prediction.time_to_disruption < 0.1f) {
        // ALTO RISCO - AÇÃO RÁPIDA
        decision.action = MITIGATION_MGI;
        decision.urgency = 0.8f;
        printf("[SEG] DECISÃO: Ação rápida - MGI\n");
    }
    else if (prediction.disruption_probability > 0.3f ||
             prediction.time_to_disruption < 1.0f) {
        // RISCO MODERADO - AÇÃO PREVENTIVA
        decision.action = MITIGATION_PELLET;
        decision.urgency = 0.5f;
        printf("[SEG] DECISÃO: Ação preventiva - Pellets\n");
    }
    else if (prediction.disruption_probability > 0.1f) {
        // RISCO BAIXO - AJUSTE DE CONTROLE
        decision.action = MITIGATION_CONTROL_ADJUST;
        decision.urgency = 0.2f;
        printf("[SEG] DECISÃO: Ajuste de controle\n");
    }

    // Considera causa específica
    if (strstr(prediction.most_likely_cause, "q95")) {
        // Para q95 baixo, aumentar corrente ou reduzir campo
        decision.control_adjustment = "Aumentar Ip ou reduzir Bt";
    }
    else if (strstr(prediction.most_likely_cause, "Beta")) {
        // Para beta alto, reduzir aquecimento
        decision.control_adjustment = "Reduzir aquecimento";
    }
    else if (strstr(prediction.most_likely_cause, "Densidade")) {
        // Para densidade alta, reduzir injeção de combustível
        decision.control_adjustment = "Reduzir injeção de combustível";
    }

    return decision;
}
```

## 🖥️ 5. SIMULAÇÃO PRINCIPAL: npe_simulation.c

```c
// npe_simulation.c - Simulação principal do sistema NPE-PSQ
#include "npe_config.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
```

```
// Protótipos
void simulate_normal_operation(const char* filename);
void simulate_disruption_scenario(const char* filename);
void simulate_mitigation_test(const char* filename);
void print_simulation_header();
void print_simulation_results(PlasmaControlSystem *ctrl);

/**
 * @brief Simulação de operação normal (sem disrupções)
 */
void simulate_normal_operation(const char* filename) {
    printf("\n=== SIMULAÇÃO: OPERAÇÃO NORMAL ===\n");

    FILE *csv = fopen(filename, "w");
    if (!csv) {
        printf("Erro ao criar arquivo CSV\n");
        return;
    }

    // Cabeçalho CSV
    fprintf(csv,
"time_s,Ip_MA,Te_keV,q95,beta_N,Z_m,MHD_level,Q_fusion,P_heat_MW,tau_E_s\n");

    // Inicializa sistemas
    PlasmaControlSystem ctrl;
    plasma_control_init(&ctrl);

    DiagnosticsSystem diag;
    memset(&diag, 0, sizeof(DiagnosticsSystem));
    diag.system_ok = true;

    SafetyMitigationSystem safety;
    memset(&safety, 0, sizeof(SafetyMitigationSystem));

    // Parâmetros da simulação
    float dt = 0.01f; // 10 ms (tempo típico de controle)
    float total_time = 150.0f; // 150 segundos

    printf("Simulando %.1f segundos (dt=%.3f s)...\n", total_time, dt);

    for (float time = 0.0f; time < total_time; time += dt) {
        // Executa passo de controle
        plasma_control_step(&ctrl, dt);

        // Verifica segurança
        if (!safety_interlock_check(&ctrl.current_state, &ctrl)) {
            printf("[SIM] Interlock ativado - desligando...\n");
            break;
```

```
        }

        // Detecta disrupções
        DisruptionPrediction pred = realtime_disruption_prediction(
            &ctrl.current_state, &diag, 1.0f);

        if (pred.disruption_probability > 0.7f) {
            printf("[SIM] Disrupção prevista! Prob=%.1f%%, Tempo=%.3f s\n",
                   pred.disruption_probability * 100.0f,
                   pred.time_to_disruption);
        }

        // Log para CSV
        fprintf(csv, "%.3f,%.2f,%.1f,%.2f,%.2f,%.3f,%.2f,%.3f,%.1f,%.1f\n",
                time,
                ctrl.current_state.plasma_current,
                ctrl.current_state.temperature_core,
                ctrl.current_state.safety_factor_q95,
                ctrl.current_state.beta_normalized,
                ctrl.current_state.vertical_position,
                ctrl.current_state.mhd_activity_level,
                ctrl.fusion_gain_Q,
                ctrl.heating_systems[0].power +
                ctrl.heating_systems[1].power +
                ctrl.heating_systems[2].power,
                ctrl.energy_confinement_time);

        // Progresso
        if (fmodf(time, 10.0f) < dt) {
            printf("[SIM] t=%.1f s: Ip=%.1f MA, Te=%.1f keV, Q=%.3f\n",
                   time, ctrl.current_state.plasma_current,
                   ctrl.current_state.temperature_core,
                   ctrl.fusion_gain_Q);
        }
    }

    fclose(csv);

    // Resultados finais
    print_simulation_results(&ctrl);

    printf("\nDados salvos em: %s\n", filename);
}

/**
 * @brief Simulação de cenário de disrupção
 */
void simulate_disruption_scenario(const char* filename) {
```

```
printf("\n=== SIMULAÇÃO: CENÁRIO DE DISRUPÇÃO ===\n");
printf("      (Teste sem mitigação)\n");

FILE *csv = fopen(filename, "w");
if (!csv) {
    printf("Erro ao criar arquivo CSV\n");
    return;
}

fprintf(csv, "time_s,Ip_MA,Te_keV,q95,beta_N,Z_m,MHD_level,Force_MN,Event\n");

PlasmaControlSystem ctrl;
plasma_control_init(&ctrl);

// Acelera simulação para chegar no flat-top mais rápido
ctrl.controller_state = PSQ_STATE_FLAT_TOP;
ctrl.current_state.plasma_current = 15.0f;
ctrl.current_state.temperature_core = 10.0f;
ctrl.current_state.density_core = 0.8f;
ctrl.current_state.safety_factor_q95 = 3.5f;

// Induz disrupção em t = 5s
float disruption_time = 5.0f;
float dt = 0.001f; // 1 ms (precisa ser pequeno para disrupção)
bool disruption_triggered = false;

printf("Induzindo disrupção em t=%.1f s...\n", disruption_time);

for (float time = 0.0f; time < 10.0f; time += dt) {
    // Induz disrupção
    if (!disruption_triggered && time >= disruption_time) {
        printf("[DISRUPÇÃO] DISPARANDO DISRUPÇÃO!\n");

        // Causa: q95 muito baixo + modo locked
        ctrl.current_state.safety_factor_q95 = 2.0f;
        ctrl.current_state.mhd_activity_level = 0.9f;
        ctrl.current_state.vertical_position = 0.2f; // VDE

        disruption_triggered = true;
    }

    // Durante disrupção, física diferente
    if (disruption_triggered) {
        // Modelo de disrupção
        float time_since_disruption = time - disruption_time;

        if (time_since_disruption < 0.001f) {
            // Thermal Quench (1 ms)
```

```
            ctrl.current_state.temperature_core *= 0.1f; // Queda de 90%
            printf("[DISRUPÇÃO] THERMAL QUENCH! Te cai para %.1f keV\n",
                   ctrl.current_state.temperature_core);
        }
        else if (time_since_disruption < 0.01f) {
            // Current Quench (10 ms)
            float decay_rate = 150.0f; // MA/s
            ctrl.current_state.plasma_current -= decay_rate * dt;
            printf("[DISRUPÇÃO] CURRENT QUENCH! Ip=%.1f MA\n",
                   ctrl.current_state.plasma_current);
        }

        // Calcula forças
        float force = calculate_disruption_forces(&ctrl.current_state,
                                                  ctrl.pf_coil_currents);

        fprintf(csv, "%.3f,%.2f,%.1f,%.2f,%.2f,%.3f,%.2f,%.1f,DISRUPTION\n",
                time,
                ctrl.current_state.plasma_current,
                ctrl.current_state.temperature_core,
                ctrl.current_state.safety_factor_q95,
                ctrl.current_state.beta_normalized,
                ctrl.current_state.vertical_position,
                ctrl.current_state.mhd_activity_level,
                force / 1e6);
    }
    else {
        // Operação normal
        plasma_control_step(&ctrl, dt);

        fprintf(csv, "%.3f,%.2f,%.1f,%.2f,%.2f,%.3f,%.2f,0.0,NORMAL\n",
                time,
                ctrl.current_state.plasma_current,
                ctrl.current_state.temperature_core,
                ctrl.current_state.safety_factor_q95,
                ctrl.current_state.beta_normalized,
                ctrl.current_state.vertical_position,
                ctrl.current_state.mhd_activity_level);
    }

    if (ctrl.current_state.plasma_current < 0.1f && disruption_triggered) {
        printf("[DISRUPÇÃO] Plasma extinto em t=%.3f s\n", time);
        break;
    }
}

fclose(csv);
printf("\nDados de disrupção salvos em: %s\n", filename);
```

```
}

/**
 * @brief Teste de sistema de mitigação
 */
void simulate_mitigation_test(const char* filename) {
    printf("\n=== SIMULAÇÃO: TESTE DE MITIGAÇÃO ===\n");

    FILE *csv = fopen(filename, "w");
    if (!csv) {
        printf("Erro ao criar arquivo CSV\n");
        return;
    }

    fprintf(csv, "time_s,Ip_MA,Te_keV,q95,MHD_level,Action,Success\n");

    PlasmaControlSystem ctrl;
    plasma_control_init(&ctrl);
    ctrl.controller_state = PSQ_STATE_FLAT_TOP;
    ctrl.current_state.plasma_current = 15.0f;

    SafetyMitigationSystem safety;
    memset(&safety, 0, sizeof(SafetyMitigationSystem));

    float dt = 0.001f;
    float trigger_time = 3.0f;
    bool mitigation_triggered = false;
    float mitigation_start = 0.0f;

    printf("Testando sistema de mitigação...\n");

    for (float time = 0.0f; time < 6.0f; time += dt) {
        // Induz condição de disrupção
        if (!mitigation_triggered && time >= trigger_time) {
            printf("[MITIGAÇÃO] Condição de disrupção detectada!\n");
            ctrl.current_state.mhd_activity_level = 0.8f;
            ctrl.current_state.safety_factor_q95 = 2.2f;

            // Ativa mitigação
            massive_gas_injection(&safety, 500.0f, 0.01f); // 500 Pa.m³
            killer_pulse(&safety, 50.0f, 0.02f); // 50 kA

            mitigation_triggered = true;
            mitigation_start = time;
        }

        // Durante mitigação
        if (mitigation_triggered) {
```

```
            float mitigation_time = time - mitigation_start;

            // Efeito da mitigação
            if (mitigation_time < 0.005f) {
                // Gás começa a entrar
                ctrl.current_state.density_core *= 1.5f;
            }
            else if (mitigation_time < 0.01f) {
                // Killer pulse atua
                ctrl.current_state.plasma_current *= 0.8f;
                ctrl.current_state.mhd_activity_level *= 0.5f;
            }

            // Sucesso da mitigação
            bool success = (ctrl.current_state.mhd_activity_level < 0.3f &&
                            ctrl.current_state.plasma_current > 5.0f);

            fprintf(csv, "%.3f,%.2f,%.1f,%.2f,%.2f,MITIGATION,%d\n",
                    time,
                    ctrl.current_state.plasma_current,
                    ctrl.current_state.temperature_core,
                    ctrl.current_state.safety_factor_q95,
                    ctrl.current_state.mhd_activity_level,
                    success ? 1 : 0);

            if (success && mitigation_time > 0.02f) {
                printf("[MITIGAÇÃO] SUCESSO! Disrupção evitada.\n");
                break;
            }
        }
        else {
            plasma_control_step(&ctrl, dt);

            fprintf(csv, "%.3f,%.2f,%.1f,%.2f,%.2f,NORMAL,1\n",
                    time,
                    ctrl.current_state.plasma_current,
                    ctrl.current_state.temperature_core,
                    ctrl.current_state.safety_factor_q95,
                    ctrl.current_state.mhd_activity_level);
        }
    }

    fclose(csv);
    printf("\nDados de mitigação salvos em: %s\n", filename);
}

/**
 * @brief Imprime resultados da simulação
```

```
 */
void print_simulation_results(PlasmaControlSystem *ctrl) {
    printf("\n=== RESULTADOS DA SIMULAÇÃO ===\n");
    printf("\n📊 PARÂMETROS FINAIS DO PLASMA:\n");
    printf("   Corrente do plasma (Ip):      %.2f MA\n", ctrl->current_state.plasma_current);
    printf("   Temperatura central (Te):     %.1f keV\n", ctrl->current_state.temperature_core);
    printf("   Densidade central (ne):       %.2f ×10¹⁹ m⁻³\n", ctrl->current_state.density_core);
    printf("   Fator de segurança (q95):     %.2f\n", ctrl->current_state.safety_factor_q95);
    printf("   Beta normalizado (βN):        %.2f\n", ctrl->current_state.beta_normalized);
    printf("   Ganho de fusão (Q):           %.4f\n", ctrl->fusion_gain_Q);
    printf("   Tempo de confinamento (τ_E):  %.2f s\n", ctrl->energy_confinement_time);

    printf("\n⚡ SISTEMAS DE AQUECIMENTO:\n");
    float total_power = 0.0f;
    for (int i = 0; i < NUM_HEATING_SYSTEMS; i++) {
        if (ctrl->heating_systems[i].enabled) {
            printf("   Sistema %d: %.1f MW", i, ctrl->heating_systems[i].power);
            if (i == 0) printf(" (ECRH)");
            else if (i == 1) printf(" (ICRH)");
            else printf(" (NBI)");
            printf("\n");
            total_power += ctrl->heating_systems[i].power;
        }
    }
    printf("   Total: %.1f MW\n", total_power);

    printf("\n🧲 BOBINAS DE CAMPO POLOIDAL:\n");
    float max_current = 0.0f;
    for (int i = 0; i < NUM_PF_COILS; i++) {
        if (fabsf(ctrl->pf_coil_currents[i]) > fabsf(max_current)) {
            max_current = ctrl->pf_coil_currents[i];
        }
    }
    printf("   Corrente máxima: %.1f kA\n", max_current);

    printf("\n🎯 ESTABILIDADE:\n");
    printf("   Nível de atividade MHD:       %.2f\n", ctrl->current_state.mhd_activity_level);
    printf("   Posição vertical:             %.3f m\n", ctrl->current_state.vertical_position);

    if (ctrl->current_state.mhd_activity_level > 0.7f) {
        printf("   ⚠️  ALTA ATIVIDADE MHD DETECTADA!\n");
    }

    printf("\n⏱️  TEMPO DE SIMULAÇÃO:\n");
    printf("   Tempo total:                  %.1f s\n", ctrl->simulation_time);
    printf("   Iterações:                    %u\n", ctrl->iteration_count);
}
```

```
/**
 * @brief Menu principal da simulação
 */
int main() {
    printf("\n");
    printf("================================================\n");
    printf("  NPE-PSQ - SIMULAÇÃO DE REATOR DE FUSÃO\n");
    printf("  Nuclear Plasma Equilibrium - Plasma Stability and Quenching\n");
    printf("================================================\n");

    // Semente para números aleatórios
    srand(time(NULL));

    // Cria diretório para resultados
    #ifdef _WIN32
        system("mkdir results 2>nul");
    #else
        system("mkdir -p results");
    #endif

    int choice = 0;

    printf("\nSelecione o modo de simulação:\n");
    printf("1. Operação normal (sem disrupções)\n");
    printf("2. Cenário de disrupção (sem mitigação)\n");
    printf("3. Teste de sistema de mitigação\n");
    printf("4. Todas as simulações\n");
    printf("Escolha (1-4): ");

    // Para simplicidade, executa todas
    choice = 4;
    // scanf("%d", &choice); // Descomente para interação

    switch (choice) {
        case 1:
            simulate_normal_operation("results/normal_operation.csv");
            break;
        case 2:
            simulate_disruption_scenario("results/disruption_no_mitigation.csv");
            break;
        case 3:
            simulate_mitigation_test("results/mitigation_test.csv");
            break;
        case 4:
        default:
            simulate_normal_operation("results/normal_operation.csv");
            simulate_disruption_scenario("results/disruption_no_mitigation.csv");
            simulate_mitigation_test("results/mitigation_test.csv");
```

```
            break;
        }

        printf("\n================================================\n");
        printf("  SIMULAÇÃO CONCLUÍDA\n");
        printf("================================================\n");
        printf("\nExecute para visualização:\n");
        printf("  python3 plasma_visualization.py\n");
        printf("\nArquivos gerados:\n");
        printf("  results/normal_operation.csv\n");
        printf("  results/disruption_no_mitigation.csv\n");
        printf("  results/mitigation_test.csv\n");

        return 0;
}
```

## 📈 6. VISUALIZAÇÃO 3D: plasma_visualization.py

```python
# plasma_visualization.py - Visualização 3D do plasma tokamak
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
from matplotlib.animation import FuncAnimation
import plotly.graph_objects as go
from plotly.subplots import make_subplots

def plot_plasma_equilibrium_3d():
    """Visualização 3D do equilíbrio do plasma"""

    fig = plt.figure(figsize=(16, 12))

    # 1. Geometria do tokamak
    ax1 = fig.add_subplot(231, projection='3d')

    # Parâmetros do tokamak
    R0 = 1.8  # Raio maior (m)
    a = 0.6   # Raio menor (m)
    kappa = 1.8  # Elongação

    # Superfície do plasma (D-shaped)
    theta = np.linspace(0, 2*np.pi, 100)
    phi = np.linspace(0, 2*np.pi, 100)
    theta, phi = np.meshgrid(theta, phi)

    # Coordenadas toroidais
```

```
R = R0 + a * np.cos(theta + 0.3 * np.sin(theta))  # Triangularidade
Z = kappa * a * np.sin(theta)

# Coordenadas cartesianas
X = R * np.cos(phi)
Y = R * np.sin(phi)

# Plota superfície do plasma
ax1.plot_surface(X, Y, Z, alpha=0.5, color='red', label='Plasma')

# Bobinas toroidais
n_coils = 18
for i in range(n_coils):
    phi_coil = 2*np.pi*i/n_coils
    R_coil = R0 + 1.2*a
    theta_coil = np.linspace(0, 2*np.pi, 50)

    X_coil = R_coil * np.cos(phi_coil) + 0.1*np.cos(theta_coil)*np.cos(phi_coil)
    Y_coil = R_coil * np.sin(phi_coil) + 0.1*np.cos(theta_coil)*np.sin(phi_coil)
    Z_coil = 0.1*np.sin(theta_coil)

    ax1.plot(X_coil, Y_coil, Z_coil, 'b-', alpha=0.3, linewidth=0.5)

ax1.set_xlabel('X (m)')
ax1.set_ylabel('Y (m)')
ax1.set_zlabel('Z (m)')
ax1.set_title('Geometria 3D do Tokamak')
ax1.set_xlim([-3, 3])
ax1.set_ylim([-3, 3])
ax1.set_zlim([-1.5, 1.5])

# 2. Perfis radiais
ax2 = fig.add_subplot(232)

r = np.linspace(0, 1, 100)

# Perfis típicos
temperature_profile = 15 * (1 - r**2)**2  # keV
density_profile = 1.0 * (1 - r**2)**1.5  # 10^20 m^-3
q_profile = 1.0 + 2.5 * r**2  # Safety factor
pressure_profile = temperature_profile * density_profile  # arbitrário

ax2.plot(r, temperature_profile, 'r-', label='Temperatura (keV)', linewidth=2)
ax2.plot(r, density_profile * 10, 'b-', label='Densidade (10^19 m^-3)', linewidth=2)
ax2.plot(r, q_profile, 'g-', label='Fator q', linewidth=2)
ax2.plot(r, pressure_profile, 'm-', label='Pressão (arb.)', linewidth=2)

ax2.set_xlabel('Raio normalizado (r/a)')
```

```
ax2.set_ylabel('Valor')
ax2.set_title('Perfis Radiais do Plasma')
ax2.legend()
ax2.grid(True, alpha=0.3)

# 3. Diagrama de estabilidade
ax3 = fig.add_subplot(233)

# Limites de estabilidade
beta_N = np.linspace(0, 5, 100)
l_i = np.linspace(0.5, 1.5, 100)

# Limite de Troyon (βN < g)
g_troyon = 3.5  # Fator Troyon típico
stable_region = beta_N[:, None] < g_troyon * l_i[None, :]

# Plota região estável
X, Y = np.meshgrid(l_i, beta_N)
ax3.contourf(X, Y, stable_region.astype(float), levels=[0, 0.5, 1],
           alpha=0.3, colors=['red', 'green'])

# Trajetória típica
l_i_traj = 0.8 + 0.2 * np.sin(np.linspace(0, 2*np.pi, 50))
beta_N_traj = 2.0 + 1.0 * np.sin(np.linspace(0, 2*np.pi, 50))
ax3.plot(l_i_traj, beta_N_traj, 'b-', linewidth=2, label='Trajetória operacional')

ax3.axhline(y=g_troyon, color='r', linestyle='--', label=f'Limite Troyon (βN={g_troyon})')
ax3.set_xlabel('Indutância interna (li)')
ax3.set_ylabel('Beta normalizado (βN)')
ax3.set_title('Diagrama de Estabilidade')
ax3.legend()
ax3.grid(True, alpha=0.3)

# 4. Evolução temporal
ax4 = fig.add_subplot(234)

# Dados simulados (ou lê do CSV)
time = np.linspace(0, 150, 1500)

# Simula sinais típicos
Ip = 15 * (1 - np.exp(-time/30))  # Ramp-up da corrente
Te = 15 * (1 - np.exp(-time/40))  # Aquecimento
q95 = 20 * np.exp(-time/50) + 3   # q95 decaindo para valor operacional
beta_N = 3 * (1 - np.exp(-time/60))  # βN aumentando

ax4.plot(time, Ip, 'b-', label='Ip (MA)', linewidth=2)
ax4.plot(time, Te, 'r-', label='Te (keV)', linewidth=2)
ax4.plot(time, q95, 'g-', label='q95', linewidth=2)
```

```
ax4.plot(time, beta_N, 'm-', label='βN', linewidth=2)

ax4.set_xlabel('Tempo (s)')
ax4.set_ylabel('Valor')
ax4.set_title('Evolução Temporal do Disparo')
ax4.legend(loc='upper right')
ax4.grid(True, alpha=0.3)

# 5. Espaço de parâmetros de operação
ax5 = fig.add_subplot(235)

# Operação normal
normal_Ip = np.random.normal(15, 0.5, 100)
normal_ne = np.random.normal(0.8, 0.1, 100)

# Região de disrupção
disruption_Ip = np.random.normal(15, 1.0, 50)
disruption_ne = np.random.normal(1.2, 0.2, 50)

# Limite de Greenwald
a = 0.6
greenwald_Ip = np.linspace(10, 20, 100)
greenwald_ne = greenwald_Ip / (np.pi * a**2)

ax5.scatter(normal_Ip, normal_ne, c='green', alpha=0.6,
          label='Operação Normal', s=20)
ax5.scatter(disruption_Ip, disruption_ne, c='red', alpha=0.6,
          label='Região de Disrupção', s=20)
ax5.plot(greenwald_Ip, greenwald_ne, 'k--', linewidth=2,
        label='Limite de Greenwald')

ax5.set_xlabel('Corrente de Plasma (MA)')
ax5.set_ylabel('Densidade (10^20 m^-3)')
ax5.set_title('Espaço de Operação')
ax5.legend()
ax5.grid(True, alpha=0.3)

# 6. Sistema de controle
ax6 = fig.add_subplot(236)

# Sinais de controle
control_time = np.linspace(0, 150, 500)

# Correntes nas bobinas
coil1 = 10 * np.sin(control_time/20) + 20
coil2 = 8 * np.sin(control_time/25 + 1) + 15
coil3 = 5 * np.sin(control_time/30 + 2) + 10
```

```
    # Potência de aquecimento
    heating = np.zeros_like(control_time)
    heating[control_time > 30] = 10
    heating[control_time > 60] = 20
    heating[control_time > 90] = 15

    ax6.plot(control_time, coil1, 'b-', label='Bobina PF1 (kA)', linewidth=1.5)
    ax6.plot(control_time, coil2, 'r-', label='Bobina PF2 (kA)', linewidth=1.5)
    ax6.plot(control_time, coil3, 'g-', label='Bobina PF3 (kA)', linewidth=1.5)
    ax6.plot(control_time, heating, 'm-', label='Aquecimento (MW)', linewidth=2)

    ax6.set_xlabel('Tempo (s)')
    ax6.set_ylabel('Sinais de Controle')
    ax6.set_title('Sistema de Controle')
    ax6.legend(loc='upper right')
    ax6.grid(True, alpha=0.3)

    plt.suptitle('SIMULAÇÃO NPE-PSQ - SISTEMA DE CONTROLE DE PLASMA
TOKAMAK',
                 fontsize=16, fontweight='bold')
    plt.tight_layout()
    plt.savefig('results/plasma_simulation_overview.png', dpi=300, bbox_inches='tight')
    plt.show()

def plot_disruption_analysis():
    """Análise detalhada de disrupção"""

    fig = plt.figure(figsize=(15, 10))

    # Simula dados de disrupção
    time = np.linspace(0, 0.1, 1000)  # 100 ms

    # Sinais típicos durante disrupção
    Ip = 15 * np.ones_like(time)
    Te = 10 * np.ones_like(time)

    # Disrupção em t=50ms
    disruption_start = 0.05

    # Thermal Quench (rápido!)
    Te[time > disruption_start] = 10 * np.exp(-(time[time > disruption_start] - disruption_start) /
0.001)

    # Current Quench (mais lento)
    Ip[time > disruption_start] = 15 * np.exp(-(time[time > disruption_start] - disruption_start) /
0.01)

    # Sinais de diagnóstico
```

```
    mhd = 0.1 * np.ones_like(time)
    mhd[time > disruption_start - 0.01] = 0.8  # Aumento pré-disrupção

    radiation = 1 * np.ones_like(time)
    radiation[time > disruption_start] = 10 * np.exp(-(time[time > disruption_start] -
disruption_start) / 0.002)

    # Forças eletromagnéticas
    force = np.zeros_like(time)
    dIp_dt = np.gradient(Ip, time[1]-time[0])
    force[time > disruption_start] = 1e6 * np.abs(dIp_dt[time > disruption_start])  # MN

    # 1. Evolução dos parâmetros do plasma
    ax1 = fig.add_subplot(221)

    ax1.plot(time*1000, Ip, 'b-', label='Ip (MA)', linewidth=2)
    ax1.plot(time*1000, Te, 'r-', label='Te (keV)', linewidth=2)
    ax1.axvline(x=disruption_start*1000, color='k', linestyle='--',
             label='Início da Disrupção')

    ax1.set_xlabel('Tempo (ms)')
    ax1.set_ylabel('Parâmetros do Plasma')
    ax1.set_title('Evolução durante Disrupção')
    ax1.legend(loc='upper right')
    ax1.grid(True, alpha=0.3)

    # 2. Sinais de diagnóstico
    ax2 = fig.add_subplot(222)

    ax2.plot(time*1000, mhd, 'g-', label='Atividade MHD', linewidth=2)
    ax2.plot(time*1000, radiation, 'm-', label='Radiação (MW)', linewidth=2)
    ax2.axvline(x=disruption_start*1000, color='k', linestyle='--')

    ax2.set_xlabel('Tempo (ms)')
    ax2.set_ylabel('Sinais de Diagnóstico')
    ax2.set_title('Diagnóstico Pré-Disrupção')
    ax2.legend(loc='upper right')
    ax2.grid(True, alpha=0.3)

    # 3. Forças eletromagnéticas
    ax3 = fig.add_subplot(223)

    ax3.plot(time*1000, force, 'r-', linewidth=2)
    ax3.fill_between(time*1000, 0, force, alpha=0.3, color='red')
    ax3.axvline(x=disruption_start*1000, color='k', linestyle='--')

    ax3.set_xlabel('Tempo (ms)')
    ax3.set_ylabel('Força (MN)')
```

```
    ax3.set_title('Forças Eletromagnéticas durante Disrupção')
    ax3.grid(True, alpha=0.3)

    # 4. Eficácia da mitigação
    ax4 = fig.add_subplot(224)

    # Com mitigação
    Ip_mitigated = 15 * np.ones_like(time)
    Ip_mitigated[time > disruption_start] = 15 * np.exp(-(time[time > disruption_start] -
disruption_start) / 0.02)

    # Sem mitigação
    Ip_no_mit = 15 * np.ones_like(time)
    Ip_no_mit[time > disruption_start] = 15 * np.exp(-(time[time > disruption_start] -
disruption_start) / 0.005)

    ax4.plot(time*1000, Ip_mitigated, 'g-', label='Com Mitigação', linewidth=2)
    ax4.plot(time*1000, Ip_no_mit, 'r-', label='Sem Mitigação', linewidth=2)
    ax4.axvline(x=disruption_start*1000, color='k', linestyle='--')

    ax4.set_xlabel('Tempo (ms)')
    ax4.set_ylabel('Corrente de Plasma (MA)')
    ax4.set_title('Eficácia do Sistema de Mitigação')
    ax4.legend(loc='upper right')
    ax4.grid(True, alpha=0.3)

    plt.suptitle('ANÁLISE DE DISRUPÇÃO DE PLASMA - NPE-PSQ',
                 fontsize=16, fontweight='bold')
    plt.tight_layout()
    plt.savefig('results/disruption_analysis.png', dpi=300, bbox_inches='tight')
    plt.show()

def create_interactive_3d_plot():
    """Cria visualização 3D interativa com Plotly"""

    # Parâmetros do tokamak
    R0 = 1.8
    a = 0.6
    kappa = 1.8

    # Gera superfície do plasma
    u = np.linspace(0, 2*np.pi, 50)
    v = np.linspace(0, 2*np.pi, 50)
    u, v = np.meshgrid(u, v)

    R = R0 + a * np.cos(u + 0.3 * np.sin(u))
    Z = kappa * a * np.sin(u)
```

```
X = R * np.cos(v)
Y = R * np.sin(v)

# Cria figura
fig = go.Figure()

# Adiciona superfície do plasma
fig.add_trace(go.Surface(
    x=X, y=Y, z=Z,
    colorscale='Reds',
    opacity=0.7,
    name='Plasma',
    showscale=False
))

# Adiciona bobinas toroidais
n_coils = 12
for i in range(n_coils):
    phi = 2*np.pi*i/n_coils
    R_coil = R0 + 1.5*a
    theta = np.linspace(0, 2*np.pi, 50)

    X_coil = R_coil * np.cos(phi) + 0.15*np.cos(theta)*np.cos(phi)
    Y_coil = R_coil * np.sin(phi) + 0.15*np.cos(theta)*np.sin(phi)
    Z_coil = 0.15*np.sin(theta)

    fig.add_trace(go.Scatter3d(
        x=X_coil, y=Y_coil, z=Z_coil,
        mode='lines',
        line=dict(color='blue', width=2),
        name=f'Bobina {i+1}',
        showlegend=False
    ))

# Adiciona linhas de campo magnético
n_lines = 20
for i in range(n_lines):
    phi0 = 2*np.pi*i/n_lines
    s = np.linspace(0, 2*np.pi, 100)

    # Linhas de campo toroidal
    X_field = (R0 + 0.5*a*np.cos(s)) * np.cos(phi0 + 0.1*s)
    Y_field = (R0 + 0.5*a*np.cos(s)) * np.sin(phi0 + 0.1*s)
    Z_field = kappa*a*np.sin(s)

    fig.add_trace(go.Scatter3d(
        x=X_field, y=Y_field, z=Z_field,
        mode='lines',
```

```
            line=dict(color='green', width=1, dash='dot'),
            name='Linhas de Campo',
            showlegend=False
        ))

    # Configura layout
    fig.update_layout(
        title='VISUALIZAÇÃO 3D INTERATIVA - TOKAMAK NPE-PSQ',
        scene=dict(
            xaxis=dict(title='X (m)', range=[-3, 3]),
            yaxis=dict(title='Y (m)', range=[-3, 3]),
            zaxis=dict(title='Z (m)', range=[-1.5, 1.5]),
            aspectmode='manual',
            aspectratio=dict(x=2, y=2, z=1)
        ),
        width=1000,
        height=800
    )

    fig.write_html('results/tokamak_3d_interactive.html')
    print("Visualização 3D interativa salva em: results/tokamak_3d_interactive.html")

    return fig

def main():
    """Função principal"""

    print("\n🔥 GERANDO VISUALIZAÇÕES NPE-PSQ")
    print("==================================\n")

    # Cria diretório de resultados
    import os
    if not os.path.exists('results'):
        os.makedirs('results')

    # Gera gráficos
    print("1. Visão geral da simulação...")
    plot_plasma_equilibrium_3d()

    print("2. Análise de disrupção...")
    plot_disruption_analysis()

    print("3. Criando visualização 3D interativa...")
    create_interactive_3d_plot()

    # Se houver dados CSV, plota também
    try:
        if os.path.exists('results/normal_operation.csv'):
```

```
        print("\n4. Plotando dados da simulação...")

        data = pd.read_csv('results/normal_operation.csv')

        fig, axes = plt.subplots(3, 2, figsize=(14, 10))

        axes[0, 0].plot(data['time_s'], data['Ip_MA'])
        axes[0, 0].set_xlabel('Tempo (s)')
        axes[0, 0].set_ylabel('Ip (MA)')
        axes[0, 0].set_title('Corrente do Plasma')
        axes[0, 0].grid(True, alpha=0.3)

        axes[0, 1].plot(data['time_s'], data['Te_keV'])
        axes[0, 1].set_xlabel('Tempo (s)')
        axes[0, 1].set_ylabel('Te (keV)')
        axes[0, 1].set_title('Temperatura Eletrônica')
        axes[0, 1].grid(True, alpha=0.3)

        axes[1, 0].plot(data['time_s'], data['q95'])
        axes[1, 0].set_xlabel('Tempo (s)')
        axes[1, 0].set_ylabel('q95')
        axes[1, 0].set_title('Fator de Segurança')
        axes[1, 0].grid(True, alpha=0.3)

        axes[1, 1].plot(data['time_s'], data['beta_N'])
        axes[1, 1].set_xlabel('Tempo (s)')
        axes[1, 1].set_ylabel('βN')
        axes[1, 1].set_title('Beta Normalizado')
        axes[1, 1].grid(True, alpha=0.3)

        axes[2, 0].plot(data['time_s'], data['MHD_level'])
        axes[2, 0].set_xlabel('Tempo (s)')
        axes[2, 0].set_ylabel('Nível MHD')
        axes[2, 0].set_title('Atividade MHD')
        axes[2, 0].grid(True, alpha=0.3)

        axes[2, 1].plot(data['time_s'], data['Q_fusion'])
        axes[2, 1].set_xlabel('Tempo (s)')
        axes[2, 1].set_ylabel('Q')
        axes[2, 1].set_title('Ganho de Fusão')
        axes[2, 1].grid(True, alpha=0.3)

        plt.suptitle('DADOS DA SIMULAÇÃO NPE-PSQ', fontsize=16, fontweight='bold')
        plt.tight_layout()
        plt.savefig('results/simulation_data.png', dpi=300, bbox_inches='tight')
        plt.show()

    except Exception as e:
```

```
        print(f"   Nota: Dados CSV não disponíveis ou erro: {e}")

    print("\n✅ Visualizações concluídas!")
    print("\nArquivos gerados:")
    print("  - results/plasma_simulation_overview.png")
    print("  - results/disruption_analysis.png")
    print("  - results/tokamak_3d_interactive.html")
    print("  - results/simulation_data.png (se dados disponíveis)")

if __name__ == "__main__":
    main()
```

## 🛠️ 7. MAKE E EXECUÇÃO: Makefile

```makefile
# Makefile para simulação NPE-PSQ
CC = gcc
CFLAGS = -Wall -Wextra -O3 -std=c99 -lm -D_XOPEN_SOURCE=500
TARGET = npe_psq_simulator
SRC = plasma_physics.c plasma_controller.c plasma_safety.c npe_simulation.c
HEADERS = npe_config.h

# Flags para desenvolvimento
DEBUG_FLAGS = -g -DDEBUG -fsanitize=address
OPT_FLAGS = -O3 -march=native -ffast-math
PROFILE_FLAGS = -pg

.PHONY: all clean run plot profile debug help

all: $(TARGET)

$(TARGET): $(SRC) $(HEADERS)
	$(CC) $(CFLAGS) $(OPT_FLAGS) -o $(TARGET) $(SRC)

debug: $(SRC) $(HEADERS)
	$(CC) $(CFLAGS) $(DEBUG_FLAGS) -o $(TARGET)_debug $(SRC)

profile: $(SRC) $(HEADERS)
	$(CC) $(CFLAGS) $(PROFILE_FLAGS) -o $(TARGET)_profile $(SRC)

run: $(TARGET)
	@echo "Executando simulação NPE-PSQ..."
	./$(TARGET)

plot:
	@echo "Gerando visualizações..."
	python3 plasma_visualization.py
```

```
test: $(TARGET)
	@echo "Executando teste rápido..."
	@./$(TARGET) 2>&1 | grep -A10 "RESULTADOS"

clean:
	rm -f $(TARGET) $(TARGET)_debug $(TARGET)_profile gmon.out
	rm -rf results/*.csv results/*.png results/*.html

install-deps:
	# Instala dependências Python
	pip3 install numpy matplotlib pandas plotly

install-system-deps:
	# Dependências do sistema (Ubuntu/Debian)
	sudo apt-get install gcc python3 python3-pip python3-tk

setup: install-system-deps install-deps
	@echo "Configuração completa"

docs:
	# Gera documentação (requer Doxygen)
	doxygen Doxyfile
	@echo "Documentação gerada em docs/html/"

benchmark: $(TARGET)
	@echo "Benchmarking simulação..."
	time ./$(TARGET) > /dev/null

help:
	@echo "Comandos disponíveis para NPE-PSQ:"
	@echo ""
	@echo "  make all        - Compila o simulador com otimização"
	@echo "  make debug      - Compila para depuração"
	@echo "  make profile    - Compila para profiling"
	@echo "  make run        - Executa a simulação"
	@echo "  make plot       - Gera visualizações (requer Python)"
	@echo "  make test       - Executa teste rápido"
	@echo "  make benchmark  - Mede tempo de execução"
	@echo "  make clean      - Remove arquivos gerados"
	@echo "  make setup      - Instala dependências do sistema"
	@echo "  make docs       - Gera documentação (requer Doxygen)"
	@echo "  make help       - Mostra esta ajuda"
	@echo ""
	@echo "Fluxo recomendado:"
	@echo "  1. make setup    # Instala dependências"
	@echo "  2. make all      # Compila"
	@echo "  3. make run      # Executa simulação"
```

```
	@echo "  4. make plot     # Gera gráficos"
```

🚀 8. COMO EXECUTAR:

```bash
# 1. Clone e prepare
git clone https://github.com/seu-usuario/npe-psq-simulation.git
cd npe-psq-simulation

# 2. Instale dependências
make setup

# 3. Compile
make all

# 4. Execute
make run

# 5. Visualize resultados
make plot

# 6. Execute tudo
make clean all run plot
```

📊 CARACTERÍSTICAS DO SISTEMA NPE-PSQ:

🔬 Física Implementada:

1. Equilíbrio MHD (Grad-Shafranov simplificado)
2. Perfis de plasma (temperatura, densidade, corrente)
3. Estabilidade (critérios Troyon, Greenwald, q95)
4. Disrupções (thermal quench, current quench, VDEs)
5. Transporte (confinamento de energia, scaling ITER)

🎮 Sistema de Controle:

1. MPC avançado para estabilidade do plasma
2. Máquina de estados para operação do tokamak
3. Controle de bobinas PF, vertical, horizontal
4. Controle de aquecimento (ECRH, ICRH, NBI)
5. Controle de combustível e densidade

⚠️ Sistema de Segurança:

1. Detecção precoce de disrupções
2. Mitigação ativa (MGI, pellets, killer pulse)

3. Interlock system multi-camadas
4. Predição em tempo real usando múltiplos indicadores
5. Forças eletromagnéticas durante disrupção

📈 Visualização:

1. 3D interativo do tokamak (Plotly)
2. Evolução temporal dos parâmetros
3. Diagramas de estabilidade
4. Análise de disrupção
5. Espaço de operação

🎯 APLICAÇÕES E USOS:

1. Projeto de tokamaks (ITER, DEMO, reactors comerciais)
2. Desenvolvimento de algoritmos de controle
3. Teste de estratégias de mitigação de disrupções
4. Treinamento de operadores de reatores
5. Pesquisa acadêmica em física de plasmas

🔬 PRÓXIMOS PASSOS (MELHORIAS):

1. Modelos mais realistas (transporte não-difusivo, turbulência)
2. Diagnósticos avançados (interferometria, espalhamento Thomson)
3. Controle adaptativo (machine learning para previsão de disrupções)
4. Otimização multi-objetivo (Q vs estabilidade vs vida útil)
5. Integração com códigos reais (ASTRA, TRANSP, DINA)
6. Hardware-in-the-loop para teste de controladores reais

Este sistema NPE-PSQ é uma ferramenta poderosa para simulação e desenvolvimento de sistemas de controle para reatores de fusão nuclear, especialmente crítico para operação segura de tokamaks de próxima geração como o ITER e DEMO!