“””

AI-Optimized p-B¹¹ Fusion Systems: Computational Implementation

Author: Mustafa Serkan Taşkoyan

Date: August 2025

This code supplement accompanies the theoretical paper on p-B¹¹ fusion systems.

All calculations were developed with collaborative assistance from xAI’s Grok AI.

“””

import numpy as np

import matplotlib.pyplot as plt

from scipy import constants, optimize, integrate

from scipy.stats import norm

import pandas as pd

# =============================================================================

# 1. FUNDAMENTAL p-B¹¹ FUSION PARAMETERS

# =============================================================================

class PB11FusionPhysics:

“”“Core physics calculations for p-B¹¹ fusion”””

```

def \_\_init\_\_(self):

# Fundamental constants

self.Q\_reaction = 8.68e6 \* constants.eV # Total energy release (J)

self.N\_alpha = 3 # Number of alpha particles produced

self.E\_alpha = self.Q\_reaction / self.N\_alpha # Energy per alpha (J)

self.m\_alpha = 4 \* constants.atomic\_mass # Alpha particle mass (kg)

self.m\_proton = constants.proton\_mass

self.m\_boron = 11 \* constants.atomic\_mass

# Calculate alpha particle velocity

self.v\_alpha = np.sqrt(2 \* self.E\_alpha / self.m\_alpha)

def alpha\_velocity(self):

"""Calculate alpha particle velocity"""

return self.v\_alpha

def cross\_section(self, energy\_keV):

"""

Approximate p-B¹¹ fusion cross-section

Based on Nevins & Swain parameterization

"""

if energy\_keV < 10:

return 0

# Simplified model - actual cross-section is much more complex

sigma\_peak = 1e-31 # Peak cross-section (m²)

E\_peak = 600 # Peak energy (keV)

width = 200 # Width parameter

sigma = sigma\_peak \* np.exp(-(energy\_keV - E\_peak)\*\*2 / (2 \* width\*\*2))

return max(sigma, 1e-35) # Minimum cross-section floor

def fusion\_rate(self, density\_p, density\_B, temperature\_keV):

"""Calculate fusion reaction rate"""

sigma = self.cross\_section(temperature\_keV)

# Simplified rate calculation

rate = density\_p \* density\_B \* sigma \* self.v\_alpha

return rate

```

# =============================================================================

# 2. MAGNETIC CONFINEMENT AND COLLIMATION

# =============================================================================

def alpha\_trajectory\_magnetic(B\_field, initial\_angle, time\_steps=1000):

“””

Simulate alpha particle trajectory in uniform magnetic field

Simplified 2D cyclotron motion

“””

dt = 1e-9 # Time step (s)

t\_max = 1e-6 # Total simulation time (s)

t = np.linspace(0, t\_max, time\_steps)

```

# Cyclotron frequency

q\_alpha = 2 \* constants.e # Alpha charge

m\_alpha = 4 \* constants.atomic\_mass

omega\_c = q\_alpha \* B\_field / m\_alpha

# Initial conditions

v0 = np.sqrt(2 \* 2.89e6 \* constants.eV / m\_alpha) # Alpha velocity

vx0 = v0 \* np.cos(initial\_angle)

vy0 = v0 \* np.sin(initial\_angle)

# Cyclotron motion equations

x = (vx0 / omega\_c) \* np.sin(omega\_c \* t) + (vy0 / omega\_c) \* (1 - np.cos(omega\_c \* t))

y = (vx0 / omega\_c) \* (1 - np.cos(omega\_c \* t)) - (vy0 / omega\_c) \* np.sin(omega\_c \* t)

return t, x, y

```

def monte\_carlo\_collimation(B\_field=1.0, n\_particles=10000, collector\_radius=0.1):

“””

Monte Carlo simulation of alpha particle collimation efficiency

“””

np.random.seed(42) # For reproducibility

```

# Random initial angles (isotropic emission)

theta = np.random.uniform(0, 2\*np.pi, n\_particles)

phi = np.random.uniform(0, np.pi, n\_particles)

# Cyclotron radius for alpha particles

q\_alpha = 2 \* constants.e

m\_alpha = 4 \* constants.atomic\_mass

v\_alpha = np.sqrt(2 \* 2.89e6 \* constants.eV / m\_alpha)

r\_cyclotron = m\_alpha \* v\_alpha / (q\_alpha \* B\_field)

# Calculate final positions after one cyclotron period

x\_final = r\_cyclotron \* np.sin(phi) \* np.cos(theta)

y\_final = r\_cyclotron \* np.sin(phi) \* np.sin(theta)

r\_final = np.sqrt(x\_final\*\*2 + y\_final\*\*2)

# Count particles within collector radius

collected = r\_final < collector\_radius

efficiency = np.sum(collected) / n\_particles

return efficiency, r\_final, collected

```

# =============================================================================

# 3. MHD STABILITY ANALYSIS

# =============================================================================

def mhd\_growth\_rate(k\_perp, B0=1.0, beta=0.1, density=1e20):

“””

Calculate MHD instability growth rates

Simplified interchange mode analysis

“””

# Plasma parameters

T\_alpha = 2.89e6 \* constants.eV / constants.k # Alpha temperature (K)

omega\_ci = constants.e \* B0 / constants.proton\_mass # Ion cyclotron frequency

v\_thermal = np.sqrt(constants.k \* T\_alpha / constants.proton\_mass)

```

# Simplified growth rate for interchange modes

# Real calculation would require solving full MHD equations

gamma = np.sqrt(beta) \* omega\_ci \* (k\_perp \* v\_thermal / omega\_ci)\*\*0.5

return gamma

```

def stability\_analysis():

“”“Comprehensive MHD stability analysis”””

k\_values = np.logspace(-2, 2, 100) # Wavenumber range

B\_fields = [0.5, 1.0, 2.0, 5.0] # Tesla

```

results = {}

for B in B\_fields:

growth\_rates = [mhd\_growth\_rate(k, B0=B) for k in k\_values]

results[f'B={B}T'] = {

'k': k\_values,

'gamma': growth\_rates,

'max\_growth': max(growth\_rates)

}

return results

```

# =============================================================================

# 4. AI-ASSISTED OPTIMIZATION

# =============================================================================

def ai\_optimize\_confinement(target\_efficiency=0.8):

“””

AI-guided optimization of magnetic confinement parameters

Uses scipy.optimize with physics-based constraints

“””

```

def objective\_function(params):

"""Multi-objective optimization function"""

B\_field, collector\_radius = params

# Calculate collimation efficiency

eff, \_, \_ = monte\_carlo\_collimation(B\_field, n\_particles=1000,

collector\_radius=collector\_radius)

# Calculate power cost (B² scaling)

power\_penalty = (B\_field / 5.0)\*\*2 # Normalized to 5T reference

# Calculate stability (want low growth rates)

max\_growth = max([mhd\_growth\_rate(k, B0=B\_field) for k in [0.1, 1.0, 10.0]])

stability\_penalty = max\_growth / 1e6 # Normalize to 10⁶ s⁻¹

# Combined objective: maximize efficiency, minimize power and instability

objective = -(eff - 0.1\*power\_penalty - 0.05\*stability\_penalty)

return objective

# Optimization constraints and bounds

initial\_guess = [1.0, 0.1] # [B\_field (T), collector\_radius (m)]

bounds = [(0.1, 5.0), (0.01, 0.5)] # Physical limits

# Run optimization

result = optimize.minimize(objective\_function, initial\_guess,

bounds=bounds, method='L-BFGS-B')

return result

```

# =============================================================================

# 5. SYSTEM PERFORMANCE CALCULATIONS

# =============================================================================

def calculate\_thrust\_power(mass\_flow\_rate=1e-6, exhaust\_velocity=6e5):

“”“Calculate thrust and required power for space propulsion”””

thrust = mass\_flow\_rate \* exhaust\_velocity # Newton’s law

```

# Power calculation (simplified)

# Real calculation requires detailed thermodynamic analysis

kinetic\_power = 0.5 \* mass\_flow\_rate \* exhaust\_velocity\*\*2

efficiency = 0.35 # Assumed direct energy conversion efficiency

required\_power = kinetic\_power / efficiency

return thrust, required\_power

```

def energy\_conversion\_efficiency(alpha\_energy, collector\_efficiency=0.8,

conversion\_efficiency=0.6):

“”“Calculate direct energy conversion (DEC) efficiency”””

# Multi-stage energy conversion

stage1\_eff = collector\_efficiency # Magnetic collimation

stage2\_eff = conversion\_efficiency # Electrostatic conversion

```

total\_efficiency = stage1\_eff \* stage2\_eff

useful\_energy = alpha\_energy \* total\_efficiency

return total\_efficiency, useful\_energy

```

# =============================================================================

# 6. VALIDATION AND BENCHMARKING

# =============================================================================

def validate\_calculations():

“”“Validate key calculations against known results”””

print(”=== p-B¹¹ Fusion Calculation Validation ===”)

```

# Initialize physics

physics = PB11FusionPhysics()

# Test 1: Alpha particle velocity

v\_calc = physics.alpha\_velocity()

v\_expected = 1.18e7 # Expected value (m/s)

error = abs(v\_calc - v\_expected) / v\_expected \* 100

print(f"Alpha velocity: {v\_calc:.2e} m/s (Error: {error:.1f}%)")

# Test 2: Energy per alpha

E\_alpha\_MeV = physics.E\_alpha / (1e6 \* constants.eV)

print(f"Energy per alpha: {E\_alpha\_MeV:.2f} MeV")

# Test 3: Collimation efficiency

eff, \_, \_ = monte\_carlo\_collimation(B\_field=1.0, n\_particles=5000)

print(f"Collimation efficiency (1T): {eff:.1%}")

# Test 4: MHD stability

max\_growth = max([mhd\_growth\_rate(k, B0=1.0) for k in [0.1, 1.0, 10.0]])

print(f"Max MHD growth rate: {max\_growth:.2e} s⁻¹")

print("=== Validation Complete ===\n")

```

def benchmark\_against\_fission():

“”“Compare p-B¹¹ performance to fission systems”””

print(”=== p-B¹¹ vs Fission Benchmark ===”)

```

# Asteroid deflection scenario

asteroid\_mass = 8.835e14 # kg (example asteroid)

asteroid\_velocity = 68e3 # m/s

deflection\_time = 365 \* 24 \* 3600 # 1 year (s)

# p-B¹¹ system

pb11\_thrust, pb11\_power = calculate\_thrust\_power()

pb11\_delta\_v = pb11\_thrust \* deflection\_time / asteroid\_mass

# Simplified fission comparison (lower thrust/weight ratio)

fission\_thrust = pb11\_thrust \* 0.1 # Assumed 10x lower

fission\_delta\_v = fission\_thrust \* deflection\_time / asteroid\_mass

print(f"p-B¹¹ - Thrust: {pb11\_thrust:.1f} N, ΔV: {pb11\_delta\_v:.3f} m/s")

print(f"Fission - Thrust: {fission\_thrust:.1f} N, ΔV: {fission\_delta\_v:.3f} m/s")

print(f"p-B¹¹ advantage: {pb11\_delta\_v/fission\_delta\_v:.1f}x\n")

```

# =============================================================================

# 7. VISUALIZATION AND RESULTS

# =============================================================================

def plot\_optimization\_results():

“”“Create visualization of optimization results”””

fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 10))

```

# 1. Collimation efficiency vs B-field

B\_range = np.linspace(0.1, 5.0, 20)

efficiencies = []

for B in B\_range:

eff, \_, \_ = monte\_carlo\_collimation(B, n\_particles=1000)

efficiencies.append(eff)

ax1.plot(B\_range, efficiencies, 'b-o', linewidth=2)

ax1.set\_xlabel('Magnetic Field (T)')

ax1.set\_ylabel('Collimation Efficiency')

ax1.set\_title('Alpha Particle Collimation')

ax1.grid(True, alpha=0.3)

# 2. MHD stability

stability\_data = stability\_analysis()

for label, data in stability\_data.items():

ax2.loglog(data['k'], data['gamma'], label=label, linewidth=2)

ax2.set\_xlabel('Wavenumber k (m⁻¹)')

ax2.set\_ylabel('Growth Rate (s⁻¹)')

ax2.set\_title('MHD Stability Analysis')

ax2.legend()

ax2.grid(True, alpha=0.3)

# 3. Cross-section vs energy

energy\_range = np.linspace(10, 1000, 100)

physics = PB11FusionPhysics()

cross\_sections = [physics.cross\_section(E) for E in energy\_range]

ax3.semilogy(energy\_range, cross\_sections, 'r-', linewidth=2)

ax3.set\_xlabel('Energy (keV)')

ax3.set\_ylabel('Cross-section (m²)')

ax3.set\_title('p-B¹¹ Fusion Cross-section')

ax3.grid(True, alpha=0.3)

# 4. Particle trajectory example

t, x, y = alpha\_trajectory\_magnetic(1.0, np.pi/4)

ax4.plot(x\*1e3, y\*1e3, 'g-', linewidth=2)

ax4.set\_xlabel('x (mm)')

ax4.set\_ylabel('y (mm)')

ax4.set\_title('Alpha Trajectory (B=1T)')

ax4.axis('equal')

ax4.grid(True, alpha=0.3)

plt.tight\_layout()

plt.savefig('pb11\_optimization\_results.png', dpi=300, bbox\_inches='tight')

plt.show()

```

# =============================================================================

# 8. MAIN EXECUTION AND RESULTS

# =============================================================================

if \*\*name\*\* == “\*\*main\*\*”:

print(“AI-Optimized p-B¹¹ Fusion Systems - Computational Analysis”)

print(”=” \* 60)

```

# Run validation

validate\_calculations()

# Run benchmark

benchmark\_against\_fission()

# AI optimization

print("=== AI-Guided Optimization ===")

opt\_result = ai\_optimize\_confinement()

if opt\_result.success:

B\_opt, r\_opt = opt\_result.x

print(f"Optimal B-field: {B\_opt:.2f} T")

print(f"Optimal collector radius: {r\_opt:.3f} m")

print(f"Optimization objective: {-opt\_result.fun:.3f}")

else:

print("Optimization failed - using default parameters")

# Generate performance summary

print("\n=== System Performance Summary ===")

physics = PB11FusionPhysics()

eff, r\_final, \_ = monte\_carlo\_collimation()

thrust, power = calculate\_thrust\_power()

dec\_eff, \_ = energy\_conversion\_efficiency(physics.E\_alpha)

print(f"Alpha particle velocity: {physics.v\_alpha:.2e} m/s")

print(f"Collimation efficiency: {eff:.1%}")

print(f"System thrust: {thrust:.1f} N")

print(f"Required power: {power:.1f} MW")

print(f"DEC efficiency: {dec\_eff:.1%}")

# Create visualizations

print("\nGenerating optimization plots...")

plot\_optimization\_results()

print("\n=== Analysis Complete ===")

print("Results saved to 'pb11\_optimization\_results.png'")

print("Code available for peer review and collaboration")

```