Hybrid Rocket Engine Combustion Chamber Performance Analysis System

Technical Documentation v1.0

Edgardo Rossi on Cristian Casalanguida's code
Documentation compiled up until 19/10/2025

October 19, 2025

Abstract

This document provides comprehensive technical documentation for a numerical simulation framework designed to analyze hybrid rocket engine combustion chamber performance. The system integrates fluid injection modeling, thermochemical equilibrium calculations, and iterative pressure convergence algorithms to predict engine performance across parametric design spaces. The framework is particularly suited for self-pressurizing oxidizer systems (e.g., nitrous oxide) with solid fuel grains (e.g., paraffin wax).

Contents

1 Introduction

1.1 Overview

The hybrid rocket engine simulation framework consists of three primary modules:

1. **Injection Module (`PyInjection.py`)**: Models two-phase flow through injector orifices using the Dyer model with Self-Pressurizing Injector (SPI) and Homogeneous Equilibrium Model (HEM) approaches.
2. **Performance Module (`performance_singlepoint.py`)**: Calculates thermochemical and propulsive performance parameters at a single operating point using NASA CEA (Chemical Equilibrium with Applications).
3. **Chamber Pressure Convergence Module (`pressure_convergence.py`)**: Implements Newton-like iterative methods to solve for equilibrium chamber pressure and performs parametric design space exploration.

1.2 Physical Problem

The simulation addresses the coupled nonlinear problem of hybrid rocket combustion:

- Oxidizer mass flow through injector: $\dot{m}_{ox} = f(p_{tank}, p_c, T_{tank})$
- Fuel regression rate (Marxman-Gilbert): $\dot{r} = a \cdot G_{ox}^n$
- Mass conservation at throat: $\dot{m}_{total} = \frac{p_c A_t}{c^*}$
- Thermochemical equilibrium determines c^*, γ, and performance

The chamber pressure p_c must be found iteratively to satisfy mass balance.

1.3 Key Assumptions

- Steady-state operation
- Isothermal flow in feed lines
- Ideal gas behavior in combustion products
- One-dimensional flow in nozzle
- Instantaneous chemical equilibrium
- Uniform regression along fuel grain

2 Module 1: Injection System Model

2.1 Theoretical Background

The injection model implements the methodology from Waxman et al. (Stanford University) for self-pressurizing oxidizers. The model accounts for phase change phenomena critical in nitrous oxide systems.

2.1.1 Flow Regimes

Three distinct flow regimes are identified based on vapor pressure p_v:

1. **All-Liquid Flow** ($p_v < p_c$):

$$\dot{m}'' = C_D \sqrt{2\rho_{SPI}(p_1 - p_2)} \tag{1}$$

   where ρ_{SPI} is the saturated liquid density.

2. **Two-Phase Flow** ($p_c < p_v < p_1$):

$$\dot{m}'' = \frac{k \cdot \dot{m}''_{SPI} + \dot{m}''_{HEM}}{k+1} \tag{2}$$

   where:

$$k = \sqrt{\frac{p_1 - p_2}{p_v - p_2}} \tag{3}$$

$$\dot{m}''_{HEM} = C_D \rho_2 \sqrt{2|h_1 - h_2|} \tag{4}$$

3. **All-Gas Flow** ($p_v < p_1$):

$$\dot{m}'' = \frac{p_1}{\sqrt{RT}} \cdot \Gamma(\gamma, p_2/p_1) \tag{5}$$

   with critical flow function:

$$\Gamma = \sqrt{\gamma \left(\frac{2}{\gamma+1}\right)^{\frac{\gamma+1}{\gamma-1}}} \tag{6}$$

2.2 Implementation Details

2.2.1 Class Structure

```
class Injector:
    def __init__(self, fluid)
    def injection_area(self, D, n)
    def massflow(self, p1, p2, T, cD)
```

2.2.2 Key Methods

`__init__(fluid)` Initializes injector with specified fluid from CoolProp database.

Parameters:

- `fluid` (str): CoolProp fluid name (e.g., "NitrousOxide")

`injection_area(D, n)` Calculates total injection area for circular orifices.

Parameters:

- `D` (float): Orifice diameter [m]
- `n` (int): Number of orifices

Formula:

$$A_{inj} = n \cdot \frac{\pi D^2}{4} \tag{7}$$

`massflow(p1, p2, T, cD)` Computes mass flux through injector accounting for phase transitions.

Parameters:

- `p1` (float): Upstream pressure [Pa]
- `p2` (float): Downstream pressure [Pa]
- `T` (float): Fluid temperature [K]
- `cD` (float): Discharge coefficient [-]

Returns:

- `self.mdot` (float): Mass flux [kg/(s·m²)]

2.3 Usage Example

```
import PyInjection as injection

# Initialize N2O injector
ox = injection.Injector('NitrousOxide')

# Define geometry
ox.injection_area(D=0.0127, n=4)  # 4 holes of 12.7mm

# Calculate mass flow
ox.massflow(p1=5.5e6, p2=4.3e6, T=288, cD=0.8)
mdot_total = ox.mdot * ox.A  # [kg/s]
```

3 Module 2: Single-Point Performance Analysis

3.1 Theoretical Framework

This module calculates thermochemical and propulsive performance at a specified chamber pressure.

3.1.1 Fuel Regression Rate

The classical Marxman-Gilbert correlation:

$$\dot{r} = a \cdot G_{ox}^{n} \tag{8}$$

where:

- $\dot{r}$ is regression rate [m/s]
- $G_{ox} = \dot{m}_{ox}/A_{port}$ is oxidizer mass flux [kg/(s·m²)]
- a, n are empirical constants (e.g., paraffin: $a = 0.17 \times 10^{-3}$, $n = 0.5$)

Fuel mass flow:

$$\dot{m}_{fuel} = \rho_{fuel} \cdot A_b \cdot \dot{r} \tag{9}$$

3.1.2 Mixture Ratio

$$MR = \frac{\dot{m}_{ox}}{\dot{m}_{fuel}} \tag{10}$$

3.1.3 Nozzle Performance

From CEA thermochemical equilibrium calculations:

- Characteristic velocity: $c^* = \sqrt{\frac{RT_c}{\gamma}} \left(\frac{\gamma+1}{2}\right)^{\frac{\gamma+1}{2(\gamma-1)}}$ [m/s]
- Vacuum thrust coefficient: $C_{F,vac}$ [-]
- Thrust coefficient: $C_F = C_{F,vac} - \varepsilon \frac{p_{amb}}{p_c}$ [-]
- Specific impulse: $I_{sp} = \frac{c^* C_F}{g_0}$ [s]

3.1.4 Expansion Ratio

For adaptive nozzles (`eps = "adapt"`):

$$\varepsilon = \frac{A_e}{A_t} = \frac{\Gamma}{\sqrt{\frac{2\gamma}{\gamma-1}\left[\left(\frac{p_e}{p_c}\right)^{2/\gamma} - \left(\frac{p_e}{p_c}\right)^{(\gamma+1)/\gamma}\right]}} \tag{11}$$

where $p_e = p_{amb}$ for perfect expansion.

3.2 Key Functions

3.2.1 `calculate_performance(...)`

Primary function that computes all performance metrics at a given p_c.

Inputs:

- `Ainj`: Injection area [m^2]
- `Aport`: Port cross-sectional area [m^2]
- `Ab`: Fuel burning surface area [m^2]
- `eps`: Expansion ratio (float) or "adapt"
- `ptank`: Tank pressure [Pa]
- `Ttank`: Tank temperature [K]
- `pc`: Chamber pressure [Pa]
- `CD`: Discharge coefficient [-]
- `a`, `n`: Regression parameters
- `rho_fuel`: Fuel density [kg/m^3]
- `oxidizer`: Dictionary with oxidizer properties
- `fuel`: Dictionary with fuel properties
- `pamb`: Ambient pressure [Pa] (default: 0)
- `gamma0`: Initial γ guess (default: 1.3)

Outputs (17 parameters):

1. `p_inj`: Injection pressure [Pa]
2. `mdot_ox`: Oxidizer mass flow [kg/s]
3. `mdot_fuel`: Fuel mass flow [kg/s]
4. `mdot`: Total mass flow [kg/s]
5. `Gox`: Oxidizer mass flux [$kg/(s \cdot m^2)$]
6. `r`: Regression rate [m/s]
7. `MR`: Mixture ratio [-]
8. `Tc`: Chamber temperature [K]
9. `MW`: Molecular weight [kg/kmol]
10. `gamma`: Specific heat ratio [-]
11. `eps_out`: Actual expansion ratio [-]
12. `cs`: Characteristic velocity [m/s]
13. `CF_vac`: Vacuum thrust coefficient [-]

14. `CF`: Thrust coefficient [-]
15. `Ivac`: Vacuum specific impulse [s]
16. `Is`: Specific impulse [s]
17. `flag_performance`: Convergence flag (0=success, 1=failure)

3.2.2 `pressure_fun(...)`

Residual function for Newton iteration:

$$F(p_c) = \frac{\dot{m} \cdot c^*(p_c)}{A_t} - p_c \tag{12}$$

The zero of this function represents mass balance equilibrium.

3.3 Propellant Specification Format

3.3.1 Oxidizer Dictionary

```
oxidizer = {
    "OxidizerCP": "NitrousOxide",  # CoolProp name
    "OxidizerCEA": "N2O",           # CEA name
    "Weight fraction": "100",
    "Exploded Formula": "",
    "Temperature [K]": "",
    "Specific Enthalpy [kj/mol]": ""
}
```

3.3.2 Fuel Dictionary

```
fuel = {
    "Fuels": ["paraffin"],
    "Weight fraction": ["100"],
    "Exploded Formula": ["C 73 H 124"],
    "Temperature [K]": [533.0],
    "Specific Enthalpy [kj/mol]": [-1860.6]
}
```

4 Module 3: Chamber Pressure Convergence

4.1 Mathematical Formulation

4.1.1 Nonlinear System

The chamber pressure must satisfy:

$$\dot{m}_{total}(p_c) = \frac{p_c A_t}{c^*(p_c, MR(p_c))} \tag{13}$$

where all right-hand terms depend implicitly on p_c through:

$$\dot{m}_{ox}(p_c) = f_{inj}(p_{tank} - \Delta p_{lines}, p_c) \tag{14}$$

$$\dot{m}_{fuel}(p_c) = \rho_{fuel} A_b a \left(\frac{\dot{m}_{ox}}{A_{port}}\right)^n \tag{15}$$

$$MR(p_c) = \frac{\dot{m}_{ox}(p_c)}{\dot{m}_{fuel}(p_c)} \tag{16}$$

4.1.2 Newton-Raphson Method

Iterative scheme with adaptive damping:

$$p_c^{(k+1)} = p_c^{(k)} - k_{Newton} \frac{F(p_c^{(k)})}{F'(p_c^{(k)})} \tag{17}$$

where:

- $k_{Newton} \leq 1$ is damping factor (reduces if bounds violated)
- $F'(p_c) \approx \frac{F(p_c+\Delta p)-F(p_c)}{\Delta p}$ (finite difference)
- $\Delta p = 10$ Pa

Convergence criterion:

$$|F(p_c)| < \tau = 0.1 \text{ Pa} \tag{18}$$

4.2 Algorithm Flow

4.2.1 Initial Guess Calculation

Function: `starting_pressure(...)`

Strategy:

1. Define search range: $p_c \in [p_{amb}, 0.9 \cdot p_{tank}]$
2. Evaluate $F(p_c)$ at 50 uniformly spaced points
3. Select p_c with minimum $|F(p_c)|$
4. Check for sign change (zero crossing)
5. Return 0 if no valid solution exists

Returns:

- $p_c^{(0)}$ [Pa] if solvable
- 0 if no solution exists

4.2.2 Iterative Solver

Function: `get_pressure(...)`

Algorithm:

1. Initialize: $k_{Newton} = 1.0$, $n_{iter} = 0$
2. Get initial guess: $p_c^{(0)} =$ `starting_pressure(...)`
3. If $p_c^{(0)} = 0$: return failure
4. While $|F(p_c)| > \tau$ and $n_{iter} < 100$:
   (a) Compute: $F' = [F(p_c + 10) - F(p_c)]/10$
   (b) Update: $p_c^{new} = p_c - k_{Newton} \cdot F/F'$
   (c) If $p_c^{new} < p_{amb}$: $p_c^{new} = \max(0.2p_{tank}, 1.5p_{amb})$, $k_{Newton} -= 0.05$
   (d) If $p_c^{new} \geq 0.9p_{tank}$: $p_c^{new} = 0.75p_{tank}$, $k_{Newton} -= 0.05$
   (e) Calculate performance at p_c^{new}
   (f) Update $\gamma_0 = \gamma(p_c^{new})$
   (g) $n_{iter} += 1$
5. Return: p_c, $F(p_c)$, n_{iter}, γ_0

Convergence Flags:

- 0: Successfully converged
- 1: Pressure iteration diverged (reached max iterations)
- -1: CEA calculation failed
- 2: Both pressure and CEA failed
- 10: No initial solution found

4.3 Parametric Design Space Exploration

4.3.1 Function: `full_range_simulation(...)`

Performs exhaustive search over dimensionless design parameters:

- D_{port}/D_t: Port-to-throat diameter ratio
- D_{inj}/D_t: Injector-to-throat diameter ratio
- L_c/D_t: Chamber length-to-throat diameter ratio

Normalization: All dimensions scaled by throat diameter $D_t = 1$ m (arbitrary reference).
Geometric Relations:

$$A_{port} = \frac{\pi}{4}(D_{port})^2 \tag{19}$$

$$A_{inj} = \frac{\pi}{4}(D_{inj})^2 \tag{20}$$

$$A_b = \pi D_{port} L_c \tag{21}$$

$$A_t = \frac{\pi}{4}(D_t)^2 = \frac{\pi}{4} \tag{22}$$

Output Arrays: All outputs are 3D arrays with shape (`n_Dport`, `n_Dinj`, `n_Lc`).
19 output arrays:

1. `pc_array`: Chamber pressure [Pa]
2. `Fpc_array`: Residual [Pa]
3. `p_inj_array`: Injection pressure [Pa]
4. `mdot_ox_array`: Oxidizer flow [kg/s]
5. `mdot_fuel_array`: Fuel flow [kg/s]
6. `mdot_array`: Total flow [kg/s]
7. `Gox_array`: Oxidizer flux [kg/(s·m²)]
8. `r_array`: Regression rate [m/s]
9. `MR_array`: Mixture ratio [-]
10. `eps_array`: Expansion ratio [-]
11. `Tc_array`: Chamber temperature [K]
12. `MW_array`: Molecular weight [kg/kmol]
13. `gamma_array`: γ [-]
14. `cs_array`: c^* [m/s]
15. `CF_vac_array`: $C_{F,vac}$ [-]
16. `CF_array`: C_F [-]
17. `Ivac_array`: Vacuum I_{sp} [s]
18. `Is_array`: I_{sp} [s]
19. `flag_array`: Convergence status

5 Integration Example

5.1 Complete Workflow

```
import numpy as np
import pressure_convergence as pconv

# Define parameter ranges
Dport_Dt_range = np.arange(3.5, 5.0, 0.5)  # [-]
Dinj_Dt_range = np.arange(0.8, 1.0, 0.05)  # [-]
Lc_Dt_range = np.arange(8, 10, 1)          # [-]

# Operating conditions
eps = "adapt"
ptank = 55e5   # Pa (55 bar)
Ttank = 288    # K
pamb = 1e5     # Pa (1 bar)
CD = 0.8

# Fuel properties (paraffin)
a = 0.17e-3
n = 0.5
rho_fuel = 850  # kg/m^3

# Define propellants
oxidizer = {
    "OxidizerCP": "NitrousOxide",
    "OxidizerCEA": "N2O",
    "Weight fraction": "100",
    "Exploded Formula": "",
    "Temperature [K]": "",
    "Specific Enthalpy [kj/mol]": ""
}

fuel = {
    "Fuels": ["paraffin"],
    "Weight fraction": ["100"],
    "Exploded Formula": ["C 73 H 124"],
    "Temperature [K]": [533.0],
    "Specific Enthalpy [kj/mol]": [-1860.6]
}

# Run simulation
results = pconv.full_range_simulation(
    Dport_Dt_range, Dinj_Dt_range, Lc_Dt_range,
    eps, ptank, Ttank, CD, a, n, rho_fuel,
    oxidizer, fuel, pamb
)

# Unpack results
(pc, Fpc, p_inj, mdot_ox, mdot_fuel, mdot,
 Gox, r, MR, eps_out, Tc, MW, gamma, cs,
 CF_vac, CF, Ivac, Is, flags) = results

# Post-process
converged_points = (flags == 0)
optimal_isp = np.max(Is[converged_points])
```

5.2 Typical Parameter Values

Table 1: Representative Values for N_2O/Paraffin Hybrid

Parameter	**Symbol**	**Value**	**Unit**
Tank pressure	p_{tank}	50–60	bar
Tank temperature	T_{tank}	280–295	K
Chamber pressure	p_c	30–50	bar
Mixture ratio	MR	4–8	-
Regression rate coeff.	a	0.15–0.20	$\mathrm{mm/s/(kg/m^2s)}^n$
Regression rate exp.	n	0.5–0.7	-
Discharge coefficient	C_D	0.7–0.85	-
Fuel density	ρ_{fuel}	850–920	$\mathrm{kg/m^3}$
Expansion ratio	ε	6–12	-

6 Numerical Considerations

6.1 Convergence Stability

6.1.1 Known Issues

1. **No solution exists**: Occurs when $\dot{m}_{inj} < \dot{m}_{throat,min}$ at all feasible p_c
2. **Multiple solutions**: Rare, but possible in narrow pressure ranges
3. **Stiff behavior near bounds**: Damping factor prevents overshooting

6.1.2 Recommended Practices

- Ensure $p_{tank} \gg p_{amb}$ (typically $>$ 3:1 ratio)
- Verify propellant data completeness
- Check CEA convergence separately for expected MR range
- Use adaptive expansion ratio initially, then refine with fixed ε

6.2 Computational Performance

Typical execution times (Intel i7, single core):

- Single-point performance: 10–50 ms
- Pressure convergence: 200–800 ms (4–15 iterations)
- Full parametric sweep (3×4×2 grid): 5–15 seconds

Bottlenecks:

1. CEA thermochemistry calls (∼70% of runtime)
2. CoolProp property evaluations (∼20%)
3. Newton iteration overhead (∼10%)

6.3 Accuracy and Validation

6.3.1 Expected Uncertainties

- Regression rate: ±15–25% (empirical correlation scatter)
- Discharge coefficient: ±5–10% (geometry-dependent)
- CEA thermochemistry: ±1–2% (well-validated)
- Overall I_{sp}: ±3–5% (combined uncertainties)

6.3.2 Validation Sources

Model components validated against:

- Waxman et al. (2013): N_2O injection mass flow
- Marxman & Gilbert (1963): Fuel regression correlations
- Gordon & McBride (1994): NASA CEA code
- Karabeyoglu et al. (2004): Paraffin hybrid data

7 Troubleshooting Guide

7.1 Common Error Messages

7.1.1 "Fluid not found"

Cause: Invalid CoolProp fluid name
Solution: Check available fluids:

```
import CoolProp.CoolProp as cp
print(cp.FluidsList())
```

7.1.2 Flag = 10 (No solution found)

Cause: No pressure satisfies mass balance
Diagnosis:

- Check $p_{tank} > 3 \cdot p_{amb}$
- Verify A_{inj} not too small
- Ensure A_t appropriately sized

7.1.3 Flag = 1 (Pressure diverged)

Cause: Newton iteration exceeded 100 steps
Solutions:

- Increase `MAX_ITERATIONS`
- Adjust initial pressure search range
- Check for discontinuities in $F(p_c)$

7.1.4 Flag = -1 (CEA failed)

Cause: Thermochemical equilibrium not achieved
Solutions:

- Verify fuel/oxidizer specifications
- Check mixture ratio within reasonable bounds (1 ¡ MR ¡ 15)
- Ensure valid enthalpy values

7.2 Performance Optimization Tips

1. **Vectorize loops**: Use NumPy array operations where possible
2. **Cache CEA results**: Store $(p_c, MR) \rightarrow c^*$ lookup table
3. **Parallel execution**: Use `multiprocessing` for parameter sweeps
4. **Reduce CEA calls**: Reuse γ between iterations

8 Extensions and Future Work

8.1 Planned Enhancements

1. **Transient analysis**: Time-dependent burn simulation
2. **Erosive burning**: Port diameter evolution
3. **Multi-port geometry**: Wagon-wheel fuel grains
4. **Heat transfer**: Thermal ablation modeling
5. **3D visualization**: Interactive performance maps

8.2 Research Applications

- Optimization studies (maximize I_{sp}, minimize GLOW)
- Sensitivity analysis (Monte Carlo uncertainty quantification)
- Propellant screening (alternative fuels/oxidizers)
- System trade studies (tankage, feed systems)
- Flight trajectory optimization with realistic performance curves

9 Appendix A: Mathematical Derivations

9.1 Mass Conservation at Throat

Starting from isentropic flow relations, the mass flow through a choked nozzle:

$$\dot{m} = \frac{p_c A_t}{\sqrt{RT_c}} \sqrt{\gamma} \left(\frac{2}{\gamma+1} \right)^{\frac{\gamma+1}{2(\gamma-1)}} \tag{23}$$

Define characteristic velocity:

$$c^* = \frac{p_c A_t}{\dot{m}} \tag{24}$$

Therefore:

$$c^* = \sqrt{\frac{RT_c}{\gamma}} \left(\frac{\gamma+1}{2} \right)^{\frac{\gamma+1}{2(\gamma-1)}} \tag{25}$$

The pressure function becomes:

$$F(p_c) = \dot{m}(p_c) - \frac{p_c A_t}{c^*(p_c)} = 0 \tag{26}$$

9.2 Expansion Ratio for Perfect Expansion

For isentropic expansion from chamber to exit:

$$\frac{p_e}{p_c} = \left(\frac{A_t}{A_e} \right)^{\gamma} \left[\frac{2}{\gamma+1} \left(1 + \frac{\gamma-1}{2} M_e^2 \right) \right]^{\frac{\gamma}{\gamma-1}} \tag{27}$$

Inverting for area ratio when $p_e = p_{amb}$:

$$\varepsilon = \frac{1}{M_e} \left[\frac{2}{\gamma+1} \left(1 + \frac{\gamma-1}{2} M_e^2 \right) \right]^{\frac{\gamma+1}{2(\gamma-1)}} \tag{28}$$

where M_e is found from:

$$\frac{p_{amb}}{p_c} = \left(1 + \frac{\gamma-1}{2} M_e^2 \right)^{-\frac{\gamma}{\gamma-1}} \tag{29}$$

9.3 Thrust Coefficient Derivation

From momentum balance:

$$F = \dot{m} v_e + (p_e - p_{amb}) A_e \tag{30}$$

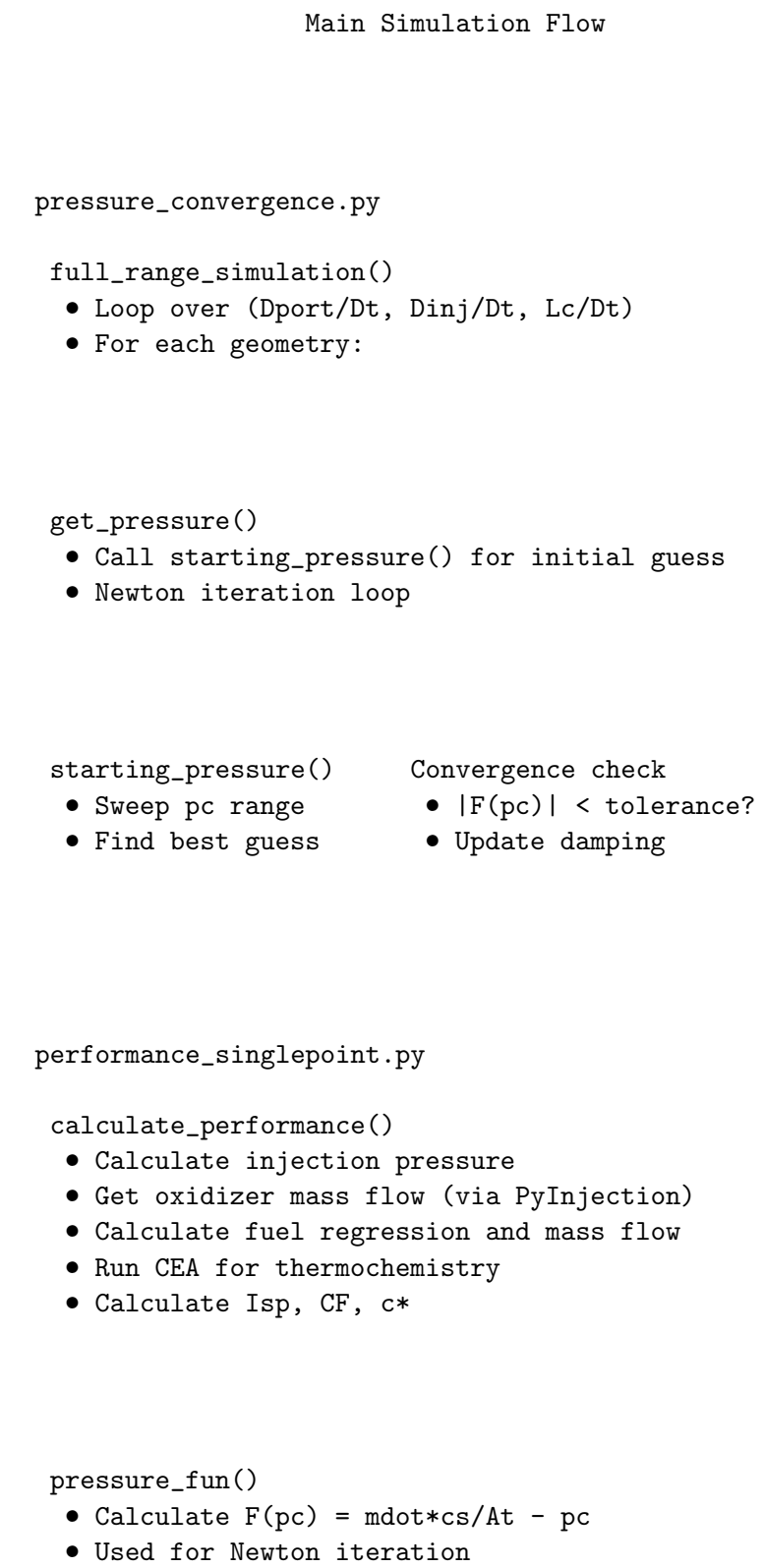
Using isentropic relations and $v_e = M_e \sqrt{\gamma R T_e}$:

$$C_F = \frac{F}{p_c A_t} = \sqrt{\frac{2\gamma^2}{\gamma-1} \left[1 - \left(\frac{p_e}{p_c} \right)^{\frac{\gamma-1}{\gamma}} \right]} + \varepsilon \frac{p_e - p_{amb}}{p_c} \tag{31}$$

For vacuum ($p_{amb} = 0$):

$$C_{F,vac} = \sqrt{\frac{2\gamma^2}{\gamma-1} \left[1 - \left(\frac{p_e}{p_c} \right)^{\frac{\gamma-1}{\gamma}} \right]} + \varepsilon \frac{p_e}{p_c} \tag{32}$$

10 Appendix B: Code Architecture Diagram



PyInjection.py

Injector class
- massflow()
- Phase logic
- SPI/HEM/Gas

CEA_py.py (external)

runCEA()
- Thermochemistry
- Returns Tc, MW, gamma, cs, CF

CoolProp
- Fluid properties
- Phase transitions

NASA CEA Database
- Chemical equilibrium
- Product composition

11 Appendix C: Variable Nomenclature

11.1 Geometric Parameters

Symbol	Description	Units
A_{inj}	Total injection area	m^2
A_{port}	Port cross-sectional area	m^2
A_b	Fuel burning surface area	m^2
A_t	Throat area	m^2
A_e	Exit area	m^2
D_{inj}	Injector orifice diameter	m
D_{port}	Port diameter	m
D_t	Throat diameter	m
L_c	Combustion chamber length	m
ε	Expansion ratio (A_e/A_t)	-

11.2 Flow Parameters

Symbol	Description	Units
$\dot{m}$	Total mass flow rate	kg/s
$\dot{m}_{ox}$	Oxidizer mass flow rate	kg/s
$\dot{m}_{fuel}$	Fuel mass flow rate	kg/s
$\dot{m}''$	Mass flux (per unit area)	kg/(s·m^2)
G_{ox}	Oxidizer mass flux	kg/(s·m^2)
$\dot{r}$	Fuel regression rate	m/s
MR	Mixture ratio (O/F)	-

11.3 Thermodynamic Properties

Symbol	Description	Units
p_c	Chamber total pressure	Pa
p_{tank}	Tank pressure	Pa
p_{inj}	Injection pressure	Pa
p_{amb}	Ambient pressure	Pa
p_v	Vapor pressure	Pa
T_c	Chamber temperature	K
T_{tank}	Tank temperature	K
ρ	Density	kg/m^3
γ	Specific heat ratio (c_p/c_v)	-
MW	Molecular weight	kg/kmol
R	Specific gas constant	J/(kg·K)
h	Specific enthalpy	J/kg

Symbol	Description	Units
c^*	Characteristic velocity	m/s
C_F	Thrust coefficient	-
$C_{F,vac}$	Vacuum thrust coefficient	-
I_{sp}	Specific impulse	s
$I_{sp,vac}$	Vacuum specific impulse	s
F	Thrust	N
C_D	Discharge coefficient	-

11.4 Performance Metrics

11.5 Empirical Coefficients

Symbol	Description	Units
a	Regression rate coefficient	m/s / $(\text{kg}/(\text{m}^2 \cdot \text{s}))^n$
n	Regression rate exponent	-
ρ_{fuel}	Fuel density	kg/m^3

12 Appendix D: Sample Input/Output Files

12.1 Configuration File Example (JSON)

```
{
  "geometry": {
    "Dport_Dt_range": [3.5, 4.0, 4.5],
    "Dinj_Dt_range": [0.8, 0.85, 0.9],
    "Lc_Dt_range": [8, 9, 10]
  },
  "operating_conditions": {
    "ptank_bar": 55,
    "Ttank_K": 288,
    "pamb_bar": 1,
    "expansion_ratio": "adapt"
  },
  "propellants": {
    "oxidizer": {
      "name_coolprop": "NitrousOxide",
      "name_cea": "N2O",
      "temperature_K": 288,
      "weight_fraction": 100
    },
    "fuel": {
      "name": "paraffin",
      "formula": "C␣73␣H␣124",
      "temperature_K": 533,
      "enthalpy_kj_mol": -1860.6,
      "density_kg_m3": 850,
      "regression_a": 0.00017,
      "regression_n": 0.5
    }
  },
  "coefficients": {
    "discharge_coefficient": 0.8
  }
}
```

12.2 Output Data Structure (Python Dictionary)

```
results = {
    'geometry': {
        'Dport_Dt': numpy.ndarray,  # Shape (n1, n2, n3)
        'Dinj_Dt': numpy.ndarray,
        'Lc_Dt': numpy.ndarray
    },
    'pressures': {
        'pc': numpy.ndarray,        # Chamber pressure [Pa]
        'p_inj': numpy.ndarray,     # Injection pressure [Pa]
        'Fpc': numpy.ndarray        # Residual [Pa]
    },
    'mass_flows': {
        'mdot_ox': numpy.ndarray,   # [kg/s]
        'mdot_fuel': numpy.ndarray,
        'mdot_total': numpy.ndarray,
        'Gox': numpy.ndarray,       # [kg/(s*m^2)]
        'r': numpy.ndarray          # Regression [m/s]
```

```
    },
    'thermochemistry': {
        'Tc': numpy.ndarray,          # [K]
        'MW': numpy.ndarray,          # [kg/kmol]
        'gamma': numpy.ndarray,
        'MR': numpy.ndarray
    },
    'performance': {
        'cs': numpy.ndarray,          # [m/s]
        'CF': numpy.ndarray,
        'CF_vac': numpy.ndarray,
        'Isp': numpy.ndarray,         # [s]
        'Isp_vac': numpy.ndarray,
        'eps': numpy.ndarray
    },
    'convergence': {
        'flags': numpy.ndarray        # Integer status codes
    }
}
```

13 Appendix E: Unit Testing Guidelines

13.1 Test Cases for Injection Module

```
def test_injector_liquid_flow():
    """Test all-liquid N2O flow"""
    ox = Injector('NitrousOxide')
    ox.injection_area(D=0.01, n=4)

    # Conditions: well below vapor pressure
    ox.massflow(p1=60e5, p2=50e5, T=280, cD=0.8)

    # Expected: SPI model dominates
    assert ox.mdot > 0
    assert ox.mdot < 2000  # Reasonable upper bound

def test_injector_two_phase():
    """Test two-phase N2O flow"""
    ox = Injector('NitrousOxide')
    ox.injection_area(D=0.01, n=4)

    # Conditions: chamber below vapor pressure
    ox.massflow(p1=55e5, p2=30e5, T=288, cD=0.8)

    # Expected: mixed SPI/HEM model
    assert ox.mdot > 0

def test_injector_backflow():
    """Test backflow prevention"""
    ox = Injector('NitrousOxide')
    ox.injection_area(D=0.01, n=4)

    # Conditions: pc > ptank
    ox.massflow(p1=40e5, p2=50e5, T=288, cD=0.8)

    # Expected: zero mass flow
    assert ox.mdot == 0
```

13.2 Test Cases for Performance Module

```
def test_performance_physical_bounds():
    """Verify physical reasonableness of outputs"""
    # Setup typical conditions
    Ainj = 0.05  # m^2
    Aport = 5.0
    Ab = 15.0
    eps = 8.0
    ptank = 55e5
    Ttank = 288
    pc = 43e5
    CD = 0.8
    a = 0.17e-3
    n = 0.5
    rho_fuel = 850

    # Run calculation
    results = calculate_performance(
```

```
        Ainj, Aport, Ab, eps, ptank, Ttank, pc,
        CD, a, n, rho_fuel, oxidizer, fuel
    )

    # Unpack
    (p_inj, mdot_ox, mdot_fuel, mdot, Gox, r, MR,
     Tc, MW, gamma, eps_out, cs, CF_vac, CF,
     Ivac, Is, flag) = results

    # Physical bounds checks
    assert 0 < MR < 20, "Mixture ratio out of range"
    assert 2000 < Tc < 4000, "Temperature unrealistic"
    assert 1.1 < gamma < 1.4, "Gamma out of range"
    assert 1400 < cs < 1800, "c* unrealistic"
    assert 100 < Is < 300, "Isp unrealistic"
    assert flag == 0, "Performance calculation failed"
```

13.3 Test Cases for Convergence Module

```
def test_pressure_convergence_typical():
    """Test convergence for typical operating point"""
    Dinj = 0.8
    Dport = 4.0
    Lc = 9.0
    Dt = 1.0

    Ainj = 0.25 * np.pi * Dinj**2
    Aport = 0.25 * np.pi * Dport**2
    At = 0.25 * np.pi * Dt**2
    Ab = np.pi * Dport * Lc

    pc, Fpc, n_iter, maxit, gamma = get_pressure(
        Ainj, Aport, At, Ab, "adapt",
        55e5, 288, 0.8, 0.17e-3, 0.5, 850,
        oxidizer, fuel, 1e5
    )

    # Convergence checks
    assert pc > 0, "No solution found"
    assert abs(Fpc) < 1.0, "Did not converge"
    assert n_iter < maxit, "Exceeded max iterations"
    assert 30e5 < pc < 50e5, "Pressure out of range"

def test_no_solution_case():
    """Test behavior when no solution exists"""
    # Impossible geometry: throat too large
    Ainj = 0.01
    Aport = 1.0
    At = 10.0  # Throat larger than port!
    Ab = 10.0

    pc, Fpc, n_iter, maxit, gamma = get_pressure(
        Ainj, Aport, At, Ab, "adapt",
        55e5, 288, 0.8, 0.17e-3, 0.5, 850,
        oxidizer, fuel, 1e5
    )
```

```
    # Should detect no solution
    assert pc == 0, "Should return zero for no solution"
    assert n_iter == maxit + 1, "Should flag no solution"
```

14 References

[1] Waxman, B. S., Zimmerman, J. E., Cantwell, B. J., and Zilliac, G. G., "Mass Flow Rate and Isolation Characteristics of Injectors for Use with Self-Pressurizing Oxidizers in Hybrid Rockets," *49th AIAA/ASME/SAE/ASEE Joint Propulsion Conference*, AIAA Paper 2013-3636, 2013.

[2] Marxman, G. A., and Gilbert, M., "Turbulent Boundary Layer Combustion in the Hybrid Rocket," *Symposium (International) on Combustion*, Vol. 9, No. 1, 1963, pp. 371–383.

[3] Gordon, S., and McBride, B. J., "Computer Program for Calculation of Complex Chemical Equilibrium Compositions and Applications," NASA Reference Publication 1311, 1994.

[4] Karabeyoglu, M. A., Altman, D., and Cantwell, B. J., "Combustion of Liquefying Hybrid Propellants: Part 1, General Theory," *Journal of Propulsion and Power*, Vol. 18, No. 3, 2002, pp. 610–620.

[5] Sutton, G. P., and Biblarz, O., *Rocket Propulsion Elements*, 9th ed., John Wiley & Sons, Hoboken, NJ, 2017.

[6] Bell, I. H., Wronski, J., Quoilin, S., and Lemort, V., "Pure and Pseudo-pure Fluid Thermophysical Property Evaluation and the Open-Source Thermophysical Property Library CoolProp," *Industrial & Engineering Chemistry Research*, Vol. 53, No. 6, 2014, pp. 2498–2508.

[7] Humble, R. W., Henry, G. N., and Larson, W. J., *Space Propulsion Analysis and Design*, McGraw-Hill, New York, 1995.

[8] Chiaverini, M. J., and Kuo, K. K., *Fundamentals of Hybrid Rocket Combustion and Propulsion*, AIAA Progress in Astronautics and Aeronautics Series, Vol. 218, 2007.

15 Glossary

CEA Chemical Equilibrium with Applications - NASA thermochemical code

CoolProp Open-source thermophysical property library

HEM Homogeneous Equilibrium Model - two-phase flow model assuming thermal and mechanical equilibrium

Hybrid Rocket
Propulsion system using propellants in different phases (typically liquid oxidizer, solid fuel)

Isentropic Thermodynamic process with constant entropy (reversible adiabatic)

Mixture Ratio (MR)
Mass ratio of oxidizer to fuel (O/F)

Newton-Raphson
Iterative root-finding algorithm using function derivatives

Regression Rate
Rate at which solid fuel surface recedes during combustion

SPI Self-Pressurizing Injector - model for subcooled liquid flow

Characteristic Velocity (c^*)
Performance parameter: $c^* = p_c A_t / \dot{m}$

Thrust Coefficient (C_F)
Dimensionless thrust: $C_F = F/(p_c A_t)$

Specific Impulse (I_{sp})
Thrust per unit weight flow rate: $I_{sp} = F/(\dot{m} g_0)$

Expansion Ratio (ε)
Nozzle area ratio: $\varepsilon = A_e / A_t$

16 Contact and Support

16.1 Author Information

- **Author:** Edgardo Rossi, Cristian Casalanguida, Ceren Su Trakyalı
- **Year:** 2025
- **Institution:** [Your Institution]

16.2 Version History

Version	Date	Changes
1.0	2025-01-XX	Initial release - Injection module - Performance module - Convergence solver

16.3 License

This software is provided for educational and research purposes. Please cite appropriately when using this code in publications or presentations.

16.4 Acknowledgments

This work builds upon established models from:

- Stanford University Hybrid Rocket Program
- NASA Glenn Research Center (CEA code)
- CoolProp development team

End of Documentation
Hybrid Rocket Engine Performance Analysis System v1.0