Hybrid Rocket Optimization GUI

Integration Documentation

Technical Documentation

October 28, 2025

Contents

1 Introduction

This document explains the integration between the graphical user interface (GUI) and the hybrid rocket optimization backend. The integration enables users to configure parameters, run optimization simulations, and visualize results through an intuitive interface.

1.1 Objectives

The main objectives of the integration are:

- Provide a user-friendly interface for parameter configuration
- Execute optimization simulations without manual scripting
- Display results in real-time during computation
- Export optimization results for further analysis

2 Architecture Overview

2.1 System Components

The integrated system consists of three main components:

1. **GUI Frontend** (`GUI.py`): Tkinter-based interface
2. **Optimization Backend** (`optimization.py`): Numerical simulation engine
3. **Performance Calculator** (`performance_singlepoint.py`): Single-point performance evaluation

2.2 Data Flow

Data Flow Diagram

User Input → **GUI Configuration** → **Parameter Extraction** → **Optimization Engine** → **Results Display** → **Export**

3 Key Implementation Details

3.1 Configuration Management

3.1.1 Input Parameter Structure

The GUI collects parameters organized in four sections:

1. **Fuel & Oxidizer**: Material properties and regression rate parameters
2. **Injector**: Discharge coefficient and dimensional ranges
3. **Chamber**: Port and length dimensional ranges
4. **Nozzle & Conditions**: Operating conditions

3.1.2 Parameter Storage

All input fields are stored in a dictionary:

```python
def create_section(self, parent, title, fields):
    for label_text, key, default_value in fields:
        # Create entry widget
        entry = tk.Entry(row, font=('Arial', 11),
                        width=30, relief=tk.SUNKEN, bd=2)
        entry.insert(0, default_value)

        # Store reference with unique key
        self.inputs[key] = entry
```

Listing 1: Input Storage Implementation

3.2 Configuration Extraction

The `get_config_dict()` method extracts all user inputs and converts them to appropriate data types:

```python
def get_config_dict(self):
    """Extract configuration from inputs"""
    try:
        config = {}
        for key, entry in self.inputs.items():
            value = entry.get().strip()
            if not value:
                return None

            # String fields (no conversion)
            if key in ['oxidizer_cp', 'oxidizer_cea',
                       'fuel_name', 'fuel_formula', 'eps']:
                config[key] = value
            else:
                # Numeric fields (convert to float)
                config[key] = float(value)

        return config
    except ValueError:
        return None
```

Listing 2: Configuration Dictionary Creation

4 Optimization Integration

4.1 Threading Implementation

To prevent the GUI from freezing during long computations, the optimization runs in a separate thread:

```python
def run_optimization(self):
    # Validate configuration
    config = self.get_config_dict()
    if not config:
        messagebox.showerror("Error", "Invalid configuration")
        return

```

```python
    # Update UI state
    self.run_btn.configure(state='disabled')
    self.progress_bar.start(10)
    self.is_optimizing = True

    # Launch worker thread
    thread = threading.Thread(
        target=self._optimization_worker,
        args=(config,)
    )
    thread.daemon = True
    thread.start()
```

Listing 3: Thread-Based Optimization

4.1.1 Worker Thread Function

The worker thread performs the actual optimization:

```python
def _optimization_worker(self, config):
    try:
        # Prepare parameter ranges
        Dport_Dt_range = np.arange(
            config['dport_dt_min'],
            config['dport_dt_max'],
            config['dport_dt_step']
        )
        Dinj_Dt_range = np.arange(
            config['dinj_dt_min'],
            config['dinj_dt_max'],
            config['dinj_dt_step']
        )
        Lc_Dt_range = np.arange(
            config['lc_dt_min'],
            config['lc_dt_max'],
            config['lc_dt_step']
        )

        # Prepare fuel dictionary
        fuel = {
            "Fuels": [config['fuel_name']],
            "Weight fraction": ["100"],
            "Exploded Formula": [config['fuel_formula']],
            "Temperature [K]": [config['fuel_temp']],
            "Specific Enthalpy [kj/mol]": [config['fuel_enthalpy']]
        }

        # Prepare oxidizer dictionary
        oxidizer = {
            "OxidizerCP": config['oxidizer_cp'],
            "OxidizerCEA": config['oxidizer_cea'],
            "Weight fraction": "100",
            "Exploded Formula": "",
            "Temperature [K]": "",
            "Specific Enthalpy [kj/mol]": ""
        }

        # Call optimization function
```

```python
        results = optimization.full_range_simulation(
            Dport_Dt_range, Dinj_Dt_range, Lc_Dt_range,
            config['eps'], config['ptank'], config['Ttank'],
            config['CD'], config['a'], config['n'],
            config['rho_fuel'], oxidizer, fuel,
            config['pamb'], config['gamma0']
        )

        # Store results
        self.optimization_results = {
            'arrays': results,
            'ranges': {...},
            'config': config
        }

    except Exception as e:
        self.log_to_console(f"ERROR: {str(e)}")
```

Listing 4: Optimization Worker Thread

4.2 Result Storage Structure

Results are stored in a nested dictionary structure:

Result Dictionary Structure

```
optimization_results = {
    'arrays': (pc, Fpc, p_inj, mdot_ox, ..., flag),
    'ranges': {
        'Dport_Dt': array([...]),
        'Dinj_Dt': array([...]),
        'Lc_Dt': array([...])
    },
    'config': {...}
}
```

The `arrays` tuple contains 19 3D arrays corresponding to:

4.3 Convergence Flags

The convergence flag indicates the status of each configuration:

5 User Interface Features

5.1 Progress Monitoring

5.1.1 Console Output

Real-time logging to a scrolled text widget:

```python
def log_to_console(self, message):
    """Add message to console output"""
    if hasattr(self, 'console_output'):
        self.console_output.insert(tk.END, message + "\n")
        self.console_output.see(tk.END)  # Auto-scroll
        self.console_output.update()     # Force update
```

Index	Parameter	Units
0	Chamber pressure (p_c)	Pa
1	Pressure function (F_{pc})	Pa
2	Injection pressure (p_{inj})	Pa
3	Oxidizer mass flow ($\dot{m}_{ox}$)	kg/(s·m^2)
4	Fuel mass flow ($\dot{m}_{fuel}$)	kg/(s·m^2)
5	Total mass flow ($\dot{m}$)	kg/(s·m^2)
6	Oxidizer mass flux (G_{ox})	kg/(s·m^2)
7	Regression rate (r)	m/s
8	Mixture ratio (MR)	-
9	Expansion ratio (ϵ)	-
10	Chamber temperature (T_c)	K
11	Molecular weight (MW)	kg/kmol
12	Specific heat ratio (γ)	-
13	Characteristic velocity (c^*)	m/s
14	Vacuum thrust coefficient ($C_{F,vac}$)	-
15	Thrust coefficient (C_F)	-
16	Vacuum specific impulse ($I_{sp,vac}$)	s
17	Specific impulse (I_{sp})	s
18	Convergence flag	-

Table 1: Output Array Contents

Flag Value	Meaning
0	Converged successfully
1	Pressure iteration diverged
-1	CEA (chemical equilibrium) diverged
2	Both pressure and CEA diverged
10	No pressure solution exists

Table 2: Convergence Flag Values

Listing 5: Console Logging Function

5.1.2 Progress Bar

An indeterminate progress bar shows activity:

```
# Start animation
self.progress_bar.start(10)  # Update every 10ms

# Stop animation
self.progress_bar.stop()
```

Listing 6: Progress Bar Control

5.2 Output Visualization

5.2.1 Results Summary

The output page displays convergence statistics:

```
flag_array = results[-1]
converged = np.sum(flag_array == 0)
total = flag_array.size

summary_text = f"""
Total configurations: {total}
Converged: {converged} ({100*converged/total:.1f}%)
Pressure diverged: {np.sum(flag_array == 1)}
CEA diverged: {np.sum(flag_array == -1)}
Both diverged: {np.sum(flag_array == 2)}
No solution: {np.sum(flag_array == 10)}
"""
```

Listing 7: Summary Statistics

5.2.2 Best Configuration

The configuration with maximum specific impulse is identified:

```
# Extract specific impulse array
Is_array = results[16]

# Mask non-converged solutions
mask = flag_array == 0
Is_converged = np.where(mask, Is_array, -np.inf)

# Find maximum
best_idx = np.unravel_index(
    np.argmax(Is_converged),
    Is_converged.shape
)

# Extract optimal parameters
best_dport = ranges['Dport_Dt'][best_idx[0]]
best_dinj = ranges['Dinj_Dt'][best_idx[1]]
best_lc = ranges['Lc_Dt'][best_idx[2]]
```

Listing 8: Finding Optimal Configuration

6 File Operations

6.1 Configuration Save/Load

6.1.1 JSON Export

Configurations are saved in JSON format:

```
def _save_to_file(self, filename):
    config = self.get_config_dict()
    if config:
        with open(filename, 'w') as f:
            json.dump(config, f, indent=4)
        messagebox.showinfo("Saved",
            f"Configuration saved to:\n{filename}")
```

Listing 9: Configuration Export

6.1.2 JSON Import

Saved configurations can be loaded:

```python
def open_config(self):
    filename = filedialog.askopenfilename(
        filetypes=[("JSON files", "*.json"),
                   ("All files", "*.*")]
    )
    if filename:
        with open(filename, 'r') as f:
            config = json.load(f)

        # Populate input fields
        for key, value in config.items():
            if key in self.inputs:
                self.inputs[key].delete(0, tk.END)
                self.inputs[key].insert(0, str(value))
```

Listing 10: Configuration Import

6.2 Results Export

Results are exported as compressed NumPy arrays:

```python
def export_results(self):
    filename = filedialog.asksaveasfilename(
        defaultextension=".npz",
        filetypes=[("NumPy files", "*.npz")]
    )

    if filename:
        results = self.optimization_results['arrays']
        ranges = self.optimization_results['ranges']

        np.savez(filename,
            pc=results[0], Fpc=results[1],
            p_inj=results[2], mdot_ox=results[3],
            # ... (all 19 arrays)
            Dport_Dt_range=ranges['Dport_Dt'],
            Dinj_Dt_range=ranges['Dinj_Dt'],
            Lc_Dt_range=ranges['Lc_Dt'])
```

Listing 11: NumPy Results Export

7 Key Tkinter Commands Used

7.1 Widget Creation

7.2 Layout Management

Three geometry managers are used:

1. `pack()`: Stack widgets vertically or horizontally

```python
widget.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
```

2. `grid()`: Organize widgets in rows and columns (not used here)
3. `place()`: Absolute positioning (not used here)

Command	Description
`tk.Frame()`	Container widget for organizing layouts
`tk.Label()`	Display text or images
`tk.Entry()`	Single-line text input field
`tk.Button()`	Clickable button with command callback
`tk.Canvas()`	Drawing surface, used with scrollbar
`ttk.Progressbar()`	Progress indicator (determinate/indeterminate)
`scrolledtext.ScrolledText()`	Multi-line text widget with scrollbar

Table 3: Primary Tkinter Widgets

7.3 Event Binding

```
# Bind keyboard event
entry.bind('<KeyRelease>', lambda e: self.validate_inputs())

# Button command callback
button = tk.Button(frame, command=self.run_optimization)

# Canvas scrolling
canvas.configure(yscrollcommand=scrollbar.set)
scrollbar.config(command=canvas.yview)
```

Listing 12: Event Handling

7.4 Dialog Windows

```
# File save dialog
filename = filedialog.asksaveasfilename(
    defaultextension=".json",
    filetypes=[("JSON files", "*.json")]
)

# File open dialog
filename = filedialog.askopenfilename(
    filetypes=[("JSON files", "*.json")]
)

# Information message
messagebox.showinfo("Title", "Message text")

# Error message
messagebox.showerror("Error", "Error description")

# Warning message
messagebox.showwarning("Warning", "Warning text")
```

Listing 13: File Dialogs and Message Boxes

8 Threading and Thread Safety

8.1 Thread Creation

```
thread = threading.Thread(
```

```python
    target=self._optimization_worker,
    args=(config,)
)
thread.daemon = True  # Terminates with main program
thread.start()
```

Listing 14: Creating Daemon Thread

8.2 Thread-Safe GUI Updates

GUI updates from worker thread must use `root.after()`:

```python
# From worker thread
self.root.after(0, self._finish_optimization)

# _finish_optimization runs in main thread
def _finish_optimization(self):
    self.progress_bar.stop()
    self.run_btn.configure(state='normal')
```

Listing 15: Thread-Safe Updates

9 Validation System

9.1 Input Validation

Real-time validation on every keystroke:

```python
def validate_inputs(self):
    all_valid = True

    for key, entry in self.inputs.items():
        value = entry.get().strip()

        # Check if empty
        if not value:
            all_valid = False
            continue

        # Skip string fields
        if key in ['oxidizer_cp', 'oxidizer_cea',
                   'fuel_name', 'fuel_formula', 'eps']:
            continue

        # Validate numeric fields
        try:
            float(value)
        except ValueError:
            all_valid = False

    # Update button color
    if all_valid:
        self.save_btn.configure(bg='#006400')  # Green
    else:
        self.save_btn.configure(bg='#8b0000')  # Red
```

Listing 16: Input Validation Logic

10 Usage Workflow

10.1 Step-by-Step Process

1. **Launch Application**

```
python GUI.py
```

2. **Configure Parameters**

   - Navigate to Configuration page (default)
   - Fill in all required fields or use defaults
   - Button turns green when all inputs are valid
   - Click "Validate Configuration" to confirm

3. **Run Optimization**

   - Navigate to Optimization page
   - Click "Run Optimization" button
   - Monitor progress in console output
   - Wait for completion message

4. **View Results**

   - Navigate to Output page
   - Review convergence statistics
   - Examine best configuration
   - Click "Export Results" to save data

5. **Save Configuration**

   - Click "Main Menu "
   - Select "Save as" to save configuration
   - Use "Open" to load previous configurations

11 Error Handling

11.1 Exception Management

The worker thread includes comprehensive error handling:

```python
def _optimization_worker(self, config):
    try:
        # Optimization code
        results = optimization.full_range_simulation(...)
        self.optimization_results = {...}

    except Exception as e:
        # Log error to console
        self.log_to_console(f"ERROR: {str(e)}")

        # Print full traceback
        import traceback
        self.log_to_console(traceback.format_exc())
```

```python

    finally:
        # Always clean up
        self.root.after(0, self._finish_optimization)
```

Listing 17: Exception Handling

12 Performance Considerations

12.1 Computational Complexity

The total number of configurations evaluated is:

$$N_{total} = N_{D_{port}} \times N_{D_{inj}} \times N_{L_c} \tag{1}$$

For the default ranges:

$$\begin{aligned} N_{D_{port}} &= \frac{5.0 - 3.5}{0.5} = 3 \\ N_{D_{inj}} &= \frac{1.0 - 0.8}{0.05} = 4 \\ N_{L_c} &= \frac{10 - 8}{1} = 2 \\ N_{total} &= 3 \times 4 \times 2 = 24 \text{ configurations} \end{aligned}$$

Each configuration requires:

- CEA thermochemical calculations
- Iterative pressure convergence (Newton-like method)
- Performance parameter calculations

12.2 Optimization Suggestions

To improve performance for large parameter sweeps:

1. **Parallel Processing**: Modify optimization.py to use multiprocessing

```python
from multiprocessing import Pool

with Pool(processes=4) as pool:
    results = pool.starmap(compute_single_config, configs)
```

2. **Progress Updates**: Add callback for GUI updates

```python
# In optimization loop
if ind % 10 == 0:  # Every 10 iterations
    progress = ind / total * 100
    callback(progress)
```

3. **Checkpointing**: Save intermediate results

```python
# Save every 100 configurations
if ind % 100 == 0:
    np.savez('checkpoint.npz', partial_results)
```

13 Future Enhancements

13.1 Planned Features

1. **Real-time Plotting**
   - Integrate matplotlib for live visualization
   - Plot convergence history during optimization
   - 2D contour plots of performance surfaces
2. **Multi-objective Optimization**
   - Pareto front visualization
   - Weighted objective function
   - Interactive trade-off exploration
3. **Database Integration**
   - Store all optimization runs
   - Compare different configurations
   - Historical performance tracking
4. **Advanced Analysis**
   - Sensitivity analysis
   - Uncertainty quantification
   - Design space exploration

14 Troubleshooting

14.1 Common Issues

Problem	**Solution**
Import error for optimization module	Ensure optimization.py is in correct path or add to PYTHONPATH
GUI freezes during optimization	Check that threading is properly implemented
Results not displaying	Verify optimization completed and results stored in self.optimization_results
Console output not updating	Ensure log_to_console() includes update() call
Export fails	Check write permissions and valid file path

Table 4: Common Issues and Solutions

15 Conclusion

This integrated GUI provides a complete workflow for hybrid rocket optimization:

- **User-friendly**: No Python knowledge required
- **Comprehensive**: All parameters configurable

- **Responsive**: Threading prevents UI freezing
- **Informative**: Real-time progress and detailed results
- **Portable**: Results exportable for external analysis

The modular design allows easy extension with additional features and analysis tools.