Energy converter modelling

Technical Documentation Manual

Version 1.0

Ben Noble

June 2, 2025

Contents

1 Ship Equipment exploration **2**

1.1 Energy converter . . . 2
1.2 System Architecture . . . 2
1.2.1 Abstract Base Class Design . . . 2
1.2.2 Core Design Principles . . . 3
1.2.3 Class Hierarchy . . . 4
1.3 Internal Combustion Engine Implementation . . . 4
1.3.1 Performance Modelling Hierarchy . . . 4
1.3.2 Configuration Types Demonstrated . . . 4
1.3.3 Demonstrated Functionality . . . 4
1.4 Fuel Cell Implementation . . . 5
1.4.1 Electrochemical Modelling . . . 5
1.4.2 Technology Types Demonstrated . . . 5
1.5 Abstract Base Class Benefits in Demonstrations . . . 6
1.5.1 Enabling Polymorphic Analysis . . . 6
1.5.2 Simplified Extension for New Technologies . . . 6
1.5.3 Type Safety and Error Prevention . . . 7
1.5.4 Data-Driven Configuration . . . 7
1.6 Plotting and Visualization . . . 7
1.6.1 Performance Plotting . . . 7
1.6.2 Plot Types and Applications . . . 7
1.7 Real-World Applications . . . 8
1.7.1 Marine Vessel Integration . . . 8
1.7.2 Operational Scenarios . . . 8
1.8 Validation and Testing . . . 8
1.8.1 Data Sources . . . 8
1.8.2 Validation Results . . . 8
1.9 Error Handling and Robustness . . . 9
1.9.1 Input Validation . . . 9
1.9.2 Graceful Degradation . . . 9
1.10 Performance and Scalability . . . 9
1.10.1 Computational Efficiency . . . 9
1.10.2 Memory Management . . . 9
1.11 Extension and Customization . . . 10
1.11.1 Adding New Technologies . . . 10
1.11.2 Custom Configurations . . . 10
1.12 Future Developments . . . 10
1.12.1 Planned Enhancements . . . 10
1.12.2 Integration Opportunities . . . 10
1.13 Conclusion . . . 11

1. Ship Equipment exploration

1.1 Energy converter

The energy converter framework provides a unified approach to modelling different energy conversion technologies for marine applications. The system is designed with modularity and flexibility in mind. Whilst also maitaining the concept of having an object with a direct link to a performance database, keeping it grounded in the real world. Two primary converter types are implemented:

- **Internal Combustion Engines (ICE)**: Traditional combustion-based power generation
- **Fuel Cells (FC)**: Electrochemical energy conversion systems

Each converter type features its own configuration management system, performance calculation methods, and visualization tools, while maintaining a consistent API for interoperability.

1.2 System Architecture

1.2.1 Abstract Base Class Design

The framework is built around a central `EnergyConverter` abstract base class that defines the contract all energy conversion technologies must implement. This ensures consistency and interoperability across different converter types.

EnergyConverter Abstract Base Class

The `EnergyConverter` class defines four mandatory abstract methods:

```python
@abstractmethod
def get_operating_limits(self) -> Dict[str, float]:
    """Returns min_load and max_load in kW"""

@abstractmethod
def load_point(self, load: float) -> float:
    """Returns load as fraction of rated power (0-1)"""

@abstractmethod
def efficiency_at_load(self, load: float) -> float:
    """Returns efficiency as decimal (0-1)"""

@abstractmethod
def fuel_consumption_at_load(self, load: float) -> float:
    """Returns fuel consumption in kg/h"""
```

Listing 1.1: EnergyConverter Abstract Methods

Benefits of Abstract Base Class Approach

This design provides several key advantages:

1. **Interface Consistency**: All converters provide the same methods with identical signatures
2. **Polymorphism**: Different converter types can be used interchangeably in analysis code
3. **Type Safety**: Abstract methods must be implemented, preventing incomplete implementations
4. **Documentation**: Clear contract definition for what each converter must provide
5. **Testing**: Common test suites can be applied to all converter implementations

Implementation Pattern

Each concrete converter class inherits from `EnergyConverter` and implements the abstract methods according to its specific technology:

```
class InternalCombustionEngine(EnergyConverter, LoggerMixin):
    def efficiency_at_load(self, load: float) -> float:
        # ICE-specific efficiency calculation using SFOC, curves, etc.

class FuelCell(EnergyConverter, LoggerMixin):
    def efficiency_at_load(self, load: float) -> float:
        # FC-specific efficiency calculation using electrochemical model
```

Listing 1.2: Implementation Pattern Example

This allows the same analysis code to work with different technologies:

```
def analyze_converter(converter: EnergyConverter, load_profile):
    """Works with any converter type"""
    efficiencies = [converter.efficiency_at_load(load) for load in load_profile]
    consumptions = [converter.fuel_consumption_at_load(load, fuel)
                    for load in load_profile]
    return efficiencies, consumptions

# Same function works for both ICE and FC
ice_results = analyze_converter(ice_engine, loads)
fc_results = analyze_converter(fuel_cell, loads)
```

Listing 1.3: Polymorphic Usage

1.2.2 Core Design Principles

Building on the abstract base class foundation, the framework follows several key design principles:

1. **Unified Interface**: All converters implement the `EnergyConverter` contract
2. **Configuration-Driven**: Performance characteristics are defined through configuration objects
3. **Data-Driven Modelling**: Real test data is preferred over simplified models (performance data)
4. **Extensible Registry System**: New configurations can be easily added and managed
5. **Comprehensive Validation**: Input validation and error handling throughout
6. **Technology Abstraction**: Implementation details are hidden behind consistent interfaces

1.2.3 Class Hierarchy

ICE Framework:

- `InternalCombustionEngine`
- `ICEConfig`
- `ICEConfigRegistry`

FC Framework:

- `FuelCell`
- `FuelCellConfig`
- `FuelCellConfigRegistry`

Figure 1.1: Parallel architecture for different converter technologies

1.3 Internal Combustion Engine Implementation

1.3.1 Performance Modelling Hierarchy

The ICE implementation uses a sophisticated hierarchy for performance calculations:

1. **Efficiency Curves**: Direct interpolation from test data
2. **Efficiency Coefficients**: Polynomial approximation ($ax^2 + bx + c$)
3. **SFOC-based Calculation**: Using Specific Fuel Oil Consumption data

SFOC Calculation Methods

The SFOC (Specific Fuel Oil Consumption) calculation supports multiple approaches:

$$\text{SFOC}_{load} = \text{SFOC}_{base} \times (a \cdot \text{load\_point}^2 + b \cdot \text{load\_point} + c) \tag{1.1}$$

$$\text{Efficiency} = \frac{1000}{\text{SFOC} \times \text{LHV}_{fuel}} \tag{1.2}$$

where load_point is the fraction of rated power (0-1).

1.3.2 Configuration Types Demonstrated

Config Name	Data Source	Method	Characteristics
MAHARAJ_ICE	Real test data	Efficiency curves	51-55% efficiency, HFO fuel
Jalkanen_Diesel	Literature	SFOC coefficients	Polynomial model, diesel fuel
Basic_Engine	Simplified	Constant values	40% flat efficiency

Table 1.1: ICE Configuration Types

1.3.3 Demonstrated Functionality

Performance Calculations

The ICE demonstration script showcases performance calculations across multiple load points:

```
# Calculate performance at 75% load
load_kw = 0.75 * engine.rated_power
efficiency = engine.efficiency_at_load(load_kw)
sfoc = engine.SFOC_at_load(load_kw)
fuel_consumption = engine.fuel_consumption_at_load(load_kw)
```

Listing 1.4: Example Performance Calculation

Plotting Capabilities

Two types of plots are demonstrated:

1. **Individual Performance Plots**: Efficiency, SFOC, and fuel consumption vs. load
2. **Comparison Plots**: Side-by-side comparison of different engine configurations

1.4 Fuel Cell Implementation

1.4.1 Electrochemical Modelling

Fuel cells use a simplified but accurate modelling approach focused on electrical efficiency:

$$\text{Fuel Consumption} = \frac{\text{Electrical Load}}{\text{Efficiency} \times \text{LHV}_{fuel}} \text{ [kg/h]} \tag{1.3}$$

This direct approach is more appropriate for electrochemical conversion than SFOC-based methods.

1.4.2 Technology Types Demonstrated

Technology	Efficiency	Temperature	Characteristics
PEMFC	50%	80°C	Low temperature, fast start-up
SOFC	53%	230°C	High temperature, waste heat recovery
Various H Sources	Same	-	Different upstream emissions

Table 1.2: Fuel Cell Technology Types

Load Profile Adaptations

Fuel cell specific load profiles include:

- **Ramping**: Transient response testing
- **Grid Following**: Daily demand cycles
- **Frequency Regulation**: Fast response scenarios

1.5 Abstract Base Class Benefits in Demonstrations

1.5.1 Enabling Polymorphic Analysis

The abstract base class design is crucial to the functionality demonstrated in both executable scripts. It enables several key capabilities:

Technology-Agnostic Comparison Functions

The demonstration scripts can compare different technologies using the same analysis functions:

```
def demonstrate_performance_calculations(converters: Dict[str, EnergyConverter]):
    """Works for both ICE and FC converters"""
    for name, converter in converters.items():
        for load_percent in [25, 50, 75, 100]:
            load_kw = (load_percent / 100) * converter.rated_power

            # Same method calls work for all converter types
            efficiency = converter.efficiency_at_load(load_kw)
            consumption = converter.fuel_consumption_at_load(load_kw, fuel)

            print(f"{name}: {efficiency:.3f} efficiency, {consumption:.2f} kg/h")
```

Listing 1.5: Universal Comparison Function

Unified Plotting Infrastructure

Both ICE and FC plotting methods can share common infrastructure because they implement the same interface:

```
def create_comparison_plot(converters: List[EnergyConverter]):
    """Single function plots any converter type"""
    for converter in converters:
        load_array = np.linspace(converter.get_operating_limits()['min_load'],
                                 converter.get_operating_limits()['max_load'],
    50)

        # Same method calls for all types
        efficiencies = [converter.efficiency_at_load(load) for load in
    load_array]
        plt.plot(load_array, efficiencies, label=converter.name)
```

Listing 1.6: Shared Plotting Logic

1.5.2 Simplified Extension for New Technologies

The abstract base class makes it trivial to add new converter types to the demonstration scripts:

```
class GasTurbine(EnergyConverter, LoggerMixin):
    def get_operating_limits(self):
        return {'min_load': 0.3 * self.rated_power, 'max_load': self.rated_power
    }

    def efficiency_at_load(self, load):
        # Gas turbine specific calculation
        return 0.35  # Simplified

    # ... implement other abstract methods

```

```
# Immediately works with all existing demonstration code:
gas_turbine = GasTurbine("GT", 5000, config)
demo_results = demonstrate_performance_calculations({'gt': gas_turbine})
```

Listing 1.7: Easy Extension Example

1.5.3 Type Safety and Error Prevention

The abstract base class ensures that incomplete implementations cannot be instantiated:

```
# This would raise TypeError at instantiation:
class IncompleteConverter(EnergyConverter):
    def __init__(self, rated_power):
        super().__init__(rated_power)
    # Missing required abstract methods!

# TypeError: Can't instantiate abstract class IncompleteConverter
# with abstract methods efficiency_at_load, fuel_consumption_at_load,
# get_operating_limits, load_point
```

Listing 1.8: Type Safety Example

This prevents runtime errors and ensures all converters provide the expected functionality for the demonstration scripts.

1.5.4 Data-Driven Configuration

The framework supports creating configurations from test data:

```
# ICE from performance data
ice_config = ICEConfig.from_performance_data(maharaj_data)

# Fuel Cell from test results
fc_config = FuelCellConfig.from_performance_data(pemfc_data)
```

Listing 1.9: Configuration from Test Data

1.6 Plotting and Visualization

1.6.1 Performance Plotting

Both converter types support comprehensive plotting:

1. **Static Performance**: Efficiency and fuel consumption vs. load
2. **Comparative Analysis**: Side-by-side technology comparison

1.6.2 Plot Types and Applications

Plot Type	Purpose	Key Insights
Efficiency vs Load	Design optimization	Optimal operating range
Fuel Consumption	Economic analysis	Operating costs
Technology Comparison	Technology selection	Relative advantages

Table 1.3: Plot Types and Applications

1.7 Real-World Applications

1.7.1 Marine Vessel Integration

The demonstrated functionality addresses real marine applications:

- **Main Propulsion**: High-power, variable load operation
- **Auxiliary Power**: Hotel loads and system power
- **Hybrid Systems**: ICE + FC combinations
- **Emission Reduction**: Zero-emission fuel cell operation

1.7.2 Operational Scenarios

ICE Applications

- High-power main engines (1-20 MW range)
- Variable load following for propulsion
- Auxiliary power generation
- Emergency backup systems

Fuel Cell Applications

- Zero-emission auxiliary power
- Silent operation requirements
- Frequent start-stop cycles
- Grid balancing and frequency regulation

1.8 Validation and Testing

1.8.1 Data Sources

The implementations are validated against multiple data sources:

- **MAHARAJ ICE**: Real test data from D1.3 project deliverable
- **Jalkanen Coefficients**: Literature-based polynomial models
- **Fuel Cell Data**: Performance curves from PEMFC and SOFC tests
- **Industry Standards**: Typical efficiency ranges and operating limits

1.8.2 Validation Results

Key validation points demonstrated in the scripts:

- **Efficiency Values**: Within expected ranges (ICE: 40-55%, FC: 50-60%)
- **Fuel Consumption**: Realistic values for different technologies
- **Operating Limits**: Appropriate minimum and maximum loads
- **Load Response**: Proper efficiency variations with load

1.9 Error Handling and Robustness

1.9.1 Input Validation

The framework includes comprehensive validation:

```
# Power validation
if rated_power <= 0:
    raise ValueError("Rated power cannot be negative")

# Load validation
if load > self.rated_power * 1.1:  # Allow 10% overload
    self.logger.warning("Load exceeds rated power")

# Efficiency validation
if efficiency > 0.85 or efficiency < 0.10:
    self.logger.warning("Efficiency outside expected range")
```

Listing 1.10: Validation Examples

1.9.2 Graceful Degradation

The plotting functions include error handling to ensure robustness:

- **Missing Data**: Graceful handling of missing efficiency curves
- **Calculation Errors**: Fallback methods for performance calculations
- **Plot Failures**: Individual plot failures don't crash entire analysis
- **Configuration Issues**: Clear error messages for configuration problems

1.10 Performance and Scalability

1.10.1 Computational Efficiency

The framework is designed for efficient computation:

- **Vectorized Operations**: NumPy arrays for time series calculations
- **Interpolation**: Fast linear interpolation for efficiency curves
- **Lazy Loading**: Configurations loaded only when needed
- **Caching**: Registry pattern reduces repeated lookups

1.10.2 Memory Management

Efficient memory usage through:

- **Shared Configurations**: Multiple engines can share config objects
- **Data Structures**: Efficient dictionary-based curve storage
- **Plot Management**: Figures can be closed to free memory

1.11 Extension and Customization

1.11.1 Adding New Technologies

The framework can be extended for new converter types:

```
class GasTurbine(EnergyConverter, LoggerMixin):
    def __init__(self, name, rated_power, config=None):
        super().__init__(rated_power)
        # Implementation following same patterns

    def efficiency_at_load(self, load):
        # Technology-specific implementation

    def plot_performance(self, ...):
        # Consistent plotting interface
```

Listing 1.11: Extension Pattern

1.11.2 Custom Configurations

New configurations can be easily added:

- **Test Data Integration**: Direct import from measurement data
- **Literature Models**: Implementation of published correlations
- **Hybrid Methods**: Combination of different calculation approaches
- **Custom Curves**: User-defined performance characteristics

1.12 Future Developments

1.12.1 Planned Enhancements

Several enhancements are planned for future versions:

- **Additional Technologies**: Gas turbines, hybrid systems, energy storage
- **Dynamic Modeling**: Transient response and start-up characteristics
- **Emissions Modeling**: Detailed pollutant and GHG calculations
- **Economic Analysis**: Cost modeling and optimization capabilities

1.12.2 Integration Opportunities

The framework is designed for integration with:

- **Ship Design Tools**: Integration with naval architecture software
- **Energy Management**: Real-time control and optimization systems
- **Environmental Assessment**: Life cycle analysis tools
- **Economic Modeling**: Techno-economic analysis frameworks

1.13 Conclusion

The energy converter framework demonstrates a robust, extensible approach to modeling marine energy systems. The comprehensive demonstration scripts showcase:

- **Realistic Performance Modelling**: Based on real test data and validated correlations
- **Technology Comparison**: Clear comparison between ICE and fuel cell technologies
- **Operational Analysis**: Time-series modelling with realistic load profiles
- **Visualization Capabilities**: Comprehensive plotting for analysis and presentation
- **Extensible Architecture**: Easy addition of new technologies and configurations

The framework provides a solid foundation for marine energy system analysis, supporting both detailed engineering studies and high-level technology assessments. The consistent API and comprehensive validation ensure reliable results across different applications and use cases.

The demonstrated functionality shows the framework's capability to support real-world marine energy system design, from individual component sizing to complete system integration and operational optimization.