Propeller modelling

Technical Documentation Manual

Version 1.0

Ben Noble

June 1, 2025

Contents

1. Ship Propulsion Factors

1.1 Propeller modelling

1.1.1 Propeller Configuration and Performance

The propeller system architecture consists of two complementary classes that manage propeller geometry, operating parameters, and performance characteristics. These classes provide a comprehensive framework for propeller modeling, database management, and performance analysis within the broader ship design system.

Overview

The propeller configuration system enables designers to:

- Define and validate propeller geometric parameters
- Manage propeller databases with flexible loading capabilities
- Store and analyze performance coefficient data
- Handle both fixed and controllable pitch propeller configurations
- Maintain centralized propeller registries for system-wide access
- Validate parameter consistency and physical constraints

System Architecture

The system is built around two primary classes that work together to provide complete propeller modeling capabilities:

ShipPropellerParameters Class Manages physical propeller parameters, geometric specifications, and configuration validation. This class serves as the foundation for propeller definition and includes a registry pattern for centralized management.

ShipPropellerPerformance Class Stores and validates propeller performance coefficient data, providing the link between geometric parameters and hydrodynamic performance characteristics.

Propeller Parameter Management

Basic Propeller Definition Define propeller configurations with comprehensive parameter validation:

```python
from ship_design import ShipPropellerParameters, PropellerType

# Fixed pitch propeller definition
fixed_prop = ShipPropellerParameters(
    diameter=3.2,              # Propeller diameter [m]
    blade_number=4,            # Number of blades
    h=2.8,                     # Water height [m]
    type=PropellerType.FIXED,  # Propeller type
    pitch=2.88,                # Pitch [m] (alternative: pitch_diameter=0.9)
    area_ratio=0.65,           # Expanded blade area ratio
    screws=1,                  # Number of propellers (alternative: K=0.2)
    k_p=0.00003,              # Surface roughness [m]
    name="Main_Propeller"      # Registry name
)

# Controllable pitch propeller (CPP)
cpp_prop = ShipPropellerParameters(
    diameter=4.5,
    blade_number=5,
    h=3.8,
    type=PropellerType.CPP,    # Variable pitch - no fixed pitch needed
    area_ratio=0.7,
    screws=2,                  # Twin screw configuration
    name="CPP_System"
)

print(f"Fixed prop P/D ratio: {fixed_prop._P_D:.3f}")
print(f"Registered propellers: {ShipPropellerParameters.
    list_registered_propellers()}")
```

Listing 1.1: Basic Propeller Configuration

Parameter Validation and Relationships The system automatically validates parameter relationships and calculates derived values:

Pitch Relationships For fixed pitch propellers, the system maintains consistency between pitch (P), diameter (D), and pitch-diameter ratio (P/D). If only two of these are provided, the third is automatically calculated.

Propulsion Configuration The system validates relationships between the number of screws and the K coefficient:

- Single screw: K = 0.2 (default)
- Twin screw: K = 0.1 (default)
- Custom K values automatically derive screw configuration

Physical Constraints All parameters are validated against physical limits and propeller design constraints.

Database Integration Load propeller configurations from external databases with flexible column mapping:

```python
# Load all propellers from database
propeller_database = ShipPropellerParameters.from_database(
    'propeller_database.csv'
)

# Load with specific filtering criteria
```

```python
filtered_propellers = ShipPropellerParameters.from_database(
    'propeller_database.xlsx',
    filter_criteria={
        'blade_number': 4,              # Only 4-blade propellers
        'type': ['fixed', 'FIXED'],     # Only fixed pitch
        'diameter': [3.0, 4.0, 5.0]    # Specific diameter values
    }
)

# Access loaded propellers
main_prop = ShipPropellerParameters.get_propeller('MainEngine_Prop')
aux_prop = ShipPropellerParameters.get_propeller('Auxiliary_Prop')

print(f"Main propeller diameter: {main_prop.D:.2f} m")
print(f"Auxiliary propeller blades: {aux_prop.Z}")
```

Listing 1.2: Database Loading with Filtering

The database loader supports multiple file formats (CSV, Excel, JSON) and flexible column naming conventions. Common column name variations are automatically recognized:

Table 1.1: Database Column Mapping Examples

Parameter	**Recognized Column Names**
Diameter	diameter, D, propeller_diameter
Blade Number	blade_number, blades, Z, number_of_blades
Pitch	pitch, P, propeller_pitch
Area Ratio	area_ratio, aea0, expanded_blade_area_ratio
Type	type, propeller_type

Performance Data Management

Performance Coefficient Storage Store and validate propeller performance data from experimental or computational sources:

```python
# Performance data from experiments or CFD
j_values = np.linspace(0, 1.2, 25)        # Advance coefficient
kt_values = np.array([...])               # Thrust coefficient data
kq_values = np.array([...])               # Torque coefficient data

# Create performance object with validation
propeller_performance = ShipPropellerPerformance(
    advance_coeff=j_values,
    thrust_coeff=kt_values,
    torque_coeff=kq_values,
    verbose=True
)

# Automatic validation ensures array consistency
print(f"Performance data points: {len(propeller_performance.j)}")
print(f"J range: {np.min(propeller_performance.j):.2f} to {np.max(
    propeller_performance.j):.2f}")
```

Listing 1.3: Performance Data Integration

Data Validation The performance class automatically validates coefficient data:

- Ensures all coefficient arrays have identical lengths
- Converts list inputs to numpy arrays for consistency

- Provides detailed logging of validation processes
- Raises informative errors for data inconsistencies

Registry Management

The system implements a class-level registry for centralized propeller management:

```
# Check registry status
current_propellers = ShipPropellerParameters.list_registered_propellers()
registry_count = ShipPropellerParameters.registry_size()

print(f"Registered propellers: {current_propellers}")
print(f"Total in registry: {registry_count}")

# Access specific propeller
try:
    target_prop = ShipPropellerParameters.get_propeller('Main_Propeller')
    print(f"Found propeller with diameter: {target_prop.D} m")
except KeyError as e:
    print(f"Propeller not found: {e}")

# Registry management
ShipPropellerParameters.clear_registry()  # Clear all entries
print(f"Registry after clearing: {ShipPropellerParameters.registry_size()}")
```

Listing 1.4: Registry Operations

Configuration Examples

Single Screw Merchant Vessel Typical configuration for a container ship or bulk carrier:

```
# Large single screw propeller
container_ship_prop = ShipPropellerParameters(
    diameter=8.5,              # Large diameter for efficiency
    blade_number=4,            # Standard for single screw
    h=7.2,                     # Deep immersion
    type=PropellerType.FIXED,
    pitch_diameter=0.85,       # Moderate pitch ratio
    area_ratio=0.6,            # Lower area ratio for efficiency
    screws=1,                  # Single screw
    k_p=0.00003,              # Standard roughness
    name="ContainerShip_Main"
)

# Verify configuration
print(f"Propeller configuration:")
print(f"  Type: {container_ship_prop.type}")
print(f"  Diameter: {container_ship_prop.D} m")
print(f"  Pitch: {container_ship_prop.P:.2f} m")
print(f"  P/D ratio: {container_ship_prop._P_D:.3f}")
print(f"  K coefficient: {container_ship_prop.K}")
```

Listing 1.5: Single Screw Configuration

Twin Screw Naval Vessel Configuration for vessels requiring maneuverability and redundancy:

```
# Twin controllable pitch propellers
naval_propellers = []

for i, side in enumerate(['Port', 'Starboard']):
```

```
    prop = ShipPropellerParameters(
        diameter=4.2,
        blade_number=5,            # Higher blade count for smoother operation
        h=3.5,
        type=PropellerType.CPP,    # Variable pitch for operational flexibility
        area_ratio=0.75,           # Higher area ratio for thrust
        screws=2,                  # Twin screw configuration
        k_p=0.000025,             # Smoother finish for naval application
        name=f"Naval_{side}"
    )
    naval_propellers.append(prop)

# Verify twin screw configuration
port_prop = ShipPropellerParameters.get_propeller('Naval_Port')
print(f"Naval propeller configuration:")
print(f"  Twin screw K coefficient: {port_prop.K}")
print(f"  Variable pitch type: {port_prop.type}")
print(f"  Enhanced area ratio: {port_prop.aea0}")
```

Listing 1.6: Twin Screw Naval Configuration

Integrated Performance Analysis Combine geometric parameters with performance data:

```
# Define propeller geometry
analysis_prop = ShipPropellerParameters(
    diameter=5.5,
    blade_number=4,
    h=4.8,
    type=PropellerType.FIXED,
    pitch_diameter=0.9,
    area_ratio=0.65,
    screws=1,
    name="Analysis_Propeller"
)

# Generate or load performance data
# (Could be from Wageningen B-series, CFD, or experiments)
j_range = np.linspace(0.1, 1.1, 21)
# ... performance coefficient calculation or loading ...
kt_data = np.array([...])  # From analysis or database
kq_data = np.array([...])  # From analysis or database

# Create integrated performance object
propeller_performance = ShipPropellerPerformance(
    advance_coeff=j_range,
    thrust_coeff=kt_data,
    torque_coeff=kq_data
)

# Calculate efficiency
with np.errstate(divide='ignore', invalid='ignore'):
    efficiency = (j_range * kt_data) / (2 * np.pi * kq_data)

# Find optimal operating point
max_eff_idx = np.nanargmax(efficiency)
optimal_j = j_range[max_eff_idx]
max_efficiency = efficiency[max_eff_idx]

print(f"Propeller: {analysis_prop.name}")
print(f"Diameter: {analysis_prop.D} m, P/D: {analysis_prop._P_D:.3f}")
print(f"Optimal efficiency: {max_efficiency:.3f} at J = {optimal_j:.3f}")
```

Listing 1.7: Complete Propeller Analysis Setup

Implementation Notes

Parameter Relationships The system maintains intelligent parameter relationships:

- Automatic calculation of missing geometric parameters
- Validation of physical constraints and practical limits
- Flexible handling of different input parameter combinations
- Warning system for potentially problematic configurations

Note

For fixed pitch propellers, you can specify either the pitch directly or the pitch-diameter ratio. The system will automatically calculate the missing parameter and ensure consistency between all related values.

Type Safety and Validation Comprehensive validation ensures robust operation:

- Type checking and conversion for numeric parameters
- Enumeration-based propeller type management
- Range validation for all physical parameters
- Automatic detection and handling of common input errors

Warning

When loading propellers from databases, ensure that required parameters (diameter, blade_number, h) are present. The system will skip entries with missing critical data and provide detailed error reporting for troubleshooting.

Integration with Propulsion Analysis

These propeller configuration classes integrate seamlessly with the broader propulsion analysis framework:

- Provide geometric parameters for Wageningen B-series calculations
- Enable systematic propeller database studies
- Support optimization workflows across multiple propeller designs
- Facilitate performance comparison and selection processes
- Maintain consistent parameter definitions across analysis modules

The registry pattern ensures that propeller configurations are available system-wide, enabling complex multi-propeller analyses and supporting the development of comprehensive ship design optimization workflows.

1.1.2 Propeller Performance Analysis

Propeller performance is modeled using the Wageningen B-series method, which provides systematic propeller coefficients based on experimental data. The system calculates thrust coefficient (Kt), torque coefficient (Kq), and open-water efficiency () for various operating conditions.

Overview

The Wageningen propeller model enables designers to:

- Calculate propeller coefficients across operating ranges
- Evaluate propeller efficiency characteristics
- Analyze multiple pitch ratios simultaneously
- Assess data quality and apply corrections for robust analysis
- Generate comprehensive performance reports

System Architecture

The model is implemented through the `Wageningen` class, which inherits from `LoggerMixin` for comprehensive logging capabilities. The system consists of several key components:

Input Parameters The class requires fundamental propeller geometry and operating parameters:

- Expanded blade area ratio (Ae/Ao): 0.3 to 1.05
- Number of blades (zp): 2 to 7 blades
- Pitch-to-diameter ratios (P/D): 0.5 to 1.40
- Advance coefficient range (Jc): User-defined array
- Coefficient data files: Kt and Kq coefficient matrices

Data Quality Management The system includes robust data quality tracking and correction mechanisms:

- Automatic detection of negative coefficient values
- Identification of extreme values outside physical limits
- Statistical tracking of data quality metrics
- Optional correction algorithms for invalid data points

Key Functionality

Coefficient Calculations The core calculations implement the Wageningen B-series polynomial expansions:

```
# Initialize Wageningen analysis
propeller = Wageningen(
    AEA0=0.65,              # Expanded blade area ratio
    zp=4,                   # Number of blades
    Pd=np.array([0.8, 1.0, 1.2]),  # Multiple pitch ratios
    Jc=np.linspace(0, 1.2, 25),    # Advance coefficient range
    enable_corrections=True,         # Enable data quality corrections
    quality_report=True             # Generate quality reports
)
```

Listing 1.8: Basic Propeller Analysis Setup

The thrust coefficient Kt is calculated using:

$$K_t = \sum_{i=1}^{n} C_i \cdot J^{s_i} \cdot \left(\frac{P}{D}\right)^{t_i} \cdot \left(\frac{A_e}{A_o}\right)^{u_i} \cdot Z^{v_i}$$

Similarly, the torque coefficient Kq follows:

$$K_q = \sum_{i=1}^{n} C_i \cdot J^{s_i} \cdot \left(\frac{P}{D}\right)^{t_i} \cdot \left(\frac{A_e}{A_o}\right)^{u_i} \cdot Z^{v_i}$$

Individual Coefficient Computation Access specific coefficients for detailed analysis:

```
# Calculate individual coefficients
kt_values = propeller.compute_kt()    # Thrust coefficient
kq_values = propeller.compute_kq()    # Torque coefficient
eta_values = propeller.compute_eta0() # Open-water efficiency

# For single pitch ratio: returns 1D arrays
# For multiple pitch ratios: returns 2D arrays (pitch    J)
print(f"Kt array shape: {kt_values.shape}")
print(f"Efficiency range: {np.nanmin(eta_values):.3f} to {np.nanmax(eta_values)
    :.3f}")
```

Listing 1.9: Individual Coefficient Calculations

Comprehensive Analysis Perform complete propeller analysis with integrated quality control:

```
# Comprehensive analysis with all parameters
j, kt, kq, eta = propeller.compute_all()

# Results structure depends on pitch array size:
# - Single pitch: kt, kq, eta are 1D arrays
# - Multiple pitches: kt, kq, eta are 2D arrays (n_pitches    n_J)

# Find optimal efficiency point
if eta.ndim == 1:
    # Single pitch case
    max_eff_idx = np.nanargmax(eta)
    optimal_j = j[max_eff_idx]
    max_efficiency = eta[max_eff_idx]
else:
    # Multiple pitch case
    max_eff_idx = np.unravel_index(np.nanargmax(eta), eta.shape)
    optimal_j = j[max_eff_idx[1]]
    optimal_pitch = propeller.Pd[max_eff_idx[0]]
    max_efficiency = eta[max_eff_idx]

print(f"Maximum efficiency: {max_efficiency:.3f} at J = {optimal_j:.3f}")
```

Listing 1.10: Complete Propeller Performance Analysis

Data Quality Assessment

The system provides comprehensive data quality monitoring and reporting:

Quality Statistics Tracking The class automatically tracks various data quality metrics:

- Total calculation points processed
- Count of negative Kt and Kq values
- Extreme values outside physical limits
- Invalid efficiency calculations (NaN, infinite, or outside [0,1])
- Number of corrected data points

Quality Report Generation Generate detailed quality assessment reports:

```
# Perform calculations
j, kt, kq, eta = propeller.compute_all()

# Generate comprehensive quality report
quality_report = propeller.get_data_quality_report()
print(quality_report)

# Example output:
# ============================================================
# WAGENINGEN DATA QUALITY REPORT
# ============================================================
# Total calculation points: 1,875
#
# COEFFICIENT ISSUES:
#   Negative Kq values: 125 (6.67%)
#   Negative Kt values: 45 (2.40%)
#   Extreme Kt values: 12 (0.64%)
#   Extreme Kq values: 8 (0.43%)
# ...
```

Listing 1.11: Data Quality Assessment

Correction Mechanisms When `enable_corrections=True`, the system applies intelligent corrections:

- Negative Kq values replaced with small positive values
- Extreme coefficients clipped to reasonable physical limits
- Invalid efficiency values set to NaN for proper handling
- Statistical replacement based on local data characteristics

Usage Examples

Single Pitch Analysis Detailed analysis for a specific pitch ratio:

```
# Analyze single pitch ratio
single_pitch = Wageningen(
    AEA0=0.7,
    zp=5,
    Pd=0.9,  # Single pitch ratio
    Jc=np.linspace(0, 1.4, 50),
    enable_corrections=True
)

# Calculate performance curves
j, kt, kq, eta = single_pitch.compute_all()
```

```

# Find design point (maximum efficiency)
max_eff_idx = np.nanargmax(eta)
design_j = j[max_eff_idx]
design_eta = eta[max_eff_idx]
design_kt = kt[max_eff_idx]

print(f"Design Point:")
print(f"  J = {design_j:.3f}")
print(f"        = {design_eta:.3f}")
print(f"  Kt = {design_kt:.4f}")
```

Listing 1.12: Single Pitch Propeller Analysis

Multi-Pitch Optimization Compare multiple pitch ratios to find optimal configuration:

```
# Analyze multiple pitch ratios
pitch_ratios = np.arange(0.6, 1.3, 0.1)
multi_pitch = Wageningen(
    AEA0=0.65,
    zp=4,
    Pd=pitch_ratios,
    Jc=np.linspace(0.2, 1.2, 30),
    enable_corrections=True
)

# Comprehensive analysis
j, kt, kq, eta = multi_pitch.compute_all()

# Find optimal pitch for each J value
optimal_pitch_indices = np.nanargmax(eta, axis=0)
optimal_efficiencies = np.nanmax(eta, axis=0)

# Create optimization summary
print("J-Value | Optimal P/D | Max Efficiency")
print("-" * 35)
for i, j_val in enumerate(j[::5]):  # Every 5th point for brevity
    if i*5 < len(optimal_pitch_indices):
        opt_pd = pitch_ratios[optimal_pitch_indices[i*5]]
        opt_eff = optimal_efficiencies[i*5]
        print(f"{j_val:6.2f}  |    {opt_pd:.2f}    |     {opt_eff:.3f}")
```

Listing 1.13: Multi-Pitch Optimization Analysis

Performance Visualization Generate performance characteristic plots:

```
# Built-in plotting method
propeller.plot_results()

# Custom plotting for detailed analysis
import matplotlib.pyplot as plt

fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))

# Kt vs J
ax1.plot(j, kt)
ax1.set_xlabel('Advance Coefficient J')
ax1.set_ylabel('Thrust Coefficient Kt')
ax1.grid(True)
ax1.set_title('Thrust Characteristics')

# 10*Kq vs J (traditional scaling)
```

```
ax2.plot(j, 10*kq)
ax2.set_xlabel('Advance Coefficient J')
ax2.set_ylabel('10    Torque Coefficient Kq')
ax2.grid(True)
ax2.set_title('Torque Characteristics')

# Efficiency vs J
ax3.plot(j, eta)
ax3.set_xlabel('Advance Coefficient J')
ax3.set_ylabel('Open-Water Efficiency      ')
ax3.grid(True)
ax3.set_title('Efficiency Characteristics')
ax3.set_ylim([0, 1])

plt.tight_layout()
plt.show()
```

Listing 1.14: Propeller Performance Plotting

Implementation Notes

Array Handling The class intelligently handles both single and multiple pitch scenarios:

- Single pitch (scalar): Returns 1D arrays for direct plotting
- Multiple pitches (array): Returns 2D arrays with shape (n_pitches, n_J)
- Consistent broadcasting ensures proper mathematical operations

Numerical Robustness Several mechanisms ensure reliable calculations:

- Input validation against Wageningen B-series validity ranges
- Numpy error state management for division operations
- Automatic handling of invalid mathematical operations
- Configurable correction algorithms for problematic data points

Warning

The Wageningen B-series method is valid only within specific parameter ranges. The class enforces these limits:

- Ae/Ao: 0.3 to 1.05
- Number of blades: 2 to 7
- P/D: 0.5 to 1.40

Results outside these ranges may not be physically meaningful.

Integration with Propulsion Analysis

This propeller performance module integrates with the broader propulsion system:

- Provides systematic coefficient data for resistance-propulsion matching
- Enables optimization studies across multiple design parameters

- Supports preliminary propeller selection and sizing
- Facilitates performance prediction under varying operational conditions

The robust data quality management and flexible analysis capabilities make this module suitable for both preliminary design studies and detailed propeller performance investigations.

1.1.3 Controllable Pitch Propeller Analysis

The controllable pitch propeller (CPP) analysis system provides comprehensive tools for modeling variable pitch propeller performance and optimization. The system generates multi-dimensional performance grids, enables real-time pitch optimization, and supports operational analysis across varying conditions.

Overview

The CPP analysis framework enables designers to:

- Generate comprehensive performance maps across pitch and advance coefficient ranges
- Perform real-time pitch optimization for specific operating conditions
- Calculate optimal RPM and pitch combinations for given thrust requirements
- Analyze data quality and reliability across the operating envelope
- Visualize performance characteristics through 3D surfaces and contour plots
- Support operational decision-making for variable pitch propeller systems

System Architecture

The system consists of two interconnected classes that provide complete CPP analysis capabilities:

CPPPerformanceCalculator Class Generates and manages multi-dimensional performance data arrays using vectorized Wageningen B-series calculations. This class creates comprehensive lookup tables for thrust coefficient (Kt), torque coefficient (Kq), and efficiency () across ranges of pitch-diameter ratios and advance coefficients.

CPPOptimizer Class Provides optimization algorithms for finding optimal pitch and RPM combinations based on operational requirements. This class uses the performance calculator to determine the most efficient operating points for given speed and thrust conditions.

Performance Grid Generation

Multi-Dimensional Performance Arrays The system creates comprehensive performance grids spanning the complete operating envelope:

```
# Initialize CPP performance calculator
cpp_calculator = CPPPerformanceCalculator(
    propeller_parameters=propeller_params,  # ShipPropellerParameters instance
    j_range=np.arange(0, 1.21, 0.01),      # Advance coefficient: 0 to 1.2 in
    0.01 steps
    pitch_range=np.linspace(0.5, 1.4, 30), # P/D ratio: 0.5 to 1.4 in 30 steps
    verbose=True,
    debug=False
)
```

```
9
10 # Performance data is automatically generated as 3D arrays
11 print(f"Performance grid shape: {cpp_calculator.kt_array.shape}")
12 print(f"Total data points: {cpp_calculator.kt_array.size}")
13 print(f"J range: {cpp_calculator.j_values[0]:.2f} to {cpp_calculator.j_values
       [-1]:.2f}")
14 print(f"P/D range: {cpp_calculator.pitch_values[0]:.2f} to {cpp_calculator.
       pitch_values[-1]:.2f}")
```

Listing 1.15: CPP Performance Calculator Initialization

The performance arrays have shape (n_pitches, n_J) where:

- Each row represents a different pitch-diameter ratio
- Each column represents a different advance coefficient
- Three arrays store Kt, Kq, and efficiency values respectively

Quality-Controlled Data Generation The system integrates robust data quality management:

```
1 # Generate comprehensive quality report
2 quality_report = cpp_calculator.get_data_quality_summary()
3 print(quality_report)
4
5 # Validate specific operating points
6 validation = cpp_calculator.validate_operating_point(j_value=0.8, p_d=1.0)
7 print(f"Operating point valid: {validation['is_valid']}")
8 if validation['warnings']:
9     print(f"Warnings: {validation['warnings']}")
10
11 # Check coefficient ranges for reasonableness
12 cpp_calculator.validate_coefficients()
```

Listing 1.16: Data Quality Assessment

Performance Interpolation and Analysis

Single Parameter Analysis Extract performance characteristics across all pitch values for specific advance coefficients:

```
1 # Get performance values for all pitch settings at J = 0.7
2 j_target = 0.7
3 performance_data = cpp_calculator.get_values_at_j(j_target)
4
5 kt_values = performance_data['kt']
6 kq_values = performance_data['kq']
7 efficiency_values = performance_data['efficiency']
8 pitch_values = performance_data['pitch']
9
10 # Find optimal pitch for maximum efficiency at this J
11 max_eff_idx = np.nanargmax(efficiency_values)
12 optimal_pitch = pitch_values[max_eff_idx]
13 max_efficiency = efficiency_values[max_eff_idx]
14
15 print(f"At J = {j_target}:")
16 print(f"  Optimal P/D = {optimal_pitch:.3f}")
17 print(f"  Maximum efficiency = {max_efficiency:.3f}")
18 print(f"  Corresponding Kt = {kt_values[max_eff_idx]:.4f}")
19 print(f"  Corresponding Kq = {kq_values[max_eff_idx]:.4f}")
```

Listing 1.17: Performance Analysis at Constant J

Bilinear Interpolation Obtain performance values at arbitrary operating points using sophisticated interpolation:

```
# Get interpolated values for specific operating point
target_pd = 0.85   # P/D ratio
target_j = 0.75    # Advance coefficient

interpolated = cpp_calculator.get_interpolated_value(
    p_d=target_pd,
    j_value=target_j
)

print(f"Interpolated performance at P/D={target_pd}, J={target_j}:")
print(f"  Kt = {interpolated['kt']:.6f}")
print(f"  Kq = {interpolated['kq']:.6f}")
print(f"  Efficiency = {interpolated['efficiency']:.4f}")
```

Listing 1.18: Precise Performance Interpolation

The interpolation algorithm uses bilinear interpolation:

$$f(x, y) = f_{00}(1-t)(1-u) + f_{10}t(1-u) + f_{01}(1-t)u + f_{11}tu$$

where t and u are the interpolation factors for pitch and advance coefficient respectively.

Pitch Optimization

Target-Based Optimization Find optimal pitch settings for specific thrust requirements:

```
# Find optimal pitch for target thrust coefficient
j_operating = 0.8
target_kt = 0.25
tolerance = 0.01  # 1% tolerance

optimal_pitch, max_efficiency, corresponding_kq = cpp_calculator.
    find_optimal_pitch(
    j_value=j_operating,
    target_kt=target_kt,
    tolerance=tolerance
)

print(f"Optimization results:")
print(f"  Optimal P/D: {optimal_pitch:.4f}")
print(f"  Achieved efficiency: {max_efficiency:.4f}")
print(f"  Corresponding Kq: {corresponding_kq:.6f}")
```

Listing 1.19: Optimal Pitch Finding

Complete Performance Calculation Calculate comprehensive CPP performance for operational conditions:

```
# Define operating conditions
j_value = 0.7
required_thrust = 250000  # Newtons (250 kN)
rotational_speed = 3.0    # rev/s (180 RPM)
water_density = 1025      # kg/m

# Calculate complete performance
performance = cpp_calculator.get_cpp_performance(
    j_value=j_value,
    required_thrust=required_thrust,
    water_density=water_density,
```

```
    n=rotational_speed
)

print(f"CPP Performance Analysis:")
print(f"  Optimal P/D ratio: {performance['P_D']:.4f}")
print(f"  Thrust coefficient: {performance['Kt']:.6f}")
print(f"  Torque coefficient: {performance['Kq']:.6f}")
print(f"  Efficiency: {performance['efficiency']:.4f}")
print(f"  Generated thrust: {performance['thrust']/1000:.1f} kN")
print(f"  Required torque: {performance['torque']/1000:.1f} kN m")
print(f"  Power requirement: {performance['power']/1000:.1f} kW")
```

Listing 1.20: Complete CPP Performance Analysis

Operational Optimization

Multi-Variable Optimization The CPPOptimizer class provides comprehensive optimization for real operational scenarios:

```
# Initialize the optimizer
optimizer = CPPOptimizer(
    cpp_calculator=cpp_calculator,
    propeller_diameter=propeller_params.D,
    rpm_range=(120, 250)  # Operational RPM limits
)

# Define operational requirements
ship_speed = 7.5      # m/s (approximately 15 knots)
thrust_required = 180000  # N (180 kN)

# Find optimal operating point
optimization_results = optimizer.find_optimal_pd_rpm(
    speed_of_advance=ship_speed,
    required_thrust=thrust_required,
    rpm_step=5,           # RPM search increment
    thrust_tolerance=0.05  # 5% thrust matching tolerance
)

optimal = optimization_results['optimal']
print(f"Optimal Operating Point:")
print(f"  RPM: {optimal['rpm']:.1f}")
print(f"  P/D ratio: {optimal['P_D']:.4f}")
print(f"  Efficiency: {optimal['efficiency']:.4f}")
print(f"  Power required: {optimal['power_kw']:.1f} kW")
```

Listing 1.21: CPP Optimizer Initialization and Usage

Refined Search Algorithm Perform high-resolution optimization around promising operating points:

```
# Perform refined search around optimal point
refined_results = optimizer.refined_search(
    speed_of_advance=ship_speed,
    required_thrust=thrust_required,
    initial_result=optimization_results,
    rpm_range_factor=0.1,  #  10 % around initial optimum
    rpm_step_fine=1.0      # 1 RPM resolution
)

refined_optimal = refined_results['optimal']
print(f"Refined Optimal Point:")
print(f"  RPM: {refined_optimal['rpm']:.1f}")
```

```
print(f"  P/D ratio: {refined_optimal['P_D']:.4f}")
print(f"  Efficiency: {refined_optimal['efficiency']:.4f}")
print(f"  Power: {refined_optimal['power_kw']:.1f} kW")

# Compare improvements
efficiency_improvement = refined_optimal['efficiency'] - optimal['efficiency']
power_reduction = optimal['power_kw'] - refined_optimal['power_kw']
print(f"Refinement benefits:")
print(f"  Efficiency gain: {efficiency_improvement:.4f}")
print(f"  Power reduction: {power_reduction:.1f} kW")
```

Listing 1.22: Refined Optimization Search

Visualization and Analysis

Performance Surface Plots Generate comprehensive 3D visualizations of CPP performance characteristics:

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create comprehensive performance visualization
fig = plt.figure(figsize=(15, 10))

# Thrust coefficient surface
ax1 = fig.add_subplot(221, projection='3d')
cpp_calculator.plot_kt_surface(ax=ax1)

# Efficiency surface
ax2 = fig.add_subplot(222, projection='3d')
cpp_calculator.plot_efficiency_surface(ax=ax2)

# Data quality map
ax3 = fig.add_subplot(223)
cpp_calculator.plot_data_quality_map(ax=ax3)

# Optimization results
ax4 = fig.add_subplot(224)
# (Optimization plotting would be added here)

plt.tight_layout()
plt.show()
```

Listing 1.23: 3D Performance Visualization

Optimization Results Visualization Comprehensive plotting of optimization outcomes:

```
# Plot optimization results
optimization_plot = optimizer.plot_optimization_results(
    results=optimization_results,
    speed_of_advance=ship_speed
)

# The plot automatically shows:
# - Efficiency vs RPM
# - Power vs RPM
# - P/D ratio vs RPM
# - Efficiency vs Power (Pareto frontier)

plt.show()
```

Listing 1.24: Optimization Results Visualization

Practical Application Examples

Variable Speed Operation Analysis Analyze CPP performance across multiple operating speeds:

```
# Define operational envelope
speeds = np.arange(5, 12, 1)  # 5 to 11 m/s
thrust_curve = [120000, 140000, 165000, 195000, 230000, 270000, 320000]  # N

optimization_summary = []

for speed, thrust in zip(speeds, thrust_curve):
    try:
        result = optimizer.find_optimal_pd_rpm(
            speed_of_advance=speed,
            required_thrust=thrust,
            rpm_step=10
        )

        optimal = result['optimal']
        optimization_summary.append({
            'speed_ms': speed,
            'speed_knots': speed * 1.944,
            'thrust_kn': thrust / 1000,
            'rpm': optimal['rpm'],
            'pd_ratio': optimal['P_D'],
            'efficiency': optimal['efficiency'],
            'power_kw': optimal['power_kw']
        })

    except ValueError as e:
        print(f"Optimization failed for speed {speed} m/s: {e}")

# Display optimization summary
print("Speed | Thrust | RPM  | P/D   | Eff.  | Power")
print("(kts) | (kN)   |      |       |       | (kW)")
print("-" * 45)
for result in optimization_summary:
    print(f"{result['speed_knots']:4.1f}  | {result['thrust_kn']:5.1f}  | "
          f"{result['rpm']:4.0f} | {result['pd_ratio']:.3f} | "
          f"{result['efficiency']:.3f} | {result['power_kw']:5.0f}")
```

Listing 1.25: Multi-Speed CPP Analysis

Economic Optimization Incorporate fuel consumption and operating costs:

```
# Economic analysis parameters
fuel_cost_per_kg = 0.65  # euros/kg
fuel_lhv = 11.9          # kWh/kg (marine diesel)
engine_efficiency = 0.45  # Typical marine diesel efficiency

economic_analysis = []

for result in optimization_summary:
    # Calculate fuel consumption
    brake_power = result['power_kw'] / engine_efficiency
    fuel_consumption_rate = brake_power / fuel_lhv  # kg/h

    # Daily fuel consumption at this speed
    daily_fuel = fuel_consumption_rate * 24  # kg/day
    daily_fuel_cost = daily_fuel * fuel_cost_per_kg  # euros/day

    # Distance covered per day
```

```
    daily_distance = result['speed_ms'] * 24 * 3.6 / 1000  # km/day

    economic_analysis.append({
        'speed_knots': result['speed_knots'],
        'efficiency': result['efficiency'],
        'fuel_rate_kg_h': fuel_consumption_rate,
        'daily_fuel_kg': daily_fuel,
        'daily_cost_eur': daily_fuel_cost,
        'daily_distance_km': daily_distance,
        'fuel_per_km': daily_fuel / daily_distance
    })

print("Economic Analysis:")
print("Speed | Eff.  | Fuel Rate | Daily Cost | Fuel/km")
print("(kts) |       | (kg/h)    | (EUR)      | (kg/km)")
print("-" * 50)
for econ in economic_analysis:
    print(f"{econ['speed_knots']:4.1f}  | {econ['efficiency']:.3f} | "
          f"{econ['fuel_rate_kg_h']:8.1f}  | {econ['daily_cost_eur']:9.0f}  | "
          f"{econ['fuel_per_km']:6.3f}")
```

Listing 1.26: Economic Performance Analysis

Implementation Notes

Computational Efficiency The system uses several strategies to ensure computational efficiency:

- Vectorized Wageningen calculations for rapid grid generation
- Pre-computed performance arrays for real-time interpolation
- Efficient search algorithms with configurable resolution
- Caching of intermediate results to avoid redundant calculations

Numerical Robustness Comprehensive error handling ensures reliable operation:

- Automatic clamping of input values to valid ranges
- Graceful handling of interpolation edge cases
- Quality validation for all interpolated results
- Fallback strategies for optimization failures

Note

The CPP system uses bilinear interpolation for performance data. While this provides good accuracy for most applications, higher-order interpolation methods could be implemented for critical applications requiring maximum precision.

Data Quality Management The system provides comprehensive data quality assessment:

- Real-time validation of coefficient values
- Quality maps showing reliable operating regions
- Statistical reporting of data issues

- Warning systems for questionable operating points

Warning

CPP optimization results should be validated against the data quality maps. Operating points in regions with poor data quality may not provide reliable performance predictions and should be used with caution.

Integration with Ship Design Framework

The CPP analysis system integrates seamlessly with the broader ship design framework:

- Uses ShipPropellerParameters for consistent geometry definition
- Leverages Wageningen B-series calculations for performance prediction
- Provides optimization data for propulsion system design
- Supports economic analysis through fuel consumption modeling
- Enables operational planning and route optimization studies

The modular design allows the CPP system to be used independently or as part of comprehensive ship design optimization workflows, supporting both preliminary design studies and detailed operational analysis.

1.1.4 Propeller Data Management and Integration

The propeller data management system provides comprehensive tools for importing propeller configurations from various data sources and integrating them into a unified propeller analysis framework. The system combines flexible data import capabilities with sophisticated propeller modeling to support both fixed-pitch and controllable-pitch propeller analysis.

Overview

The propeller data management framework enables designers to:

- Import propeller configurations from diverse file formats (CSV, Excel, JSON)
- Handle flexible column naming conventions and data variations
- Create comprehensive propeller instances with integrated performance modeling
- Process single propellers or entire databases automatically
- Generate template files for standardized data collection
- Combine geometric parameters with performance data seamlessly
- Support both Wageningen B-series calculations and experimental data

System Architecture

The system consists of two primary components that work together to provide complete propeller data management:

PropellerImporter Class Extends the DataImporter framework to provide specialized propeller data handling capabilities. This class manages the conversion of raw data into validated propeller configurations with comprehensive error handling and flexible column mapping.

ShipPropeller Class The main propeller class that integrates geometric parameters, performance data, and analysis capabilities. This class automatically handles both fixed-pitch and controllable-pitch propellers with intelligent initialization and caching mechanisms.

Data Import and Processing

Flexible Column Mapping The system supports various naming conventions through comprehensive column mapping:

```
from ship_design import PropellerImporter, PropellerType

# Initialize the importer
importer = PropellerImporter(
    verbose=True,
    debug=False,
    cache_dir="./propeller_cache"
)

# Import from file with automatic column detection
propeller = importer.import_propeller_file(
    file_path="propeller_data.xlsx",
    propeller_name="MainEngine_Prop",
    propeller_type=PropellerType.FIXED,
    screws=1,
    sheet_name="PropellerData",
    strip_whitespace=True
)

print(f"Imported propeller: {propeller.parameters.name}")
print(f"Diameter: {propeller.parameters.D} m")
print(f"Blades: {propeller.parameters.Z}")
print(f"Type: {propeller.parameters.type}")
```

Listing 1.27: Propeller Data Import with Flexible Naming

The column mapping system recognizes multiple naming conventions automatically:

Table 1.2: Flexible Column Naming Recognition

Parameter	Recognized Names
Diameter	diameter, D, propeller_diameter, prop_diameter
Blade Number	blade_number, blades, Z, number_of_blades
Water Height	h, water_height, immersion_depth
Pitch	pitch, P, propeller_pitch, prop_pitch
Area Ratio	area_ratio, ae0, aea0, expanded_blade_area_ratio
Performance Data	J/j, kt/Kt, kq/Kq, advance_coeff

Database Import with Filtering Process multiple propellers from databases with sophisticated filtering capabilities:

```
# Import entire propeller database with filtering
propeller_database = importer.import_propeller_database(
    file_path="propeller_database.csv",
    filter_criteria={
        'blade_number': [4, 5],              # Only 4 or 5 blade propellers
        'type': ['FIXED', 'fixed'],          # Only fixed pitch
        'diameter': (3.0, 6.0)               # Diameter range 3-6 meters
    },
    columns_to_rename={
```

```
        'prop_dia': 'diameter',              # Rename non-standard columns
        'num_blades': 'blade_number'
    },
    use_cache=True,
    save_cache=True
)

print(f"Imported {len(propeller_database)} propellers")
for name, prop in propeller_database.items():
    print(f"  {name}: {prop.parameters.D}m, {prop.parameters.Z} blades")
```

Listing 1.28: Database Import with Filtering

Directory Processing Batch import propellers from entire directories:

```
# Process all files in a directory
directory_propellers = importer.import_propeller_directory(
    directory_path="./propeller_configs/",
    file_types=['.csv', '.xlsx', '.json'],
    propeller_type=PropellerType.FIXED,    # Default type for all files
    screws=1,                              # Default configuration
    row_index=0,                           # Use first row of each file
    use_cache=True
)

print(f"Processed {len(directory_propellers)} files")
for filename, prop in directory_propellers.items():
    performance_source = "Experimental" if prop.performance else "Wageningen"
    print(f"  {filename}: {performance_source} performance data")
```

Listing 1.29: Directory Batch Processing

Performance Data Integration

Mixed Data Source Handling The system intelligently handles various performance data formats:

```
# Create template for data collection
template_df = PropellerImporter.create_template_dataframe()
print("Template columns:", template_df.columns.tolist())

# Save template for data collection
importer.save_template_csv("propeller_template.csv")

# The template shows expected format for coefficient arrays:
# J: "[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8]"
# kt: "[0.45, 0.42, 0.39, 0.35, 0.30, 0.25, 0.18, 0.10, 0.02]"
# kq: "[0.065, 0.062, 0.058, 0.053, 0.047, 0.040, 0.032, 0.023, 0.012]"
```

Listing 1.30: Performance Data Processing

The system automatically parses performance coefficient arrays from various formats:

- Numpy arrays stored directly in files
- Python lists in string format
- Comma or space-separated value strings
- Bracketed array representations

Comprehensive Propeller Modeling

Intelligent Initialization The ShipPropeller class automatically configures itself based on propeller type and available data:

```
# Create propeller with automatic performance generation
propeller = ShipPropeller(
    parameters=propeller_params,           # ShipPropellerParameters instance
    performance=None,                      # Will auto-generate if needed
    wageningen=True,                       # Enable Wageningen calculations
    verbose=True,
    debug=False
)

# For fixed pitch: generates single P/D performance curve
# For CPP: creates multi-dimensional performance grid

print(f"Propeller type: {propeller.parameters.type}")
print(f"Performance points: {len(propeller.performance.j) if propeller.
    performance else 'CPP Grid'}")
print(f"CPP calculator available: {propeller.cpp_calculator is not None}")
```

Listing 1.31: Comprehensive Propeller Creation

Performance Analysis Methods Access comprehensive performance characteristics through unified methods:

```
# Basic performance calculations
j_operating = 0.7
kt_value = propeller.get_Kt(j_operating)
kq_value = propeller.get_Kq(j_operating)
efficiency = propeller.get_efficiency(j_operating)

print(f"At J = {j_operating}:")
print(f"  Kt = {kt_value:.4f}")
print(f"  Kq = {kq_value:.4f}")
print(f"       = {efficiency:.3f}")

# Calculate advance coefficient from operational conditions
va_speed = 8.5  # m/s
rpm = 180       # RPM
n_rps = rpm / 60  # Convert to revolutions per second

j_calculated = propeller.calculate_J(va_speed, n_rps)
print(f"Calculated J = {j_calculated:.3f} for Va={va_speed}m/s at {rpm}RPM")

# Find RPM for target advance coefficient
target_j = 0.8
required_rps = propeller.find_rpm_from_J(target_j, va_speed)
required_rpm = required_rps * 60

print(f"Required RPM: {required_rpm:.0f} for J={target_j} at Va={va_speed}m/s")
```

Listing 1.32: Performance Analysis Interface

Propeller Matching Capabilities Support traditional propeller-hull matching procedures:

```
# Calculate Kt/J   for matching procedures
ktj2_array = propeller.KtJ2
print(f"Kt/J   range: {np.min(ktj2_array):.4f} to {np.max(ktj2_array):.4f}")

# Find operating point from hull resistance curve
```

```
target_ktj2 = 2.5  # From hull resistance analysis
matching_j = propeller.find_J_for_KtJ2(target_ktj2)

if matching_j:
    matching_kt = propeller.get_Kt(matching_j)
    matching_efficiency = propeller.get_efficiency(matching_j)

    print(f"Matching point found:")
    print(f"  J = {matching_j:.3f}")
    print(f"  Kt = {matching_kt:.4f}")
    print(f"       = {matching_efficiency:.3f}")

    # Calculate required RPM for this operating point
    ship_speed = 7.2  # m/s
    operating_rps = propeller.find_rpm_from_J(matching_j, ship_speed)
    print(f"  Required RPM: {operating_rps * 60:.0f}")
```

Listing 1.33: Propeller-Hull Matching Support

Controllable Pitch Propeller Operations

Automatic CPP Initialization For controllable pitch propellers, the system automatically creates comprehensive performance grids:

```
# CPP propellers automatically get multi-dimensional calculators
if propeller.parameters.type == PropellerType.CPP:
    print("CPP capabilities available:")
    print(f"  Performance grid: {propeller.cpp_calculator.kt_array.shape}")
    print(f"  Pitch range: {propeller.cpp_calculator.pitch_values[0]:.2f} - {
    propeller.cpp_calculator.pitch_values[-1]:.2f}")
    print(f"  J range: {propeller.cpp_calculator.j_values[0]:.2f} - {propeller.
    cpp_calculator.j_values[-1]:.2f}")

    # Data quality assessment
    quality_report = propeller.cpp_calculator.get_data_quality_summary()
    print("Data quality summary available")
```

Listing 1.34: CPP Automatic Setup

CPP Optimization Methods Comprehensive optimization capabilities for controllable pitch operations:

```
# Optimal pitch for specific conditions
j_target = 0.75
required_thrust = 200000  # N
rotational_speed = 2.5    # rps

# Get optimal pitch for target thrust
cpp_performance = propeller.get_cpp_performance(
    j_value=j_target,
    required_thrust=required_thrust,
    n=rotational_speed
)

print(f"CPP Performance at J={j_target}:")
print(f"  Optimal P/D: {cpp_performance['P_D']:.3f}")
print(f"  Efficiency: {cpp_performance['efficiency']:.3f}")
print(f"  Power required: {cpp_performance['power']/1000:.1f} kW")

# Complete RPM and pitch optimization
ship_speed = 8.0  # m/s
thrust_needed = 180000  # N
```

```

optimization_result = propeller.cpp_optimise_rpm_pd(
    required_thrust=thrust_needed,
    speed_of_advance=ship_speed,
    refined_search=True,
    thrust_tolerance=0.03
)

print(f"Optimal operating point:")
print(f"  RPM: {optimization_result['rpm']:.0f}")
print(f"  P/D: {optimization_result['P_D']:.3f}")
print(f"  Efficiency: {optimization_result['efficiency']:.3f}")
print(f"  Power: {optimization_result['power_kw']:.1f} kW")
```

Listing 1.35: CPP Optimization Operations

Engine-Constrained Optimization Practical optimization within engine operating limits:

```
# Optimize within engine operational envelope
engine_rated_rpm = 1800
rpm_tolerance = 0.15  #  15 % of rated RPM

constrained_result = propeller.cpp_optimise_rpm_pd_engine_constrained(
    required_thrust=thrust_needed,
    speed_of_advance=ship_speed,
    engine_rated_rpm=engine_rated_rpm,
    rpm_tolerance=rpm_tolerance,
    rpm_step=5.0
)

print(f"Engine-constrained optimization:")
print(f"  Optimal RPM: {constrained_result['rpm']:.0f}")
print(f"  Engine loading: {constrained_result['engine_loading_percent']:.1f}%")
print(f"  RPM deviation: {constrained_result['rpm_deviation']:.0f} RPM")
print(f"  Optimal P/D: {constrained_result['P_D']:.3f}")
print(f"  Efficiency: {constrained_result['efficiency']:.3f}")

# Check if within acceptable engine limits
if constrained_result['rpm_deviation_percent'] < 10:
    print("    Operating point within preferred engine range")
else:
    print("    Operating point approaches engine limits")
```

Listing 1.36: Engine-Constrained CPP Optimization

Visualization and Analysis

Performance Curve Visualization Generate comprehensive performance plots for analysis and presentation:

```
# Plot standard propeller performance curves
propeller.plot_performance()

# For CPP propellers, plot 3D performance surfaces
if propeller.parameters.type == PropellerType.CPP:
    cpp_surfaces = propeller.plot_cpp_surfaces()

    # Plot data quality assessment
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

    # Quality map
    propeller.cpp_calculator.plot_data_quality_map(ax=ax1)
```

```

    # Coefficient validation
    propeller.cpp_calculator.validate_coefficients()

    plt.tight_layout()
    plt.show()
```

Listing 1.37: Propeller Performance Visualization

Practical Application Examples

Multi-Propeller System Analysis Compare multiple propeller configurations for design optimization:

```
# Import multiple propeller options
propeller_options = importer.import_propeller_database(
    "design_candidates.xlsx",
    filter_criteria={'diameter': [4.0, 4.5, 5.0]}
)

# Analyze each option at design conditions
design_speed = 12.0  # knots
design_va = design_speed * 0.5144  # Convert to m/s
target_thrust = 150000  # N

comparison_results = []

for name, prop in propeller_options.items():
    # Calculate design point performance
    if prop.parameters.type == PropellerType.CPP:
        result = prop.cpp_optimise_rpm_pd(
            required_thrust=target_thrust,
            speed_of_advance=design_va
        )
        comparison_results.append({
            'name': name,
            'diameter': prop.parameters.D,
            'type': 'CPP',
            'optimal_rpm': result['rpm'],
            'optimal_pd': result['P_D'],
            'efficiency': result['efficiency'],
            'power_kw': result['power_kw']
        })
    else:
        # Fixed pitch analysis
        target_ktj2 = target_thrust / (design_va**2 * SEAWATER_DENSITY)
        design_j = prop.find_J_for_KtJ2(target_ktj2)
        if design_j:
            efficiency = prop.get_efficiency(design_j)
            comparison_results.append({
                'name': name,
                'diameter': prop.parameters.D,
                'type': 'Fixed',
                'design_j': design_j,
                'efficiency': efficiency,
                'fixed_pd': prop.parameters._P_D
            })

# Display comparison
print("Propeller Design Comparison:")
print("Name                | Dia  | Type | Efficiency | Power/RPM")
print("-" * 55)
for result in comparison_results:
```

```
    if result['type'] == 'CPP':
        print(f"{result['name']:18} | {result['diameter']:.1f}m | CPP  | "
              f"{result['efficiency']:.3f}     | {result['power_kw']:.0f}kW@{
    result['optimal_rpm']:.0f}")
    else:
        print(f"{result['name']:18} | {result['diameter']:.1f}m | Fix  | "
              f"{result['efficiency']:.3f}     | P/D={result['fixed_pd']:.2f}")
```

Listing 1.38: Multi-Propeller Comparison Study

Operational Envelope Analysis Analyze propeller performance across the complete operational range:

```
# Define operational envelope
speeds_knots = np.arange(8, 16, 1)  # 8 to 15 knots
speeds_ms = speeds_knots * 0.5144    # Convert to m/s

# Thrust curve based on resistance estimates
base_thrust = 120000  # N at design speed
thrust_curve = base_thrust * (speeds_ms / speeds_ms[4])**2  # Square law
    approximation

envelope_analysis = []

for speed, thrust in zip(speeds_ms, thrust_curve):
    if propeller.parameters.type == PropellerType.CPP:
        try:
            result = propeller.cpp_optimise_rpm_pd_engine_constrained(
                required_thrust=thrust,
                speed_of_advance=speed,
                engine_rated_rpm=1800,
                rpm_tolerance=0.20
            )

            envelope_analysis.append({
                'speed_kts': speed / 0.5144,
                'thrust_kn': thrust / 1000,
                'rpm': result['rpm'],
                'pd_ratio': result['P_D'],
                'efficiency': result['efficiency'],
                'power_kw': result['power_kw'],
                'engine_loading': result['engine_loading_percent']
            })

        except Exception as e:
            print(f"Analysis failed at {speed:.1f} m/s: {e}")

# Generate operational summary
print("Operational Envelope Analysis:")
print("Speed | Thrust | RPM  | P/D   | Eff.  | Power | Engine")
print("(kts) | (kN)   |      |       |       | (kW)  | Load%")
print("-" * 60)
for analysis in envelope_analysis:
    print(f"{analysis['speed_kts']:4.1f}  | {analysis['thrust_kn']:5.1f}  | "
          f"{analysis['rpm']:4.0f} | {analysis['pd_ratio']:.3f} | "
          f"{analysis['efficiency']:.3f} | {analysis['power_kw']:5.0f} | "
          f"{analysis['engine_loading']:4.1f}%")
```

Listing 1.39: Operational Envelope Analysis

Implementation Notes

Performance Caching The system implements intelligent caching to improve computational efficiency:

- Coefficient interpolation results are cached by J value
- CPP performance grids are pre-computed and stored
- File import operations leverage DataImporter caching
- Optimization results can be cached for repeated analyses

Error Handling and Validation Comprehensive validation ensures robust operation:

- Input parameter validation against physical constraints
- Graceful handling of missing or corrupted performance data
- Automatic fallback to Wageningen calculations when needed
- Detailed logging for troubleshooting data import issues

Note

When importing propeller databases, the system will attempt to create propeller instances even with incomplete data. Missing performance coefficients will trigger automatic Wageningen calculations if the area ratio is available.

Warning

CPP optimization methods require sufficient Wageningen coefficient data coverage. Ensure that the area ratio (Ae/Ao) is within the valid Wageningen range (0.3-1.05) for reliable results.

Integration with Design Framework

The propeller data management system integrates seamlessly with the broader ship design framework:

- Provides standardized propeller objects for resistance-propulsion matching
- Supports optimization workflows with consistent data interfaces
- Enables systematic design studies across multiple propeller configurations
- Facilitates integration with fuel consumption and economic analysis modules
- Maintains compatibility with both preliminary and detailed design processes

The flexible import system and comprehensive modeling capabilities make this framework suitable for both individual propeller analysis and large-scale parametric design studies, supporting the complete propeller design and selection process.

1.1.5 Practical Application Examples

This section demonstrates comprehensive practical applications of the propeller analysis framework through detailed use cases. These examples illustrate real-world implementation scenarios, from basic fixed-pitch propeller analysis to advanced controllable-pitch optimization, providing designers with practical templates for common propeller analysis tasks.

Use Case 1: Fixed-Pitch Propeller with Wageningen Performance

This example demonstrates the analysis of a fixed-pitch propeller using Wageningen B-series calculations for performance prediction. This approach is commonly used in preliminary design when experimental data is unavailable.

Configuration and Setup Establish propeller parameters and generate performance characteristics:

```
# Define propeller geometric parameters
params = ShipPropellerParameters(
    diameter=9.5,                    # 9.5 meters diameter
    blade_number=4,                  # 4 blades
    h=0.5,                          # 0.5 meters water height
    type=PropellerType.FIXED,
    pitch_diameter=0.8,             # P/D ratio of 0.8
    area_ratio=0.4077,              # Expanded blade area ratio
    screws=1,                       # Single screw configuration
    name="MainEngine_Propeller"
)

# Create propeller with automatic Wageningen performance generation
prop = ShipPropeller(
    parameters=params,
    wageningen=True,
    verbose=True
)

# Display basic propeller information
print(f"Propeller: {prop.parameters.name}")
print(f"Diameter: {prop.parameters.D} m")
print(f"P/D ratio: {prop.parameters._P_D}")
print(f"Area ratio: {prop.parameters.aea0}")
print(f"Performance points: {len(prop.performance.j)}")

# Generate performance visualization
prop.plot_performance()
```

Listing 1.40: Fixed-Pitch Propeller Setup with Wageningen

Operating Point Analysis Calculate performance characteristics at specific operational conditions:

```
# Define operating conditions
Va = 5.0    # Ship speed of advance [m/s]
n = 2.0     # Propeller rotation speed [rps] (120 RPM)

# Calculate advance coefficient
J = prop.calculate_J(Va, n)
print(f"Operating point: Va={Va} m/s, n={n} rps ({n*60} rpm)")
print(f"Advance coefficient J = {J:.4f}")

# Obtain performance coefficients
Kt = prop.get_Kt(J)
Kq = prop.get_Kq(J)
efficiency = prop.get_efficiency(J)

print(f"Performance coefficients at J={J:.4f}:")
print(f"  Thrust coefficient Kt = {Kt:.6f}")
print(f"  Torque coefficient Kq = {Kq:.6f}")
print(f"  Open-water efficiency       = {efficiency:.4f}")

```

```
# Calculate physical performance values
rho = 1025.0  # Seawater density [kg/m ]
D = params.D

thrust = Kt * rho * n**2 * D**4      # Thrust [N]
torque = Kq * rho * n**2 * D**5      # Torque [ N m ]
power = 2 * np.pi * n * torque       # Power [W]

print(f"Physical performance values:")
print(f"  Thrust = {thrust/1000:.2f} kN")
print(f"  Torque = {torque/1000:.2f}  k N m ")
print(f"  Power = {power/1000:.2f} kW")
```

Listing 1.41: Operating Point Performance Analysis

Propeller-Hull Matching Analysis Demonstrate traditional propeller-hull matching procedures:

```
# Hull resistance analysis provides target Kt/J   value
ktj2_target = 0.16  # Example value from resistance calculations

# Find matching advance coefficient
matched_j = prop.find_J_for_KtJ2(ktj2_target)
print(f"Propeller-hull matching:")
print(f"  Target Kt/J   = {ktj2_target:.4f}")
print(f"  Matched advance coefficient J = {matched_j:.4f}")

# Calculate required RPM for design speed
Va_design = 6.0  # Design speed [m/s]
n_required = prop.find_rpm_from_J(matched_j, Va_design)
rpm_required = n_required * 60

print(f"  At design speed Va = {Va_design} m/s:")
print(f"  Required RPM = {rpm_required:.1f}")

# Performance at matched operating point
Kt_matched = prop.get_Kt(matched_j)
efficiency_matched = prop.get_efficiency(matched_j)
thrust_matched = Kt_matched * rho * n_required**2 * D**4

print(f"  Matched point performance:")
print(f"    Efficiency       = {efficiency_matched:.4f}")
print(f"    Thrust = {thrust_matched/1000:.2f} kN")
```

Listing 1.42: Propeller-Hull Matching Procedure

Use Case 2: Fixed-Pitch Propeller with Experimental Data

This example shows how to integrate experimental or CFD-derived performance data into the analysis framework, providing higher accuracy for specific propeller designs.

Data Import and Integration Import propeller with custom performance coefficients:

```
# Define propeller with experimental performance data
experimental_data = {
    'name': ['MAHARAJ_Propeller'],
    'diameter': [9.5],
    'blade_number': [4],
    'h': [0.5],
    'area_ratio': [0.4077],
    'J': [np.array([0, 0.09, 0.16, 0.2, 0.225, 0.250, 0.275, 0.3,
```

```
                    0.325, 0.35, 0.4, 0.45, 0.5, 0.54, 0.6, 0.65,
                    0.71, 0.78, 0.85, 0.89])],
    'kt': [np.array([0.308, 0.289, 0.274, 0.261, 0.254, 0.248, 0.241,
                     0.234, 0.226, 0.220, 0.203, 0.186, 0.168, 0.146,
                     0.122, 0.103, 0.075, 0.046, 0.012, 0])],
    'kq': [np.array([0.0348, 0.0334, 0.0314, 0.0295, 0.0289, 0.0283,
                     0.0276, 0.0269, 0.0262, 0.0255, 0.0241, 0.0225,
                     0.0209, 0.0210, 0.0196, 0.0174, 0.0150, 0.0110,
                     0.0068, 0.0050])]
}

# Create DataFrame and import
df = pd.DataFrame(experimental_data)
importer = PropellerImporter(verbose=True)
prop_experimental = importer.from_dataframe(df)

print(f"Experimental propeller imported:")
print(f"  Name: {prop_experimental.parameters.name}")
print(f"  Diameter: {prop_experimental.parameters.D} m")
print(f"  Performance data points: {len(prop_experimental.performance.j)}")

# Visualize experimental performance
prop_experimental.plot_performance()
```

Listing 1.43: Custom Performance Data Integration

Multi-Speed Performance Analysis Analyze performance across operational speed range:

```
# Performance analysis at various ship speeds
print("Performance Analysis Across Speed Range:")
print("-" * 75)
print(f"{'Speed (knots)':<15} {'J':<10} {'Kt':<12} {'Kq':<12} {'Efficiency':<12}
    ")
print("-" * 75)

n_fixed = 3.0  # Fixed rotation speed [rps] (180 RPM)

# Analyze performance from 10 to 20 knots
for speed_knots in range(10, 21, 2):
    Va = speed_knots * 0.5144  # Convert knots to m/s

    # Calculate operating point
    J = prop_experimental.calculate_J(Va, n_fixed)

    # Get performance coefficients
    Kt = prop_experimental.get_Kt(J)
    Kq = prop_experimental.get_Kq(J)
    efficiency = prop_experimental.get_efficiency(J)

    print(f"{speed_knots:<15.1f} {J:<10.4f} {Kt:<12.6f} "
          f"{Kq:<12.6f} {efficiency:<12.4f}")

print("-" * 75)

# Find optimal operating speed for maximum efficiency
efficiency_curve = [prop_experimental.get_efficiency(
    prop_experimental.calculate_J(speed * 0.5144, n_fixed))
    for speed in range(8, 25)]

max_eff_idx = np.argmax(efficiency_curve)
optimal_speed_knots = 8 + max_eff_idx
optimal_efficiency = efficiency_curve[max_eff_idx]

```

```
print(f"Optimal operating speed: {optimal_speed_knots} knots")
print(f"Maximum efficiency: {optimal_efficiency:.4f}")
```

Listing 1.44: Multi-Speed Performance Analysis

Use Case 3: Controllable-Pitch Propeller Basic Operations

This example demonstrates fundamental CPP operations, including pitch optimization for specific thrust requirements and multi-dimensional performance analysis.

CPP System Initialization Set up controllable-pitch propeller with comprehensive performance grid:

```
# Create CPP propeller parameters
params_cpp = ShipPropellerParameters(
    diameter=5.0,                    # 5.0 meters diameter
    blade_number=4,                  # 4 blades
    h=0.5,                          # 0.5 meters water height
    type=PropellerType.CPP,
    area_ratio=0.75,                # Expanded blade area ratio
    screws=1,                       # Single screw configuration
    name="MainEngine_CPP"
)

# Create propeller with CPP capabilities
prop_cpp = ShipPropeller(
    parameters=params_cpp,
    wageningen=True,                # Enable Wageningen calculations
    verbose=True
)

print(f"CPP Propeller: {prop_cpp.parameters.name}")
print(f"Performance grid shape: {prop_cpp.cpp_calculator.kt_array.shape}")
print(f"Pitch range: {prop_cpp.cpp_calculator.pitch_values[0]:.2f} - "
      f"{prop_cpp.cpp_calculator.pitch_values[-1]:.2f}")

# Validate data quality
print("\nData Quality Assessment:")
prop_cpp.cpp_calculator.validate_coefficients()
quality_summary = prop_cpp.cpp_calculator.get_data_quality_summary()
print("Quality report generated successfully")
```

Listing 1.45: CPP System Setup and Validation

Pitch Optimization for Target Thrust Find optimal pitch settings for specific operational requirements:

```
# Define operating conditions
Va = 6.0            # Ship speed [m/s]
n = 1.8             # Propeller rotation speed [rps] (108 RPM)
required_thrust = 120000  # Required thrust [N] (120 kN)

# Calculate advance coefficient
J = prop_cpp.calculate_J(Va, n)
print(f"Operating condition:")
print(f"  Speed: {Va} m/s")
print(f"  RPM: {n*60:.0f}")
print(f"  Advance coefficient J: {J:.4f}")
print(f"  Required thrust: {required_thrust/1000:.1f} kN")

# Find optimal pitch setting
```

```
performance = prop_cpp.get_cpp_performance(
    j_value=J,
    required_thrust=required_thrust,
    n=n
)

print(f"\nOptimal CPP Configuration:")
print(f"  Optimal P/D ratio: {performance['P_D']:.4f}")
print(f"  Thrust coefficient Kt: {performance['Kt']:.6f}")
print(f"  Torque coefficient Kq: {performance['Kq']:.6f}")
print(f"  Efficiency      : {performance['efficiency']:.4f}")
print(f"  Generated thrust: {performance['thrust']/1000:.2f} kN")
print(f"  Required torque: {performance['torque']/1000:.2f}  k N m ")
print(f"  Power requirement: {performance['power']/1000:.2f} kW")
```

Listing 1.46: CPP Pitch Optimization

Multi-Condition CPP Analysis Analyze CPP performance across various operational scenarios:

```
# Define operational envelope
operating_conditions = [
    (4.0, 1.5, 80000),   # Low speed, low RPM, moderate thrust
    (6.0, 1.8, 120000),  # Medium speed, medium RPM, high thrust
    (8.0, 2.1, 160000),  # High speed, high RPM, very high thrust
]

print("CPP Performance Across Operating Envelope:")
print("-" * 90)
print(f"{'Speed (m/s)':<12} {'RPM':<8} {'Thrust (kN)':<12} {'J':<8} "
      f"{'P/D':<8} {'Efficiency':<12} {'Power (kW)':<12}")
print("-" * 90)

for Va, n, thrust in operating_conditions:
    # Calculate advance coefficient
    J = prop_cpp.calculate_J(Va, n)

    # Get optimal CPP performance
    perf = prop_cpp.get_cpp_performance(
        j_value=J,
        required_thrust=thrust,
        n=n
    )

    print(f"{Va:<12.1f} {n*60:<8.1f} {thrust/1000:<12.1f} {J:<8.3f} "
          f"{perf['P_D']:<8.3f} {perf['efficiency']:<12.3f} "
          f"{perf['power']/1000:<12.1f}")

print("-" * 90)

# Generate 3D performance surface visualization
cpp_surfaces = prop_cpp.plot_cpp_surfaces()
plt.suptitle('CPP Performance Surfaces', fontsize=16)
plt.show()
```

Listing 1.47: Multi-Condition CPP Performance

Use Case 4: Advanced CPP Optimization

This comprehensive example demonstrates advanced CPP optimization capabilities, including simultaneous RPM and pitch optimization for maximum efficiency.

Single-Point Optimization Find optimal RPM and pitch combination for specific operational requirements:

```
# Create larger CPP for optimization example
params_large_cpp = ShipPropellerParameters(
    diameter=4.5,                    # 4.5 meters diameter
    blade_number=4,                  # 4 blades
    h=3.0,                          # 3.0 meters water height
    type=PropellerType.CPP,
    area_ratio=0.65,                # Expanded blade area ratio
    screws=1,                       # Single screw configuration
    name="OptimizationTest_CPP"
)

# Create propeller for optimization
prop_opt = ShipPropeller(
    parameters=params_large_cpp,
    wageningen=True,
    verbose=True
)

# Define optimization scenario
Va_target = 7.5          # Ship speed [m/s]
thrust_target = 150000  # Required thrust [N] (150 kN)

print(f"CPP Optimization Scenario:")
print(f"  Ship speed: {Va_target} m/s")
print(f"  Required thrust: {thrust_target/1000:.1f} kN")
print(f"  Propeller diameter: {prop_opt.parameters.D} m")

# Perform comprehensive optimization
result = prop_opt.cpp_optimise_rpm_pd(
    required_thrust=thrust_target,
    speed_of_advance=Va_target,
    refined_search=True
)

print(f"\nOptimal Operating Point:")
print(f"  Optimal RPM: {result['rpm']:.1f}")
print(f"  Optimal P/D ratio: {result['P_D']:.4f}")
print(f"  Maximum efficiency: {result['efficiency']:.4f}")
print(f"  Power requirement: {result['power_kw']:.1f} kW")
print(f"  Advance coefficient J: {result['J']:.4f}")
print(f"  Thrust coefficient Kt: {result['Kt']:.6f}")
print(f"  Torque coefficient Kq: {result['Kq']:.6f}")
```

Listing 1.48: Advanced CPP Single-Point Optimization

Optimization vs Fixed Operation Comparison Compare optimized operation against fixed RPM scenarios:

```
# Compare with fixed RPM operation
fixed_rpm = 180  # Fixed RPM for comparison
fixed_n = fixed_rpm / 60  # Convert to rps
J_fixed = prop_opt.calculate_J(Va_target, fixed_n)

# Get performance at fixed RPM
fixed_performance = prop_opt.get_cpp_performance(
    j_value=J_fixed,
    required_thrust=thrust_target,
    n=fixed_n
)

```

```
print(f"\nOptimization Benefits Analysis:")
print("-" * 65)
print(f"{'Parameter':<20} {'Optimized':<15} {'Fixed RPM':<15} {'Benefit':<15}")
print("-" * 65)

# Calculate improvements
rpm_difference = result['rpm'] - fixed_rpm
efficiency_improvement = ((result['efficiency'] - fixed_performance['efficiency'])
                          / fixed_performance['efficiency'] * 100)
power_saving = fixed_performance['power']/1000 - result['power_kw']

print(f"{'RPM':<20} {result['rpm']:<15.1f} {fixed_rpm:<15.1f} "
      f"{rpm_difference:<15.1f}")
print(f"{'P/D Ratio':<20} {result['P_D']:<15.4f} "
      f"{fixed_performance['P_D']:<15.4f} "
      f"{result['P_D'] - fixed_performance['P_D']:<15.4f}")
print(f"{'Efficiency':<20} {result['efficiency']:<15.4f} "
      f"{fixed_performance['efficiency']:<15.4f} "
      f"{efficiency_improvement:<15.2f}%")
print(f"{'Power (kW)':<20} {result['power_kw']:<15.1f} "
      f"{fixed_performance['power']/1000:<15.1f} {power_saving:<15.1f}")

print("-" * 65)
print(f"Summary: {efficiency_improvement:.1f}% efficiency gain, "
      f"{power_saving:.1f} kW power saving")
```

Listing 1.49: Optimization Benefit Analysis

Multi-Condition Optimization Study Comprehensive analysis across complete operational envelope:

```
# Define comprehensive operational conditions
operating_envelope = [
    (5.0, 80000),    # Low speed, moderate thrust
    (6.0, 100000),   # Medium speed, medium thrust
    (7.0, 120000),   # Medium-high speed, high thrust
    (8.0, 140000),   # High speed, very high thrust
    (9.0, 180000),   # Maximum speed, maximum thrust
    (4.0, 120000),   # Low speed, high thrust (bollard pull)
]

print("Comprehensive Operational Envelope Analysis:")
print("-" * 100)
print(f"{'Speed (m/s)':<12} {'Thrust (kN)':<12} {'Opt RPM':<10} "
      f"{'Opt P/D':<10} {'Efficiency':<12} {'Power (kW)':<12} {'J':<8}")
print("-" * 100)

envelope_results = []

for Va, thrust in operating_envelope:
    try:
        # Optimize for each condition
        opt_result = prop_opt.cpp_optimise_rpm_pd(
            required_thrust=thrust,
            speed_of_advance=Va,
            refined_search=True
        )

        envelope_results.append({
            'speed': Va,
            'thrust': thrust,
            'rpm': opt_result['rpm'],
```

```
            'pd_ratio': opt_result['P_D'],
            'efficiency': opt_result['efficiency'],
            'power': opt_result['power_kw'],
            'J': opt_result['J']
        })

        print(f"{Va:<12.1f} {thrust/1000:<12.1f} {opt_result['rpm']:<10.0f} "
              f"{opt_result['P_D']:<10.3f} {opt_result['efficiency']:<12.3f} "
              f"{opt_result['power_kw']:<12.0f} {opt_result['J']:<8.3f}")

    except Exception as e:
        print(f"{Va:<12.1f} {thrust/1000:<12.1f} "
              f"{'OPTIMIZATION FAILED':<50}")
        print(f"  Error: {str(e)}")

print("-" * 100)

# Analysis summary
if envelope_results:
    avg_efficiency = np.mean([r['efficiency'] for r in envelope_results])
    max_efficiency = max([r['efficiency'] for r in envelope_results])
    efficiency_range = max_efficiency - min([r['efficiency'] for r in
    envelope_results])

    print(f"\nOperational Envelope Summary:")
    print(f"  Average efficiency: {avg_efficiency:.3f}")
    print(f"  Maximum efficiency: {max_efficiency:.3f}")
    print(f"  Efficiency range: {efficiency_range:.3f}")
    print(f"  Successfully analyzed: {len(envelope_results)}/{len(
    operating_envelope)} conditions")
```

Listing 1.50: Operational Envelope Optimization

Engine-Constrained Optimization Demonstrate practical optimization within engine operational limits:

```
# Engine-constrained optimization for realistic marine applications
engine_rated_rpm = 1800    # Engine rated RPM
rpm_tolerance = 0.15       #  15 % tolerance

test_conditions = [
    (6.5, 110000),  # Medium speed cruise
    (8.0, 140000),  # High speed operation
]

print("Engine-Constrained Optimization Analysis:")
print(f"Engine rated RPM: {engine_rated_rpm}")
print(f"Operating tolerance:   {rpm_tolerance*100:.1f}%")
print(f"Acceptable RPM range: {engine_rated_rpm*(1-rpm_tolerance):.0f} - "
      f"{engine_rated_rpm*(1+rpm_tolerance):.0f}")

for Va, thrust in test_conditions:
    print(f"\nCondition: {Va} m/s, {thrust/1000:.1f} kN")

    # Unconstrained optimization
    unconstrained = prop_opt.cpp_optimise_rpm_pd(
        required_thrust=thrust,
        speed_of_advance=Va,
        refined_search=True
    )

    # Engine-constrained optimization
    constrained = prop_opt.cpp_optimise_rpm_pd_engine_constrained(
```

```
        required_thrust=thrust,
        speed_of_advance=Va,
        engine_rated_rpm=engine_rated_rpm,
        rpm_tolerance=rpm_tolerance
    )

    print(f"  Unconstrained optimal: {unconstrained['rpm']:.0f} RPM, "
          f"  ={unconstrained['efficiency']:.3f}")
    print(f"  Engine-constrained: {constrained['rpm']:.0f} RPM, "
          f"  ={constrained['efficiency']:.3f}")
    print(f"  Engine loading: {constrained['engine_loading_percent']:.1f}%")
    print(f"  RPM deviation: {constrained['rpm_deviation']:.0f} RPM")

    efficiency_penalty = unconstrained['efficiency'] - constrained['efficiency']
    print(f"  Efficiency penalty: {efficiency_penalty:.4f} "
          f"({efficiency_penalty/unconstrained['efficiency']*100:.1f}%)")
```

Listing 1.51: Engine-Constrained CPP Optimization

Implementation Guidelines

Performance Considerations When implementing these use cases in practice, consider the following:

- For preliminary design, Wageningen B-series provides adequate accuracy
- Experimental data should be used when available for final design validation
- CPP optimization requires careful consideration of engine constraints
- Data quality assessment is crucial for reliable CPP performance prediction
- Caching mechanisms improve computational efficiency for repeated analyses

Validation and Verification Ensure reliable results through systematic validation:

- Compare Wageningen predictions with experimental data when available
- Validate optimization results against physical constraints
- Check data quality reports for CPP calculations
- Verify that operating points fall within propeller design limits
- Cross-reference results with established design practices

Note

These examples provide templates for common propeller analysis tasks. Modify the parameters, operating conditions, and analysis approaches to suit specific design requirements and constraints.

Warning

CPP optimization results should be validated against actual engine characteristics and operational constraints. The examples use representative values that may not reflect specific engine limitations or operational requirements.

These practical examples demonstrate the comprehensive capabilities of the propeller analysis framework, from basic performance prediction to advanced multi-variable optimization. The modular design allows designers to select appropriate analysis approaches based on available data, design phase requirements, and operational constraints.