OptiX

Database-Oriented Optimization Library

OptiX Mathematical Optimization Framework Documentation

Release 1.0.0

Tolga BERBER, Beyzanur SİYAH

Aug 04, 2025

Getting Started

OptiX

Database-Oriented
Optimization Library

Welcome to **OptiX**, a comprehensive Python framework for mathematical optimization problems, supporting linear programming (LP), goal programming (GP), and constraint satisfaction problems (CSP). Built with multi-solver architecture and advanced constraint modeling capabilities.

Chapter 1

System Overview

OptiX provides a hierarchical problem-solving framework with increasing complexity, designed to handle real-world optimization challenges across diverse domains including operations research, supply chain management, resource allocation, and decision support systems.

1.1 Core Architecture

1.2 Key Features

Modeling Capabilities:

- ▯ **Flexible Modeling**: Create decision variables, constraints, and objective functions with intuitive APIs
- ▯ **Special Constraints**: Non-linear operations including multiplication (×), division (÷), modulo (mod), and conditional (if-then) logic
- ▯ **Database Integration**: Object-relational mapping for complex data structures with automatic variable generation
- ▯ **Scenario Management**: Built-in support for multi-scenario optimization and sensitivity analysis

Solver Integration:

- ▯ **Multi-Solver Architecture**: Unified interface supporting OR-Tools (open-source) and Gurobi (commercial)
- ▯ **Performance Optimization**: Efficient problem setup, constraint translation, and solution extraction
- ▯ **Extensible Design**: Easy integration of custom solvers through standardized interfaces
- ▯ **Parallel Solving**: Support for concurrent solver execution and performance comparison

Real-World Applications:

- ▯ **Operations Research**: Supply chain optimization, resource allocation, scheduling problems
- ▯ **Production Planning**: Manufacturing optimization, inventory management, capacity planning
- ▯ **Transportation**: Route optimization, vehicle assignment, logistics planning
- ▯ **Financial Modeling**: Portfolio optimization, risk management, investment planning

Chapter 2

Quick Start

Install OptiX using Poetry:

```
# Clone the repository
git clone https://github.com/yourusername/optix.git
cd OptiX

# Install dependencies
poetry install

# Activate virtual environment
poetry shell
```

Create your first optimization problem:

```
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from solvers import solve

# Create a Linear Programming problem
problem = OXLPProblem()

# Add decision variables
problem.create_decision_variable("x1", "Variable 1", 0, 10)
problem.create_decision_variable("x2", "Variable 2", 0, 15)

# Add constraints: 2x1 + 3x2 <= 20
problem.create_constraint(
    variables=[var.id for var in problem.variables.search_by_function(lambda x:
 ↪x.name in ["x1", "x2"])],
    weights=[2, 3],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=20
)

# Set objective: maximize 5x1 + 4x2
```

(continues on next page)

(continued from previous page)

```
problem.create_objective_function(
    variables=[var.id for var in problem.variables.search_by_function(lambda x:
↪x.name in ["x1", "x2"])],
    weights=[5, 4],
    objective_type=ObjectiveType.MAXIMIZE
)

# Solve the problem
status, solution = solve(problem, 'ORTools')
```

Chapter 3

Problem Types

OptiX supports three main problem types with increasing complexity:

Constraint Satisfaction Problems (CSP)

Focus on finding feasible solutions that satisfy all constraints without optimization.

```
from problem import OXCSPProblem

csp = OXCSPProblem()
# Variables and constraints only
# Focus on finding feasible solutions
```

Linear Programming (LP)

Extends CSP with objective function optimization for single-objective problems.

```
from problem import OXLPProblem, ObjectiveType

lp = OXLPProblem()
# CSP + objective function optimization
# Single objective optimization (minimize/maximize)
```

Goal Programming (GP)

Extends LP with multi-objective goal constraints and deviation variables.

```
from problem import OXGPProblem

gp = OXGPProblem()
# LP + multi-objective goal constraints with deviation variables
# Handle conflicting objectives with priority levels
```

Chapter 4

Special Constraints

OptiX supports advanced constraint types for non-linear operations that standard linear programming solvers cannot handle directly:

```python
from problem import OXLPProblem, SpecialConstraintType

problem = OXLPProblem()

# Create variables
problem.create_decision_variable("x", "Variable X", 0, 100)
problem.create_decision_variable("y", "Variable Y", 0, 100)
problem.create_decision_variable("result", "Result Variable", 0, 10000)

# Create special constraint: x * y = result
problem.create_special_constraint(
    constraint_type=SpecialConstraintType.MULTIPLICATION,
    left_variable_id=problem.variables.search_by_name("x")[0].id,
    right_variable_id=problem.variables.search_by_name("y")[0].id,
    result_variable_id=problem.variables.search_by_name("result")[0].id
)
```

Chapter 5

Supported Solvers

OR-Tools (Google)

- **Type**: Open-source optimization suite
- **Strengths**: Fast, reliable, comprehensive algorithm support
- **Installation**: Automatic with OptiX
- **License**: Apache 2.0

Gurobi (Commercial)

- **Type**: Commercial optimization solver
- **Strengths**: High performance, advanced features, excellent support
- **Installation**: Requires separate license and installation
- **License**: Commercial (free academic licenses available)

Extensible Architecture

- **Custom Solvers**: Easy to add new solvers through `OXSolverInterface`
- **Unified API**: Same code works with different solvers
- **Solver Factory**: Automatic solver selection and configuration

Chapter 6

System Requirements

Software Requirements:

- **Python**: 3.12 or higher
- **Poetry**: 1.4 or higher (for dependency management)
- **OR-Tools**: 9.0 or higher (Google's optimization library)
- **Gurobi**: 10.0 or higher (optional, commercial solver)

Hardware Requirements:

- **CPU**: Multi-core processor (recommended for complex optimization problems)
- **RAM**: Minimum 4GB (8GB recommended for large-scale problems)
- **Storage**: 1GB for framework and examples

Chapter 7

Example Problems

OptiX includes comprehensive real-world examples that demonstrate the framework's capabilities:

Bus Assignment Problem (Goal Programming)

Located in `samples/bus_assignment_problem/`, this example demonstrates:

- **Goal Programming Implementation**: Multi-objective optimization with conflicting goals
- **Database Integration**: Custom data classes for buses, routes, and schedules
- **Variable Creation**: Dynamic variable generation from database objects using Cartesian products
- **Complex Constraints**: Fleet limitations, service requirements, and operational restrictions
- **Solution Analysis**: Detailed reporting and goal deviation analysis

```
# Run the basic bus assignment problem
poetry run python samples/bus_assignment_problem/01_simple_bus_assignment_
↪problem.py

# Run the advanced version with comprehensive features
poetry run python samples/bus_assignment_problem/03_bus_assignment_problem.py
```

Diet Problem (Classic Linear Programming)

Located in `samples/diet_problem/01_diet_problem.py`, this classic example showcases:

- **Historical Context**: Implementation of Stigler's 1945 diet optimization problem
- **Cost Minimization**: Finding the cheapest combination of foods meeting nutritional requirements
- **Nutritional Constraints**: Minimum and maximum nutrient requirements
- **Practical Limitations**: Volume constraints and reasonable food quantity bounds

```
# Run the diet problem example
poetry run python samples/diet_problem/01_diet_problem.py
```

Mathematical Formulation:

$$\text{Minimize: } \sum_i \text{cost}_i \times \text{quantity}_i$$

$$\text{Subject to:}$$

$$\sum_i \text{nutrient}_{ij} \times \text{quantity}_i \geq \text{min\_requirement}_j \quad \forall j$$

$$\sum_i \text{volume}_i \times \text{quantity}_i \leq \text{max\_volume}$$

$$\text{quantity}_i \geq 0 \quad \forall i$$

Chapter 8

Performance Guidelines

Optimization Tips:

- Use appropriate variable bounds to reduce search space
- Simplify constraints when possible (e.g., use == instead of <= and >= when appropriate)
- Choose the right solver: OR-Tools for routing/scheduling, Gurobi for large-scale linear/quadratic problems
- Use special constraints judiciously as they can significantly increase solve time

Chapter 9

Documentation Structure

9.1 Installation

This guide will help you install OptiX and its dependencies on your system.

9.1.1 System Requirements

Python Requirements

OptiX requires Python 3.12 or higher. Check your Python version:

```
python --version
```

If you need to install or upgrade Python, visit the official Python website[1].

Poetry Installation

OptiX uses Poetry for dependency management. Install Poetry:

macOS/Linux

```
curl -sSL https://install.python-poetry.org | python3 -
```

Windows (PowerShell)

```
(Invoke-WebRequest -Uri https://install.python-poetry.org -UseBasicParsing).
↪Content | python -
```

Alternative (pip)

```
pip install poetry
```

Verify Poetry installation:

```
poetry --version
```

[1] https://www.python.org/downloads/

Hardware Recommendations

Component	Minimum	Recommended
CPU	Dual-core processor	Quad-core or higher
RAM	4GB	8GB or more
Storage	1GB free space	5GB+ for development
Network	Internet connection for installation	Stable connection for updates

9.1.2 Installing OptiX

Clone the Repository

```
# Clone from GitHub
git clone https://github.com/yourusername/optix.git
cd OptiX
```

Install Dependencies

```
# Install all dependencies including development tools
poetry install

# Install only production dependencies
poetry install --no-dev
```

Activate Virtual Environment

```
# Activate the Poetry virtual environment
poetry shell

# Or run commands with Poetry
poetry run python your_script.py
```

9.1.3 Solver Installation

OptiX supports multiple optimization solvers. Install the ones you need:

OR-Tools (Recommended)

OR-Tools is automatically installed with OptiX dependencies.

```
# Verify OR-Tools installation
poetry run python -c "import ortools; print('OR-Tools version:', ortools.__
↪version__)"
```

Note

OR-Tools is free and open-source, making it the recommended solver for getting started.

Gurobi (Commercial)

Download and Install Gurobi

1. Visit Gurobi Downloads[2]
2. Create a free account
3. Download the appropriate version for your platform
4. Follow the installation instructions for your operating system

Get a License

Academic License (Free)

1. Visit Gurobi Academic Licenses[3]
2. Register with your academic email
3. Download the license file
4. Follow activation instructions

Commercial License

Contact Gurobi sales for commercial licensing options.

Set Environment Variables

Linux/macOS

Add to your `.bashrc` or `.zshrc`:

```
export GUROBI_HOME="/opt/gurobi1000/linux64"
export PATH="${PATH}:${GUROBI_HOME}/bin"
export LD_LIBRARY_PATH="${LD_LIBRARY_PATH}:${GUROBI_HOME}/lib"
```

Windows

Set environment variables in System Properties:

```
GUROBI_HOME=C:\gurobi1000\win64
PATH=%PATH%;%GUROBI_HOME%\bin
LD_LIBRARY_PATH=%LD_LIBRARY_PATH%;%GUROBI_HOME%\lib
```

Install Python Interface

```
# Install Gurobi Python package
poetry run pip install gurobipy

# Verify installation
poetry run python -c "import gurobipy; print('Gurobi installed successfully')"
```

[2] https://www.gurobi.com/downloads/
[3] https://www.gurobi.com/downloads/licenses/

9.1.4 Alternative Installation Methods

Using pip (Not Recommended)

If you prefer pip over Poetry:

```
# Create virtual environment
python -m venv venv

# Activate virtual environment
# On Windows:
venv\Scripts\activate
# On macOS/Linux:
source venv/bin/activate

# Install dependencies (if requirements.txt exists)
pip install -r requirements.txt
```

Development Installation

For contributing to OptiX development:

```
# Clone the repository
git clone https://github.com/yourusername/optix.git
cd OptiX

# Install with development dependencies
poetry install --with dev,test,docs

# Install pre-commit hooks
poetry run pre-commit install

# Run tests to verify installation
poetry run pytest
```

9.1.5 Verification

Test your installation with this simple script:

```
# test_installation.py
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from solvers import solve, get_available_solvers

def test_installation():
    print("=== OptiX Installation Test ===")

    # Check available solvers
    solvers = get_available_solvers()
    print(f"Available solvers: {solvers}")

```

(continues on next page)

(continued from previous page)

```
    # Create a simple problem
    problem = OXLPProblem()
    problem.create_decision_variable(”x”, ”Test variable”, 0, 10)

    problem.create_constraint(
        variables=[problem.variables[0].id],
        weights=[1],
        operator=RelationalOperators.LESS_THAN_EQUAL,
        value=5
    )

    problem.create_objective_function(
        variables=[problem.variables[0].id],
        weights=[1],
        objective_type=ObjectiveType.MAXIMIZE
    )

    # Test solving
    for solver in solvers:
        try:
            status, solution = solve(problem, solver)
            print(f”▯  {solver}: {status}”)
            if solution:
                print(f”   Objective value: {solution[0].objective_value}”)
        except Exception as e:
            print(f”▯  {solver}: {e}”)

    print(”\n▯  Installation test completed!”)

if __name__ == ”__main__”:
    test_installation()
```

Run the test:

```
poetry run python test_installation.py
```

9.1.6 Troubleshooting

Common Issues

Poetry not found

```
# Add Poetry to PATH (macOS/Linux)
export PATH=”$HOME/.local/bin:$PATH”

# Restart your terminal and try again
```

OR-Tools import error

```
# Reinstall OR-Tools
poetry run pip uninstall ortools-python
poetry install --force
```

Gurobi license error

```
# Check license status
grbgetkey your-license-key

# Verify license file location
echo $GRB_LICENSE_FILE
```

Permission errors (Linux/macOS)

```
# Fix permissions for Poetry installation
sudo chown -R $(whoami) ~/.local/share/pypoetry
```

Getting Help

If you encounter issues:

1. Check the GitHub Issues[4] page
2. Review the ../development/troubleshooting section
3. Join our community discussions
4. Contact the development team

Tip

Quick Start: Once installed, head to the *Quick Start Guide* (page 24) guide to create your first optimization problem!

Note

Performance Note: For large-scale problems, consider installing Gurobi for better performance, especially for mixed-integer programming problems.

9.2 Quick Start Guide

This guide will get you up and running with OptiX in just a few minutes. We'll walk through creating and solving your first optimization problem step by step.

Note

Before starting, make sure you have completed the *Installation* (page 19) process.

[4] https://github.com/yourusername/optix/issues

9.2.1 Your First Optimization Problem

Let's solve a simple production planning problem:

Scenario: A factory produces two products (A and B). We want to maximize profit while respecting resource constraints.

Step 1: Import Required Modules

```
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from solvers import solve
```

Step 2: Create the Problem

```
# Create a Linear Programming problem
problem = OXLPProblem()
```

Step 3: Define Decision Variables

```
# Production quantities (units per day)
problem.create_decision_variable(
    var_name="product_A",
    description="Daily production of Product A",
    lower_bound=0,
    upper_bound=1000
)

problem.create_decision_variable(
    var_name="product_B",
    description="Daily production of Product B",
    lower_bound=0,
    upper_bound=1000
)
```

Step 4: Add Constraints

```
# Resource constraint: 2A + 3B <= 1200 (machine hours)
problem.create_constraint(
    variables=[var.id for var in problem.variables],
    weights=[2, 3],  # Hours per unit
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=1200,  # Available hours
    description="Machine hours constraint"
)

# Material constraint: 1A + 2B <= 800 (kg of material)
problem.create_constraint(
    variables=[var.id for var in problem.variables],
    weights=[1, 2],  # Material per unit
```

(continues on next page)

(continued from previous page)

```
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=800,  # Available material
    description=”Material constraint”
)
```

Step 5: Set Objective Function

```
# Maximize profit: 50A + 40B (profit per unit)
problem.create_objective_function(
    variables=[var.id for var in problem.variables],
    weights=[50, 40],  # Profit per unit
    objective_type=ObjectiveType.MAXIMIZE
)
```

Step 6: Solve the Problem

```
# Solve using OR-Tools
status, solution = solve(problem, 'ORTools')

# Check if solution was found
if solution and solution[0].objective_value is not None:
    print(f”  Optimization Status: {status}”)
    print(f”  Maximum Profit: ${solution[0].objective_value:.2f}”)

    # Display variable values
    for variable in problem.variables:
        value = solution[0].variable_values.get(variable.id, 0)
        print(f”  {variable.description}: {value:.2f} units”)
else:
    print(”  No optimal solution found”)
```

Complete Example Code

Here’s the complete code you can copy and run:

Listing 1: quickstart_example.py

```
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from solvers import solve

def solve_production_problem():
    ”””Solve a simple production planning problem.”””

    # Create problem
    problem = OXLPProblem()

    # Add variables
    problem.create_decision_variable(”product_A”, ”Daily production of Product A
```

(continues on next page)

(continued from previous page)

```
↪", 0, 1000)
    problem.create_decision_variable("product_B", "Daily production of Product B
↪", 0, 1000)

    # Add constraints
    problem.create_constraint(
        variables=[var.id for var in problem.variables],
        weights=[2, 3],
        operator=RelationalOperators.LESS_THAN_EQUAL,
        value=1200,
        description="Machine hours constraint"
    )

    problem.create_constraint(
        variables=[var.id for var in problem.variables],
        weights=[1, 2],
        operator=RelationalOperators.LESS_THAN_EQUAL,
        value=800,
        description="Material constraint"
    )

    # Set objective
    problem.create_objective_function(
        variables=[var.id for var in problem.variables],
        weights=[50, 40],
        objective_type=ObjectiveType.MAXIMIZE
    )

    # Solve
    status, solution = solve(problem, 'ORTools')

    # Display results
    if solution and solution[0].objective_value is not None:
        print("  Production Planning Results")
        print("=" * 40)
        print(f"Status: {status}")
        print(f"Maximum Profit: ${solution[0].objective_value:.2f}")
        print()

        for variable in problem.variables:
            value = solution[0].variable_values.get(variable.id, 0)
            print(f"{variable.description}: {value:.2f} units")

    return problem, solution

if __name__ == "__main__":
    problem, solution = solve_production_problem()
```

Expected Output

When you run this example, you should see output similar to:

```
▯  Production Planning Results
========================================
Status: OPTIMAL
Maximum Profit: $26666.67

Daily production of Product A: 266.67 units
Daily production of Product B: 266.67 units
```

9.2.2 Understanding the Results

The optimizer found that producing approximately 267 units of each product daily maximizes profit at $26,667 while respecting both resource constraints.

9.2.3 Problem Types Overview

OptiX supports three main problem types:

CSP (Constraint Satisfaction)

Find any solution that satisfies all constraints:

```
from problem import OXCSPProblem

csp = OXCSPProblem()
# Add variables and constraints
# No objective function needed
```

LP (Linear Programming)

Optimize a linear objective subject to linear constraints:

```
from problem import OXLPProblem, ObjectiveType

lp = OXLPProblem()
# Add variables, constraints, and objective function
```

GP (Goal Programming)

Handle multiple conflicting objectives:

```
from problem import OXGPProblem

gp = OXGPProblem()
# Add variables, constraints, and goal constraints
gp.create_goal_constraint(variables, weights, target_value, description)
```

9.2.4 Solver Selection

OptiX supports multiple solvers:

```
from solvers import solve, get_available_solvers

# Check available solvers
available = get_available_solvers()
print(f"Available solvers: {available}")

# Solve with specific solver
status, solution = solve(problem, 'ORTools')
status, solution = solve(problem, 'Gurobi')  # If installed

# Let OptiX choose the best available solver
status, solution = solve(problem)
```

9.2.5 Common Patterns

Variable Creation from Data

```
from data import OXData, OXDatabase

# Create data objects
products = OXDatabase([
    OXData(name="Product_A", cost=10, capacity=500),
    OXData(name="Product_B", cost=15, capacity=300)
])

# Create variables from data
for product in products:
    problem.create_decision_variable(
        var_name=f"production_{product.name}",
        description=f"Production of {product.name}",
        lower_bound=0,
        upper_bound=product.capacity
    )
```

Constraint Patterns

```
# Capacity constraint
problem.create_constraint(
    variables=production_vars,
    weights=[1] * len(production_vars),
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=total_capacity
)

# Demand constraint
problem.create_constraint(
```

(continues on next page)

(continued from previous page)

```
    variables=[product_var.id],
    weights=[1],
    operator=RelationalOperators.GREATER_THAN_EQUAL,
    value=minimum_demand
)

# Balance constraint
problem.create_constraint(
    variables=[inflow_var.id, outflow_var.id],
    weights=[1, -1],
    operator=RelationalOperators.EQUAL,
    value=0
)
```

9.2.6 Debugging and Validation

Problem Validation

```
def validate_problem(problem):
    """Basic problem validation."""

    issues = []

    # Check variables
    if not problem.variables:
        issues.append("No variables defined")

    # Check constraints
    if not problem.constraints:
        issues.append("No constraints defined")

    # Check objective (for LP/GP)
    if hasattr(problem, 'objective_function'):
        if not problem.objective_function:
            issues.append("No objective function defined")

    # Check variable bounds
    for var in problem.variables:
        if var.lower_bound > var.upper_bound:
            issues.append(f"Invalid bounds for {var.name}")

    return issues

# Usage
issues = validate_problem(problem)
if issues:
    print("Problem issues found:")
    for issue in issues:
        print(f"  - {issue}")
```

Solution Analysis

```
def analyze_solution(solution, problem):
    """Analyze optimization solution."""

    if not solution:
        print("No solution to analyze")
        return

    sol = solution[0]
    print(f"Objective Value: {sol.objective_value}")
    print(f"Solution Status: {sol.status}")

    # Check constraint satisfaction
    print("\nConstraint Analysis:")
    for i, constraint in enumerate(problem.constraints):
        lhs_value = sum(
            constraint.weights[j] * sol.variable_values.get(constraint.
↪variables[j], 0)
            for j in range(len(constraint.variables))
        )

        print(f"Constraint {i+1}: {lhs_value:.2f} {constraint.operator.name}
↪{constraint.value}")
```

9.2.7 Next Steps

Now that you've solved your first problem, explore these areas:

1. **Examples**: Try the *Classic Diet Problem* (page 34) and *Bus Assignment Problem (Goal Programming)* (page 43)
2. **Problem Types**: Learn about *Problem Types* (page 59)
3. **Advanced Features**: Explore user_guide/constraints and special constraints
4. **Solvers**: Configure user_guide/solvers for your needs

9.2.8 Common Issues and Solutions

Problem: Import Errors

```
# Solution: Ensure OptiX is properly installed
poetry install
poetry shell
```

Problem: No Solution Found

```
# Check if problem is feasible
if not solution:
    print("Problem may be infeasible or unbounded")
    # Review constraints and bounds
```

Problem: Unexpected Results

```
# Validate input data
print("Variable bounds:")
for var in problem.variables:
    print(f"  {var.name}: [{var.lower_bound}, {var.upper_bound}]")

print("Constraint details:")
for i, constraint in enumerate(problem.constraints):
    print(f"  Constraint {i}: {constraint.description}")
```

9.2.9 Getting Help

- **Documentation**: Comprehensive guides in this documentation
- **Examples**: Real-world examples in the `samples/` directory
- **API Reference**: Detailed API documentation for all modules
- **GitHub Issues**: Report bugs and request features

Tip

Pro Tip: Start with simple problems and gradually add complexity. Use the validation functions to catch issues early!

See also

- *Installation* (page 19) - Detailed installation instructions
- *Examples* (page 32) - More comprehensive examples
- api/index - Complete API reference

9.3 Examples

This section provides comprehensive, real-world examples demonstrating OptiX's capabilities across different optimization problem types and application domains.

9.3.1 Example Categories

By Problem Type

Linear Programming (LP)

- *Classic Diet Problem* (page 34) - Cost minimization with constraints
- production_planning - Resource allocation and planning
- Transportation and logistics examples

Goal Programming (GP)

- *Bus Assignment Problem (Goal Programming)* (page 43) - Multi-objective transportation planning
- Workforce planning with multiple criteria
- Project selection with competing goals

Constraint Satisfaction (CSP)

- Scheduling and timetabling problems
- Configuration and assignment problems
- Feasibility checking examples

By Application Domain

Transportation & Logistics

- *Bus Assignment Problem (Goal Programming)* (page 43) - Public transit optimization
- Vehicle routing and scheduling
- Supply chain optimization

Manufacturing & Production

- production_planning - Manufacturing optimization
- Inventory management
- Capacity planning

Finance & Investment

- portfolio_optimization - Investment allocation
- Risk management
- Capital budgeting

Healthcare & Resources

- *Classic Diet Problem* (page 34) - Nutritional planning
- Hospital resource allocation
- Treatment scheduling

By Complexity Level

Beginner □

- *Classic Diet Problem* (page 34) - Simple LP formulation
- Basic production planning
- Single-objective problems

Intermediate □

- *Bus Assignment Problem (Goal Programming)* (page 43) - Goal programming introduction
- Multi-constraint problems

- Database integration

Advanced ▯

- portfolio_optimization - Complex financial modeling
- Multi-period optimization
- Stochastic programming

9.3.2 Complete Example List

Classic Diet Problem

The Diet Problem is one of the foundational examples in linear programming and operations research. This tutorial demonstrates how to implement and solve Stigler's 1945 diet optimization problem using OptiX, specifically using a fast-food optimization scenario.

Note

This example is based on the complete implementation in `samples/diet_problem/01_diet_problem.py`.

Problem Background

The Diet Problem was first formulated by George Stigler in 1945 as part of his research on economic theory and nutritional planning. The question posed was: **"What is the cheapest combination of foods that will satisfy all nutritional requirements?"**

Historical Context

- **World War II Era**: Military and government agencies needed cost-effective nutrition programs
- **Stigler's Manual Solution**: Calculated by hand, cost $39.69 per year (1939 dollars)
- **Linear Programming Solution**: Later found optimal cost of $39.93, validating manual calculations
- **Modern Applications**: Supply chain management, resource allocation, dietary planning

Mathematical Formulation

Objective: Minimize total food cost

$$\text{Minimize: } \sum_i \text{cost}_i \times \text{quantity}_i$$

Subject to:

$$\sum_i \text{nutrient}_{ij} \times \text{quantity}_i \geq \text{min\_requirement}_j \quad \forall j \tag{9.1}$$

$$\sum_i \text{nutrient}_{ij} \times \text{quantity}_i \leq \text{max\_requirement}_j \quad \forall j \tag{9.2}$$

$$\sum_i \text{volume}_i \times \text{quantity}_i \leq \text{max\_volume} \tag{9.3}$$

$$\text{quantity}_i \geq 0 \quad \forall i \tag{9.4}$$

$$\text{quantity}_i \leq \text{reasonable\_limit}_i \quad \forall i \tag{9.5}$$

Implementation

Data Structures

First, let's define the data structures for foods and nutrients using custom dataclasses:

```python
from dataclasses import dataclass
from data import OXData

@dataclass
class Food(OXData):
    ”””
    Data model representing a food item in the diet optimization problem.

    Attributes:
        name (str): Human-readable identifier for the food item
        c (float): Cost per serving in dollars
        v (float): Volume per serving in standardized units
    ”””
    name: str = ””
    c: float = 0.0    # Cost per serving
    v: float = 0.0    # Volume per serving

@dataclass
class Nutrient(OXData):
    ”””
    Data model representing a nutritional requirement.

    Attributes:
        name (str): Nutrient identifier (e.g., ”Calories”, ”Protein”)
        n_min (float): Minimum required amount
        n_max (float | None): Optional maximum allowed amount
    ”””
    name: str = ””
    n_min: float = 0.0
    n_max: float | None = None
```

Problem Instance Data

This implementation uses a fast-food diet optimization problem with 9 food items and 7 nutrients:

```python
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators

def create_diet_problem():
    """Create and configure the diet optimization problem."""

    # Initialize linear programming problem
    dp = OXLPProblem()

    # Define food items with cost and volume attributes
    foods = [
        Food(name="Cheeseburger", c=1.84, v=4.0),
        Food(name="Ham Sandwich", c=2.19, v=7.5),
        Food(name="Hamburger", c=1.84, v=3.5),
        Food(name="Fish Sandwich", c=1.44, v=5.0),
        Food(name="Chicken Sandwich", c=2.29, v=7.3),
        Food(name="Fries", c=0.77, v=2.6),
        Food(name="Sausage Biscuit", c=1.29, v=4.1),
        Food(name="Lowfat Milk", c=0.60, v=8.0),
        Food(name="Orange Juice", c=0.72, v=12.0)
    ]

    # Add food objects to database
    for food in foods:
        dp.db.add_object(food)

    # Define nutritional requirements
    nutrients = [
        Nutrient(name="Cal", n_min=2000),                    # Calories
        Nutrient(name="Carbo", n_min=350, n_max=375),        # Carbohydrates (g)
        Nutrient(name="Protein", n_min=55),                  # Protein (g)
        Nutrient(name="VitA", n_min=100),                    # Vitamin A (% RDA)
        Nutrient(name="VitC", n_min=100),                    # Vitamin C (% RDA)
        Nutrient(name="Calc", n_min=100),                    # Calcium (% RDA)
        Nutrient(name="Iron", n_min=100)                     # Iron (% RDA)
    ]

    # Add nutrient objects to database
    for nutrient in nutrients:
        dp.db.add_object(nutrient)

    return dp, foods, nutrients
```

Nutritional Content Matrix

The nutritional content is organized as a matrix where rows represent foods and columns represent nutrients:

```
# Nutritional content matrix: foods (rows) × nutrients (columns)
# Columns: [Calories, Carbs, Protein, VitA, VitC, Calcium, Iron]
nutritional_matrix = [
    [510, 34, 28, 15, 6, 30, 20],    # Cheeseburger
    [370, 35, 24, 15, 10, 20, 20],   # Ham Sandwich
    [500, 42, 25, 6, 2, 25, 20],     # Hamburger
    [370, 38, 14, 2, 0, 15, 10],     # Fish Sandwich
    [400, 42, 31, 8, 15, 15, 8],     # Chicken Sandwich
    [220, 26, 3, 0, 15, 0, 2],       # Fries
    [345, 27, 15, 4, 0, 20, 15],     # Sausage Biscuit
    [110, 12, 9, 10, 4, 30, 0],      # Lowfat Milk
    [80, 20, 1, 2, 120, 2, 2]        # Orange Juice
]
```

Variable Generation

OptiX provides automatic variable generation from database objects:

```
def create_variables_and_constraints(dp, foods, nutrients, nutritional_matrix):
    """Generate decision variables and constraints."""

    # Generate decision variables automatically from food database objects
    dp.create_variables_from_db(
        Food,
        var_name_template="{food_name} to consume",
        var_description_template="Number of servings of {food_name} to consume",
        lower_bound=0,        # Non-negativity constraint
        upper_bound=2000      # Practical upper limit per food item
    )

    # Extract variable IDs for constraint creation
    variable_ids = [v.id for v in dp.variables.objects]

    return variable_ids
```

Adding Nutritional Constraints

```
def add_nutritional_constraints(dp, variable_ids, nutrients, nutritional_matrix):
    """Add nutritional requirement constraints."""

    # Generate nutritional constraints for each nutrient requirement
    for j, nutrient in enumerate(nutrients):
        # Extract nutritional content for current nutrient across all foods
        weights = [food_nutrients[j] for food_nutrients in nutritional_matrix]
```

(continues on next page)

(continued from previous page)

```
        # Create minimum nutrient requirement constraint
        dp.create_constraint(
            variables=variable_ids,
            weights=weights,
            operator=RelationalOperators.GREATER_THAN_EQUAL,
            value=nutrient.n_min,
        )

        # Create maximum nutrient constraint if upper limit is specified
        if nutrient.n_max is not None:
            dp.create_constraint(
                variables=variable_ids,
                weights=weights,
                operator=RelationalOperators.LESS_THAN_EQUAL,
                value=nutrient.n_max,
            )
```

Volume Constraint

```
def add_volume_constraint(dp, variable_ids, foods):
    """Add total volume constraint."""

    # Maximum total volume constraint (practical consumption limit)
    Vmax = 75

    dp.create_constraint(
        variables=variable_ids,
        weights=[f.v for f in foods],
        operator=RelationalOperators.LESS_THAN_EQUAL,
        value=Vmax,
    )
```

Setting Objective Function

```
def set_cost_objective(dp, variable_ids, foods):
    """Set the cost minimization objective."""

    # Define cost minimization objective function
    dp.create_objective_function(
        variables=variable_ids,
        weights=[f.c for f in foods],
        objective_type=ObjectiveType.MINIMIZE
    )
```

Complete Solution

```
from solvers import solve

def solve_diet_problem():
    """Solve the complete diet optimization problem."""

    # Create problem and setup data
    dp, foods, nutrients = create_diet_problem()

    # Nutritional content matrix
    nutritional_matrix = [
        [510, 34, 28, 15, 6, 30, 20],    # Cheeseburger
        [370, 35, 24, 15, 10, 20, 20],   # Ham Sandwich
        [500, 42, 25, 6, 2, 25, 20],     # Hamburger
        [370, 38, 14, 2, 0, 15, 10],     # Fish Sandwich
        [400, 42, 31, 8, 15, 15, 8],     # Chicken Sandwich
        [220, 26, 3, 0, 15, 0, 2],       # Fries
        [345, 27, 15, 4, 0, 20, 15],     # Sausage Biscuit
        [110, 12, 9, 10, 4, 30, 0],      # Lowfat Milk
        [80, 20, 1, 2, 120, 2, 2]        # Orange Juice
    ]

    # Create variables and constraints
    variable_ids = create_variables_and_constraints(dp, foods, nutrients,
↪nutritional_matrix)
    add_nutritional_constraints(dp, variable_ids, nutrients, nutritional_matrix)
    add_volume_constraint(dp, variable_ids, foods)
    set_cost_objective(dp, variable_ids, foods)

    # Solve the optimization problem using Gurobi solver
    print("Solving Diet Optimization Problem...")
    print("=" * 50)

    try:
        # Solve with integer programming for discrete servings
        status, solutions = solve(dp, 'Gurobi', use_continuous=False,
↪equalizeDenominators=True)

        print(f"Status: {status}")

        # Display detailed solution
        for solution in solutions:
            solution.print_solution_for(dp)

        return solutions[0] if solutions else None

    except Exception as e:
        print(f"  Solver error: {e}")
        return None
```

Solution Analysis

The solution will show optimal quantities for each food item that minimize cost while satisfying all constraints:

```
def analyze_solution_details(solution, dp, foods, nutritional_matrix):
    ”””Provide detailed analysis of the optimal solution.”””

    if not solution:
        print(”No solution to analyze”)
        return

    print(f”\n□  Optimal Daily Food Cost: ${solution.objective_value:.2f}”)
    print(”\n□  Optimal Food Quantities:”)
    print(”-” * 60)

    total_cost = 0
    total_volume = 0

    # Display food quantities with costs
    for i, var in enumerate(dp.variables.objects):
        quantity = solution.variable_values.get(var.id, 0)

        if quantity > 0.01:  # Only show significant quantities
            food = foods[i]
            cost = quantity * food.c
            volume = quantity * food.v

            total_cost += cost
            total_volume += volume

            print(f”{food.name:<20}: {quantity:>8.2f} servings ”
                  f”(${cost:>6.2f}, {volume:>6.1f} units)”)

    print(”-” * 60)
    print(f”{'Total':<20}: ${total_cost:>14.2f}, {total_volume:>6.1f} units”)
```

Advanced Features

Solver Configuration

```
# Different solver configurations
solvers_to_try = ['Gurobi', 'ORTools']

for solver_name in solvers_to_try:
    try:
        print(f”\nTrying {solver_name} solver...”)

        # Continuous variables (allow fractional servings)
        status, solution = solve(dp, solver_name, use_continuous=True)
```

(continues on next page)

(continued from previous page)

```
        # Integer variables (whole servings only)
        # status, solution = solve(dp, solver_name, use_continuous=False)

        if solution:
            print(f"□  {solver_name} found solution: ${solution[0].objective_
↪value:.2f}")
        else:
            print(f"□  {solver_name} failed")

    except Exception as e:
        print(f"□  {solver_name} error: {e}")
```

Problem Variations

```
def create_vegetarian_variant():
    """Create a vegetarian version by excluding meat items."""

    # Modify food list to exclude meat products
    vegetarian_foods = [
        Food(name="Fries", c=0.77, v=2.6),
        Food(name="Lowfat Milk", c=0.60, v=8.0),
        Food(name="Orange Juice", c=0.72, v=12.0),
        # Add more vegetarian options...
    ]

    # Use same constraint structure with modified food set
    # ... rest of problem setup

def create_budget_variant(max_budget=5.00):
    """Create a budget-constrained version."""

    dp, foods, nutrients = create_diet_problem()
    variable_ids = create_variables_and_constraints(dp, foods, nutrients,
↪nutritional_matrix)

    # Add budget constraint
    dp.create_constraint(
        variables=variable_ids,
        weights=[f.c for f in foods],
        operator=RelationalOperators.LESS_THAN_EQUAL,
        value=max_budget,
    )

    # Continue with normal setup...
```

Running the Complete Example

```
def main():
    ”””Run the complete diet problem example.”””

    print(”□  OptiX Diet Problem Optimization”)
    print(”=” * 50)
    print(”Fast-food diet optimization based on Stigler's 1945 research”)
    print(”Demonstrates cost minimization with nutritional constraints”)
    print()

    solution = solve_diet_problem()

    if solution:
        print(”\n□  Diet optimization completed successfully!”)
        print(f”Minimum daily cost: ${solution.objective_value:.2f}”)

        print(”\n□  Key Insights:”)
        print(”• Optimal diet focuses on cost-effective nutrient sources”)
        print(”• Fast-food items can meet nutritional requirements efficiently”)
        print(”• Volume constraints prevent unrealistic consumption patterns”)
        print(”• Integer constraints ensure practical serving sizes”)
    else:
        print(”□  Failed to find optimal diet solution”)

if __name__ == ”__main__”:
    main()
```

Expected Results

The optimization typically finds solutions with:

- **Daily Cost**: $4.50 - $6.00 (varies with food selection and constraints)
- **Primary Foods**: Cost-effective items like milk, fries, and sandwiches
- **Nutritional Balance**: All requirements met at minimum cost
- **Volume**: Within 75 units total consumption limit
- **Servings**: Integer values for practical implementation

Key Learning Points

1. **Linear Programming**: Classic example of LP optimization with real constraints
2. **Database-Driven Modeling**: Using OptiX's OXData system for structured problem setup
3. **Matrix-Based Constraints**: Efficient handling of nutritional content through matrices
4. **Multi-Constraint Problems**: Balancing cost, nutrition, and practical limitations
5. **Solver Integration**: Working with different optimization engines (Gurobi, OR-Tools)

Extensions

Try these modifications to explore further:

- **Meal Planning**: Separate breakfast, lunch, dinner with different constraints
- **Weekly Planning**: Optimize across multiple days with variety requirements
- **Nutritional Balance**: Add constraints for food group diversity
- **Stochastic Optimization**: Handle uncertain food prices and availability
- **Goal Programming**: Convert to multi-objective optimization with preference priorities

Implementation Details

Problem Size: 9 variables, 15+ constraints **Solving Time**: < 1 second **Memory Usage**: Minimal (< 10MB) **Scalability**: Methodology extends to larger food/nutrient sets

Tip

Next Steps: After mastering the diet problem, try the *Bus Assignment Problem (Goal Programming)* (page 43) example to learn Goal Programming techniques with the OptiX framework.

See also

- *Linear Programming Tutorial* (page 67) - LP theory and techniques
- *Problem Module* (page 81) - Problem class documentation
- *Data Module* (page 205) - Data modeling with OXData framework
- ../user_guide/constraints - Advanced constraint modeling

Bus Assignment Problem (Goal Programming)

The Bus Assignment Problem demonstrates advanced Goal Programming techniques with real-world transportation data. This example showcases multi-objective optimization for public transit systems, balancing cost efficiency, service quality, and operational constraints.

Note

This example is based on the complete implementation in `samples/bus_assignment_problem/03_bus_assignment_problem.py`.

Problem Background

Public transportation agencies face complex decisions when allocating buses to routes. They must balance multiple competing objectives:

- **Cost Minimization**: Reduce operational expenses
- **Service Quality**: Meet passenger demand and service standards

- **Fleet Utilization**: Efficiently use available bus resources
- **Operational Constraints**: Respect maintenance, driver, and route limitations

Historical Context

Bus assignment problems emerged during the rapid urbanization of the mid-20th century:

- **1960s-1970s**: Urban planning boom requiring systematic transit optimization
- **Vehicle Routing Problems**: First formulated by Dantzig and Ramser (1959)
- **Goal Programming**: Introduced by Charnes and Cooper (1961) for multi-objective optimization
- **Modern Applications**: Contemporary smart city initiatives and sustainable transportation

Problem Formulation

Decision Variables: Number of trips each bus group performs on each transit line

Goal Programming Formulation: - **Primary Goal**: Minimize deviations from fleet utilization targets - **Secondary Goals**: Service quality, cost efficiency, operational balance

Constraint Categories: 1. **Bus Group Restrictions**: Certain bus types banned from specific lines 2. **Minimum Service Requirements**: Each line must have adequate service 3. **Fleet Capacity Limits**: Cannot exceed available buses per group

Mathematical Model

Decision Variables:

$$x_{ij} = \text{number of trips bus group } i \text{ performs on line } j$$

Goal Constraints:

$$\sum_j x_{ij} + d_i^- - d_i^+ = T_i \quad \forall i$$

Where: - T_i is the target utilization for bus group i - d_i^+, d_i^- are positive and negative deviation variables

Objective Function:

$$\text{Minimize: } \sum_i w_i^+ d_i^+ + w_i^- d_i^-$$

Implementation

Data Structure Setup

```
from data import OXData, OXDatabase
from problem import OXGPProblem
from constraints import RelationalOperators
```

(continues on next page)

(continued from previous page)

```python
def create_bus_assignment_data():
    """Create comprehensive bus assignment data structure."""

    # Bus groups with operational characteristics
    bus_groups_data = [
        OXData(
            name="Standard_Buses",
            total_buses=25,
            capacity_per_bus=50,
            operating_cost_per_trip=45.0,
            maintenance_factor=1.0,
            fuel_efficiency=6.5,
            accessibility_level="basic"
        ),
        OXData(
            name="Articulated_Buses",
            total_buses=15,
            capacity_per_bus=80,
            operating_cost_per_trip=65.0,
            maintenance_factor=1.3,
            fuel_efficiency=4.8,
            accessibility_level="enhanced"
        ),
        OXData(
            name="Electric_Buses",
            total_buses=10,
            capacity_per_bus=45,
            operating_cost_per_trip=35.0,
            maintenance_factor=0.8,
            fuel_efficiency=12.0,  # km/kWh equivalent
            accessibility_level="full"
        ),
        OXData(
            name="Hybrid_Buses",
            total_buses=20,
            capacity_per_bus=55,
            operating_cost_per_trip=40.0,
            maintenance_factor=0.9,
            fuel_efficiency=8.2,
            accessibility_level="enhanced"
        )
    ]

    # Transit lines with service requirements
    transit_lines_data = [
        OXData(
            name="Line_A_Downtown",
            daily_demand=2500,
            minimum_trips=40,
```

(continues on next page)

(continued from previous page)

```
            maximum_trips=80,
            route_length=15.2,
            peak_hour_multiplier=1.8,
            accessibility_required="basic",
            restricted_bus_groups=[]
        ),
        OXData(
            name="Line_B_Suburban",
            daily_demand=1800,
            minimum_trips=30,
            maximum_trips=60,
            route_length=22.5,
            peak_hour_multiplier=1.4,
            accessibility_required="enhanced",
            restricted_bus_groups=["Standard_Buses"]
        ),
        OXData(
            name="Line_C_Express",
            daily_demand=3200,
            minimum_trips=50,
            maximum_trips=100,
            route_length=28.0,
            peak_hour_multiplier=2.1,
            accessibility_required="full",
            restricted_bus_groups=["Standard_Buses", "Hybrid_Buses"]
        ),
        OXData(
            name="Line_D_Local",
            daily_demand=1200,
            minimum_trips=25,
            maximum_trips=45,
            route_length=12.8,
            peak_hour_multiplier=1.2,
            accessibility_required="basic",
            restricted_bus_groups=["Articulated_Buses"]
        )
    ]

    return OXDatabase(bus_groups_data), OXDatabase(transit_lines_data)
```

Problem Creation

```
def create_bus_assignment_problem():
    """Create the Goal Programming problem for bus assignment."""

    bus_groups_db, transit_lines_db = create_bus_assignment_data()

    # Create Goal Programming problem
    problem = OXGPProblem()
```

(continues on next page)

(continued from previous page)

```
    # Create decision variables: trips[bus_group][transit_line]
    trip_variables = {}

    for bus_group in bus_groups_db:
        trip_variables[bus_group.name] = {}

        for transit_line in transit_lines_db:
            # Check if bus group is restricted on this line
            if bus_group.name not in transit_line.restricted_bus_groups:
                var_name = f"trips_{bus_group.name}_{transit_line.name}"

                variable = problem.create_decision_variable(
                    var_name=var_name,
                    description=f"Trips by {bus_group.name} on {transit_line.
↪name}",
                    lower_bound=0,
                    upper_bound=transit_line.maximum_trips,
                    variable_type="integer"
                )

                trip_variables[bus_group.name][transit_line.name] = variable

    return problem, trip_variables, bus_groups_db, transit_lines_db
```

Goal Constraints Implementation

```
def add_goal_constraints(problem, trip_variables, bus_groups_db):
    """Add goal programming constraints for fleet utilization."""

    goal_constraints = []

    for bus_group in bus_groups_db:
        # Calculate target utilization (80% of fleet capacity)
        target_utilization = int(bus_group.total_buses * 0.8)

        # Get all trip variables for this bus group
        bus_group_vars = []
        for line_vars in trip_variables[bus_group.name].values():
            bus_group_vars.append(line_vars.id)

        if bus_group_vars:
            # Create goal constraint: sum of trips should equal target
            goal_constraint = problem.create_goal_constraint(
                variables=bus_group_vars,
                weights=[1] * len(bus_group_vars),
                target_value=target_utilization,
                description=f"Fleet utilization target for {bus_group.name}"
            )
```

(continues on next page)

(continued from previous page)

```
            goal_constraints.append(goal_constraint)

    return goal_constraints
```

Operational Constraints

```
def add_operational_constraints(problem, trip_variables, bus_groups_db, transit_
↪lines_db):
    """Add operational constraints for the bus assignment problem."""

    # 1. Minimum service requirements for each line
    for transit_line in transit_lines_db:
        line_vars = []
        line_weights = []

        for bus_group in bus_groups_db:
            if (bus_group.name in trip_variables and
                transit_line.name in trip_variables[bus_group.name]):

                var = trip_variables[bus_group.name][transit_line.name]
                line_vars.append(var.id)
                line_weights.append(1)

        if line_vars:
            problem.create_constraint(
                variables=line_vars,
                weights=line_weights,
                operator=RelationalOperators.GREATER_THAN_EQUAL,
                value=transit_line.minimum_trips,
                description=f"Minimum service for {transit_line.name}"
            )

    # 2. Fleet capacity constraints
    for bus_group in bus_groups_db:
        if bus_group.name in trip_variables:
            group_vars = []
            for line_vars in trip_variables[bus_group.name].values():
                group_vars.append(line_vars.id)

            if group_vars:
                problem.create_constraint(
                    variables=group_vars,
                    weights=[1] * len(group_vars),
                    operator=RelationalOperators.LESS_THAN_EQUAL,
                    value=bus_group.total_buses,
                    description=f"Fleet capacity for {bus_group.name}"
                )

    # 3. Demand coverage constraints
```

(continues on next page)

(continued from previous page)

```
    for transit_line in transit_lines_db:
        line_vars = []
        capacity_weights = []

        for bus_group in bus_groups_db:
            if (bus_group.name in trip_variables and
                transit_line.name in trip_variables[bus_group.name]):

                var = trip_variables[bus_group.name][transit_line.name]
                line_vars.append(var.id)
                # Weight by bus capacity
                capacity_weights.append(bus_group.capacity_per_bus)

        if line_vars:
            # Total capacity should meet daily demand
            problem.create_constraint(
                variables=line_vars,
                weights=capacity_weights,
                operator=RelationalOperators.GREATER_THAN_EQUAL,
                value=transit_line.daily_demand,
                description=f"Demand coverage for {transit_line.name}"
            )
```

Complete Solution

```
def solve_bus_assignment_problem():
    """Solve the complete bus assignment optimization problem."""

    print("  Bus Assignment Problem - Goal Programming")
    print("=" * 60)

    # Create problem
    problem, trip_variables, bus_groups_db, transit_lines_db = create_bus_
↪assignment_problem()

    # Add constraints
    goal_constraints = add_goal_constraints(problem, trip_variables, bus_groups_
↪db)
    add_operational_constraints(problem, trip_variables, bus_groups_db, transit_
↪lines_db)

    print(f"Problem created with:")
    print(f"  Variables: {len(problem.variables)}")
    print(f"  Constraints: {len(problem.constraints)}")
    print(f"  Goal Constraints: {len(goal_constraints)}")

    # Solve with multiple solvers
    from solvers import solve
```

(continues on next page)

(continued from previous page)

```
    solvers_to_try = ['ORTools', 'Gurobi']

    for solver_name in solvers_to_try:
        try:
            print(f"\n  Solving with {solver_name}...")
            status, solution = solve(problem, solver_name)

            if solution and solution[0].objective_value is not None:
                print(f"  {solver_name} Status: {status}")
                analyze_bus_assignment_solution(
                    solution[0], trip_variables, bus_groups_db, transit_lines_db
                )
                return solution[0]
            else:
                print(f"  {solver_name} failed to find solution")

        except Exception as e:
            print(f"  {solver_name} error: {e}")

    print("  No solver could find a solution")
    return None
```

Solution Analysis

```
def analyze_bus_assignment_solution(solution, trip_variables, bus_groups_db,
↪transit_lines_db):
    """Analyze and display the optimal bus assignment solution."""

    print(f"\n  Optimal Bus Assignment Solution")
    print(f"Goal Programming Objective: {solution.objective_value:.4f}")
    print()

    # Assignment matrix display
    print("  Bus Assignment Matrix:")
    print("-" * 80)

    # Header
    header = "Bus Group".ljust(20)
    for line in transit_lines_db:
        header += line.name.ljust(15)
    header += "Total".ljust(10)
    print(header)
    print("-" * 80)

    # Assignment data
    total_assignments = {}
    line_totals = {line.name: 0 for line in transit_lines_db}

    for bus_group in bus_groups_db:
```

(continues on next page)

(continued from previous page)

```
        row = bus_group.name.ljust(20)
        group_total = 0

        for transit_line in transit_lines_db:
            if (bus_group.name in trip_variables and
                transit_line.name in trip_variables[bus_group.name]):

                var = trip_variables[bus_group.name][transit_line.name]
                trips = solution.variable_values.get(var.id, 0)
                row += f”{trips:>12.0f}   ”
                group_total += trips
                line_totals[transit_line.name] += trips
            else:
                row += f”{'---':>12}   ”

        row += f”{group_total:>8.0f}”
        total_assignments[bus_group.name] = group_total
        print(row)

    # Totals row
    totals_row = ”TOTALS”.ljust(20)
    grand_total = 0
    for line in transit_lines_db:
        totals_row += f”{line_totals[line.name]:>12.0f}   ”
        grand_total += line_totals[line.name]
    totals_row += f”{grand_total:>8.0f}”
    print(”-” * 80)
    print(totals_row)

    # Fleet utilization analysis
    print(”\n□  Fleet Utilization Analysis:”)
    print(”-” * 50)
    for bus_group in bus_groups_db:
        assigned = total_assignments.get(bus_group.name, 0)
        capacity = bus_group.total_buses
        utilization = (assigned / capacity) * 100 if capacity > 0 else 0

        status = ”□ ” if 70 <= utilization <= 90 else ”□ ” if utilization > 0□
↪else ”□ ”
        print(f”{bus_group.name:<20}: {assigned:>3.0f}/{capacity:>3} buses (
↪{utilization:>5.1f}%) {status}”)

    # Service coverage analysis
    print(”\n□  Service Coverage Analysis:”)
    print(”-” * 50)
    for transit_line in transit_lines_db:
        trips_assigned = line_totals[transit_line.name]
        min_required = transit_line.minimum_trips
        demand = transit_line.daily_demand
```

(continues on next page)

(continued from previous page)

```
        # Calculate total capacity provided
        total_capacity = 0
        for bus_group in bus_groups_db:
            if (bus_group.name in trip_variables and
                transit_line.name in trip_variables[bus_group.name]):
                var = trip_variables[bus_group.name][transit_line.name]
                trips = solution.variable_values.get(var.id, 0)
                total_capacity += trips * bus_group.capacity_per_bus

        coverage = (total_capacity / demand) * 100 if demand > 0 else 0
        service_status = ”▯ ” if trips_assigned >= min_required else ”▯ ”
        coverage_status = ”▯ ” if coverage >= 100 else ”▯ ” if coverage >= 80▯
↪else ”▯ ”

        print(f”{transit_line.name:<20}: {trips_assigned:>3.0f} trips (min: {min_
↪required}) {service_status}”)
        print(f”{'':>21} Capacity: {total_capacity:>4.0f} (demand: {demand})
↪{coverage_status}”)

    # Cost analysis
    print(”\n▯  Cost Analysis:”)
    print(”-” * 40)
    total_cost = 0

    for bus_group in bus_groups_db:
        group_cost = 0
        for transit_line in transit_lines_db:
            if (bus_group.name in trip_variables and
                transit_line.name in trip_variables[bus_group.name]):
                var = trip_variables[bus_group.name][transit_line.name]
                trips = solution.variable_values.get(var.id, 0)
                cost = trips * bus_group.operating_cost_per_trip
                group_cost += cost

        total_cost += group_cost
        print(f”{bus_group.name:<20}: ${group_cost:>8.2f}”)

    print(”-” * 40)
    print(f”{'Total Daily Cost':<20}: ${total_cost:>8.2f}”)
```

Advanced Analysis Features

```
def perform_sensitivity_analysis(base_solution, trip_variables, bus_groups_db):
    ”””Perform sensitivity analysis on fleet sizes.”””

    print(”\n▯  Fleet Size Sensitivity Analysis”)
    print(”=” * 50)
```

(continues on next page)

(continued from previous page)

```
    base_objective = base_solution.objective_value

    for bus_group in bus_groups_db:
        print(f”\nAnalyzing {bus_group.name}:”)

        # Test different fleet sizes
        fleet_sizes = [
            bus_group.total_buses - 2,
            bus_group.total_buses - 1,
            bus_group.total_buses,
            bus_group.total_buses + 1,
            bus_group.total_buses + 2
        ]

        for new_size in fleet_sizes:
            if new_size <= 0:
                continue

            # Create modified problem (simplified for demo)
            print(f”  Fleet size {new_size}: Impact analysis would go here”)
            # In real implementation, modify constraints and re-solve

def generate_alternative_scenarios(trip_variables, bus_groups_db, transit_lines_
↪db):
    ”””Generate alternative scenarios with different priorities.”””

    scenarios = [
        {
            'name': 'Cost Minimization',
            'description': 'Prioritize operational cost reduction',
            'modifications': 'Increase weight on operating costs'
        },
        {
            'name': 'Service Quality Focus',
            'description': 'Prioritize passenger service levels',
            'modifications': 'Increase minimum service requirements'
        },
        {
            'name': 'Environmental Priority',
            'description': 'Favor electric and hybrid buses',
            'modifications': 'Bonus for eco-friendly bus assignments'
        }
    ]

    print(”\n□  Alternative Scenario Analysis”)
    print(”=” * 50)

    for scenario in scenarios:
        print(f”\n{scenario['name']}:”)
```

(continues on next page)

(continued from previous page)

```
        print(f"  Description: {scenario['description']}")
        print(f"  Approach: {scenario['modifications']}")
        # Implementation would create and solve modified problems
```

Running the Complete Example

```
def main():
    """Run the complete bus assignment example."""

    print("  OptiX Bus Assignment Problem - Goal Programming Example")
    print("=" * 70)
    print("Demonstrates multi-objective optimization for public transportation")
    print("Features: Goal Programming, Real-world constraints, Multi-criteria
↪analysis")
    print()

    # Solve the main problem
    solution = solve_bus_assignment_problem()

    if solution:
        # Get problem components for analysis
        _, trip_variables, bus_groups_db, transit_lines_db = create_bus_
↪assignment_problem()

        # Additional analyses
        perform_sensitivity_analysis(solution, trip_variables, bus_groups_db)
        generate_alternative_scenarios(trip_variables, bus_groups_db, transit_
↪lines_db)

        print("\n  Bus assignment optimization completed successfully!")
        print("\n  Key Insights:")
        print("• Goal Programming effectively balances competing objectives")
        print("• Fleet utilization targets guide resource allocation")
        print("• Service quality constraints ensure passenger satisfaction")
        print("• Operational constraints maintain system feasibility")
        print("• Multi-criteria analysis reveals trade-offs and opportunities")

    else:
        print("  Failed to find optimal bus assignment solution")

if __name__ == "__main__":
    main()
```

Expected Results

The optimization typically produces solutions with:

Fleet Utilization: 75-85% for most bus groups **Service Coverage**: 100%+ demand coverage on all lines **Cost Efficiency**: Balanced operational costs across bus types **Goal Achievement**:

Minimal deviations from utilization targets

Key Learning Points

1. **Goal Programming**: Managing multiple competing objectives
2. **Real-world Complexity**: Handling operational constraints and restrictions
3. **Multi-criteria Analysis**: Understanding trade-offs in transportation planning
4. **Data Integration**: Using structured data for complex optimization
5. **Solution Interpretation**: Analyzing results for practical implementation

Extensions and Variations

Try these modifications to explore further:

- **Dynamic Scheduling**: Add time-based constraints for peak/off-peak periods
- **Maintenance Planning**: Include bus maintenance schedules and constraints
- **Driver Assignment**: Integrate crew scheduling with bus assignment
- **Route Optimization**: Combine with route planning optimization
- **Stochastic Demand**: Handle uncertain passenger demand patterns
- **Multi-day Planning**: Extend to weekly or monthly planning horizons

Tip

Advanced Technique: This example demonstrates how Goal Programming can handle the complexity of real-world transportation systems where multiple stakeholders have different priorities and constraints.

See also

- ../tutorials/goal_programming - Goal Programming theory and techniques
- *Problem Types* (page 59) - Understanding problem type selection
- *Problem Module* (page 81) - Goal Programming API documentation
- *Classic Diet Problem* (page 34) - Comparison with Linear Programming approach

9.3.3 Quick Example Browser

Linear Programming

Diet Problem - Cost minimization

Classic optimization problem minimizing food costs while meeting nutritional requirements.

- **Complexity**: Beginner
- **Concepts**: Linear constraints, objective optimization

- **Domain**: Nutrition and health

Production Planning - Resource allocation
Manufacturing optimization balancing production costs, inventory, and demand.

- **Complexity**: Intermediate
- **Concepts**: Multi-period planning, capacity constraints
- **Domain**: Manufacturing

Goal Programming

Bus Assignment - Multi-objective transportation
Public transit optimization balancing cost, service quality, and resource utilization.

- **Complexity**: Intermediate
- **Concepts**: Goal constraints, deviation variables
- **Domain**: Transportation

Advanced Applications

Portfolio Optimization - Financial planning
Investment allocation with risk management and diversification requirements.

- **Complexity**: Advanced
- **Concepts**: Risk modeling, correlation constraints
- **Domain**: Finance

9.3.4 Example Features

Each example includes:

 Complete Source Code - Fully functional implementations **Mathematical Formulation** - Clear problem definition **Step-by-Step Explanation** - Detailed implementation guide **Real-World Data** - Practical datasets and scenarios **Solution Analysis** - Results interpretation and insights **Extensions** - Ideas for further development **Performance Tips** - Optimization best practices

9.3.5 Getting Started

1. **Choose by Interest**: Select examples matching your domain
2. **Start Simple**: Begin with Diet Problem for LP basics
3. **Progress Gradually**: Move to Bus Assignment for GP concepts
4. **Customize**: Adapt examples to your specific needs
5. **Experiment**: Try the suggested extensions and variations

9.3.6 Running Examples

All examples are located in the `samples/` directory:

```
# Navigate to examples
cd samples/

# Run diet problem
poetry run python diet_problem/01_diet_problem.py

# Run bus assignment
poetry run python bus_assignment_problem/03_bus_assignment_problem.py
```

9.3.7 Common Patterns

Variable Creation

```
# From data objects
for product in products_db:
    problem.create_decision_variable(
        var_name=f"production_{product.name}",
        description=f"Production level for {product.name}",
        lower_bound=0,
        upper_bound=product.max_capacity
    )
```

Constraint Patterns

```
# Resource constraints
problem.create_constraint(
    variables=production_vars,
    weights=resource_consumption,
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=available_resources
)

# Demand constraints
problem.create_constraint(
    variables=[product_var],
    weights=[1],
    operator=RelationalOperators.GREATER_THAN_EQUAL,
    value=minimum_demand
)
```

Solution Analysis

```
def analyze_solution(solution, problem):
    print(f"Objective Value: {solution.objective_value}")

    for variable in problem.variables:
        value = solution.variable_values.get(variable.id, 0)
        if abs(value) > 1e-6:
            print(f"{variable.name}: {value:.2f}")
```

9.3.8 Best Practices

Problem Modeling

- Start with simple formulations
- Validate constraints early
- Use meaningful variable names
- Add comprehensive descriptions

Performance

- Monitor problem size (variables/constraints)
- Use appropriate variable bounds
- Choose optimal solver for problem type
- Profile large-scale problems

Development

- Implement validation functions
- Create visualization of results
- Add sensitivity analysis
- Document assumptions and limitations

9.3.9 Contributing Examples

We welcome contributions of new examples! Please ensure:

- **Complete Implementation**: Working code with all dependencies
- **Clear Documentation**: Problem description and solution approach
- **Real-World Relevance**: Practical application scenarios
- **Educational Value**: Clear learning objectives
- **Code Quality**: Following project conventions

Submit examples via GitHub pull requests with:

1. Source code in `samples/` directory
2. Documentation in `docs/source/examples/`
3. Test cases and validation
4. README with usage instructions

Tip

Learning Path: Start with Diet Problem □ Bus Assignment □ Production Planning □ Portfolio Optimization for a comprehensive understanding of OptiX capabilities.

Note

All examples include comprehensive error handling, input validation, and detailed output analysis to demonstrate production-ready optimization applications.

9.3.10 See Also

- *Quick Start Guide* (page 24) - Get started with basic concepts
- ../user_guide/index - Understanding problem types and features
- ../tutorials/index - Step-by-step learning modules
- ../api/index - Complete API reference

9.4 Problem Types

OptiX supports three main types of optimization problems with increasing complexity: Constraint Satisfaction Problems (CSP), Linear Programming (LP), and Goal Programming (GP). This guide explains when and how to use each type.

9.4.1 Constraint Satisfaction Problems (CSP)

CSPs focus on finding any solution that satisfies all constraints without optimizing any particular objective. They are the foundation for more complex problem types.

When to Use CSP

- Finding feasible solutions to complex constraint systems
- Scheduling problems where any valid schedule is acceptable
- Configuration problems with multiple requirements
- Preprocessing to check problem feasibility

Key Characteristics

- **Variables**: Decision variables with bounds and types
- **Constraints**: Linear and special constraints that must be satisfied
- **No Objective**: Focus on feasibility, not optimality
- **Solution**: Any point that satisfies all constraints

```python
from problem import OXCSPProblem
from constraints import RelationalOperators

# Create CSP for employee scheduling
csp = OXCSPProblem()

# Variables: work assignments (binary)
```

(continues on next page)

(continued from previous page)

```
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
shifts = ['Morning', 'Evening']
employees = ['Alice', 'Bob', 'Carol']

for employee in employees:
    for day in days:
        for shift in shifts:
            csp.create_decision_variable(
                var_name=f"{employee}_{day}_{shift}",
                description=f"{employee} works {shift} on {day}",
                lower_bound=0,
                upper_bound=1,
                variable_type="binary"
            )

# Constraint: Each shift must be covered
for day in days:
    for shift in shifts:
        shift_vars = [
            var.id for var in csp.variables
            if f"{day}_{shift}" in var.name
        ]
        csp.create_constraint(
            variables=shift_vars,
            weights=[1] * len(shift_vars),
            operator=RelationalOperators.GREATER_THAN_EQUAL,
            value=1,
            description=f"Cover {shift} shift on {day}"
        )

# Constraint: No employee works both shifts same day
for employee in employees:
    for day in days:
        morning_var = next(v for v in csp.variables if f"{employee}_{day}_Morning
↪" in v.name)
        evening_var = next(v for v in csp.variables if f"{employee}_{day}_Evening
↪" in v.name)

        csp.create_constraint(
            variables=[morning_var.id, evening_var.id],
            weights=[1, 1],
            operator=RelationalOperators.LESS_THAN_EQUAL,
            value=1,
            description=f"{employee} works at most one shift on {day}"
        )
```

9.4.2 Linear Programming (LP)

Linear Programming extends CSP by adding an objective function to optimize. LP problems seek to maximize or minimize a linear objective subject to linear constraints.

When to Use LP

- Resource allocation with clear optimization goals
- Production planning to maximize profit or minimize cost
- Transportation problems minimizing shipping costs
- Portfolio optimization with linear objectives

Key Characteristics

- **Variables**: Continuous, integer, or binary decision variables
- **Constraints**: Linear equality and inequality constraints
- **Objective**: Single linear objective function (minimize or maximize)
- **Solution**: Optimal point that maximizes/minimizes the objective

Mathematical Form

$$\text{minimize/maximize} \quad \sum_{i=1}^{n} c_i x_i \tag{9.6}$$

$$\text{subject to} \quad \sum_{i=1}^{n} a_{ji} x_i \leq b_j, \quad j = 1, \ldots, m \tag{9.7}$$

$$x_i \geq 0, \quad i = 1, \ldots, n \tag{9.8}$$

Where: - x_i are decision variables - c_i are objective coefficients - a_{ji} are constraint coefficients - b_j are constraint bounds

```python
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators

# Create LP for production planning
lp = OXLPProblem()

# Products to manufacture
products = [
    {'name': 'Product_A', 'profit': 50, 'labor': 2, 'material': 1},
    {'name': 'Product_B', 'profit': 40, 'labor': 3, 'material': 2},
    {'name': 'Product_C', 'profit': 60, 'labor': 1, 'material': 3}
]

# Create production variables
for product in products:
    lp.create_decision_variable(
        var_name=f”production_{product['name']}”,
        description=f”Units of {product['name']} to produce”,
```

(continues on next page)

(continued from previous page)

```
        lower_bound=0,
        upper_bound=1000,
        variable_type="continuous"
    )

# Resource constraints
# Labor constraint: total labor <= 1000 hours
labor_vars = [var.id for var in lp.variables]
labor_weights = [product['labor'] for product in products]

lp.create_constraint(
    variables=labor_vars,
    weights=labor_weights,
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=1000,
    description="Labor hours constraint"
)

# Material constraint: total material <= 800 units
material_weights = [product['material'] for product in products]

lp.create_constraint(
    variables=labor_vars,  # Same variables
    weights=material_weights,
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=800,
    description="Material constraint"
)

# Objective: maximize total profit
profit_weights = [product['profit'] for product in products]

lp.create_objective_function(
    variables=labor_vars,
    weights=profit_weights,
    objective_type=ObjectiveType.MAXIMIZE
)

# Solve the problem
from solvers import solve
status, solution = solve(lp, 'ORTools')
```

9.4.3 Goal Programming (GP)

Goal Programming handles multiple, often conflicting objectives by formulating them as goals with associated deviation variables and priorities.

When to Use GP

- Multi-criteria decision making
- Problems with conflicting objectives
- Situations where trade-offs are necessary
- When exact goal achievement is less important than minimizing deviations

Key Characteristics

- **Variables**: Decision variables plus deviation variables
- **Constraints**: Regular constraints plus goal constraints
- **Objectives**: Multiple goals with priorities or weights
- **Solution**: Minimize weighted deviations from goals

Mathematical Form

$$\text{minimize} \quad \sum_{i=1}^{k} w_i^+ d_i^+ + w_i^- d_i^- \tag{9.9}$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j + d_i^- - d_i^+ = g_i, \quad i = 1, \ldots, k \tag{9.10}$$

$$\text{regular constraints} \tag{9.11}$$

$$x_j, d_i^+, d_i^- \geq 0 \tag{9.12}$$

Where: - d_i^+, d_i^- are positive and negative deviation variables - w_i^+, w_i^- are weights for deviations - g_i are goal targets

```python
from problem import OXGPProblem
from constraints import RelationalOperators

# Create GP for workforce planning
gp = OXGPProblem()

# Decision variables: number of employees to hire
departments = ['Engineering', 'Sales', 'Support']

for dept in departments:
    gp.create_decision_variable(
        var_name=f"hire_{dept}",
        description=f"Employees to hire in {dept}",
        lower_bound=0,
        upper_bound=50,
        variable_type="integer"
    )

# Goal constraints with priorities
goals = [
    {
        'description': 'Total workforce target of 100 employees',
```

(continues on next page)

(continued from previous page)

```
        'variables': [var.id for var in gp.variables],
        'weights': [1, 1, 1],
        'target': 100,
        'priority': 1
    },
    {
        'description': 'Engineering should be 40% of workforce',
        'variables': [gp.variables[0].id],  # Engineering
        'weights': [1],
        'target': 40,
        'priority': 2
    },
    {
        'description': 'Balance between Sales and Support',
        'variables': [gp.variables[1].id, gp.variables[2].id],  # Sales, Support
        'weights': [1, -1],
        'target': 0,  # Equal hiring
        'priority': 3
    }
]

# Add goal constraints
for goal in goals:
    gp.create_goal_constraint(
        variables=goal['variables'],
        weights=goal['weights'],
        target_value=goal['target'],
        description=goal['description']
    )

# Additional regular constraints
# Budget constraint: hiring costs <= $500,000
hiring_costs = [80000, 60000, 50000]  # Cost per hire by department

gp.create_constraint(
    variables=[var.id for var in gp.variables],
    weights=hiring_costs,
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=500000,
    description="Budget constraint"
)
```

9.4.4 Problem Type Comparison

9.4.5 Advanced Problem Features

Special Constraints

All problem types support special constraints for non-linear operations:

```
from problem import SpecialConstraintType

# Multiplication constraint: production * price = revenue
problem.create_special_constraint(
    constraint_type=SpecialConstraintType.MULTIPLICATION,
    left_variable_id=production_var.id,
    right_variable_id=price_var.id,
    result_variable_id=revenue_var.id
)

# Conditional constraint: if condition then action
problem.create_special_constraint(
    constraint_type=SpecialConstraintType.CONDITIONAL,
    left_variable_id=condition_var.id,
    right_variable_id=action_var.id,
    result_variable_id=result_var.id
)
```

Database Integration

Create variables and constraints from data objects:

```
from data import OXData, OXDatabase

# Create data structure
facilities = OXDatabase([
    OXData(name="Plant_A", capacity=500, cost=1000),
    OXData(name="Plant_B", capacity=300, cost=800)
])

customers = OXDatabase([
    OXData(name="Customer_1", demand=200, location="NY"),
    OXData(name="Customer_2", demand=150, location="CA")
])

# Create variables from Cartesian product
problem.create_variables_from_database_objects(
    database_objects=[facilities, customers],
    variable_name_template="ship_{0}_{1}",
    variable_description_template="Shipment from {0} to {1}",
    lower_bound=0,
    upper_bound=1000
)
```

9.4.6 Problem Selection Guide

Choosing the Right Problem Type

Migration Between Problem Types

You can easily migrate between problem types as requirements evolve:

```
# Start with CSP to check feasibility
csp = OXCSPProblem()
# ... add variables and constraints

# Convert to LP when objective becomes clear
lp = OXLPProblem()
# Copy variables and constraints from CSP
for variable in csp.variables:
    lp.add_variable(variable)
for constraint in csp.constraints:
    lp.add_constraint(constraint)

# Add objective function
lp.create_objective_function(variables, weights, ObjectiveType.MAXIMIZE)

# Evolve to GP when multiple objectives emerge
gp = OXGPProblem()
# Copy from LP and add goal constraints
```

9.4.7 Best Practices

Problem Modeling

1. **Start Simple**: Begin with CSP to ensure feasibility
2. **Add Gradually**: Introduce objectives and goals incrementally
3. **Validate Early**: Check constraints before adding complexity
4. **Use Data**: Leverage database integration for complex scenarios

Performance Tips

1. **Variable Bounds**: Tighten bounds to reduce search space
2. **Constraint Order**: Place restrictive constraints first
3. **Problem Size**: Monitor variable and constraint counts
4. **Solver Selection**: Choose appropriate solver for problem type

Common Pitfalls

1. **Infeasible Problems**: Over-constraining the solution space
2. **Unbounded Objectives**: Missing constraints on decision variables
3. **Numerical Issues**: Very large or very small coefficients
4. **Goal Conflicts**: Incompatible goals in GP problems

Tip

Development Workflow: Start with CSP to validate your constraint model, then add objectives to create LP, and finally introduce multiple goals for GP.

9.4.8 See Also

- *Quick Start Guide* (page 24) - Get started with your first problem
- constraints - Advanced constraint modeling
- *Examples* (page 32) - Real-world problem examples
- *Problem Module* (page 81) - Complete API reference

9.5 Linear Programming Tutorial

This tutorial provides a comprehensive introduction to Linear Programming (LP) using OptiX. You'll learn the theory behind LP, how to formulate problems, and implement solutions step by step.

9.5.1 What is Linear Programming?

Linear Programming is a mathematical optimization technique for finding the best outcome (maximum or minimum value) in a mathematical model with linear relationships.

Key Components

1. **Decision Variables**: Variables you can control
2. **Objective Function**: What you want to optimize (linear combination of variables)
3. **Constraints**: Linear inequalities or equalities that limit feasible solutions
4. **Feasible Region**: Set of all points satisfying constraints
5. **Optimal Solution**: Best feasible point according to objective function

Mathematical Form

Standard LP formulation:

$$\begin{aligned} \text{minimize/maximize} \quad & c^T x && (9.13) \\ \text{subject to} \quad & Ax \leq b && (9.14) \\ & x \geq 0 && (9.15) \end{aligned}$$

Where: - x is the vector of decision variables - c is the vector of objective coefficients - A is the constraint coefficient matrix - b is the vector of constraint bounds

9.5.2 Tutorial 1: Basic LP Problem

Let's start with a simple two-variable problem that can be visualized graphically.

Problem Statement

A furniture company makes chairs and tables. Each chair requires 1 hour of labor and 2 units of wood, earning $3 profit. Each table requires 2 hours of labor and 1 unit of wood, earning $2 profit. The company has 100 hours of labor and 80 units of wood available. How many chairs and tables should they make to maximize profit?

Mathematical Formulation

Decision Variables: - x_1 = number of chairs to make - x_2 = number of tables to make

Objective Function: Maximize profit: $3x_1 + 2x_2$

Constraints: - Labor: $1x_1 + 2x_2 \leq 100$ - Wood: $2x_1 + 1x_2 \leq 80$ - Non-negativity: $x_1, x_2 \geq 0$

Implementation

```
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from solvers import solve

def solve_furniture_problem():
    """Solve the furniture production problem."""

    # Step 1: Create LP problem
    problem = OXLPProblem()

    # Step 2: Define decision variables
    problem.create_decision_variable(
        var_name="chairs",
        description="Number of chairs to produce",
        lower_bound=0,
        upper_bound=1000,  # Reasonable upper bound
        variable_type="continuous"
    )

    problem.create_decision_variable(
        var_name="tables",
        description="Number of tables to produce",
        lower_bound=0,
        upper_bound=1000,
        variable_type="continuous"
    )

    # Step 3: Add constraints
    # Labor constraint: 1*chairs + 2*tables <= 100
    problem.create_constraint(
        variables=[var.id for var in problem.variables],
        weights=[1, 2],  # Labor hours per unit
        operator=RelationalOperators.LESS_THAN_EQUAL,
        value=100,  # Available labor hours
        description="Labor hours constraint"
    )

    # Wood constraint: 2*chairs + 1*tables <= 80
    problem.create_constraint(
        variables=[var.id for var in problem.variables],
        weights=[2, 1],  # Wood units per unit
        operator=RelationalOperators.LESS_THAN_EQUAL,
```

(continues on next page)

(continued from previous page)

```
        value=80,  # Available wood units
        description=”Wood units constraint”
    )

    # Step 4: Set objective function
    # Maximize: 3*chairs + 2*tables
    problem.create_objective_function(
        variables=[var.id for var in problem.variables],
        weights=[3, 2],  # Profit per unit
        objective_type=ObjectiveType.MAXIMIZE
    )

    # Step 5: Solve the problem
    status, solution = solve(problem, 'ORTools')

    # Step 6: Analyze results
    if solution and solution[0].objective_value is not None:
        print(”□  Furniture Production Optimization”)
        print(”=” * 40)
        print(f”Status: {status}”)
        print(f”Maximum Profit: ${solution[0].objective_value:.2f}”)
        print()

        for variable in problem.variables:
            value = solution[0].variable_values.get(variable.id, 0)
            print(f”{variable.description}: {value:.2f}”)

        return problem, solution[0]
    else:
        print(”No optimal solution found”)
        return problem, None

# Run the example
problem, solution = solve_furniture_problem()
```

Understanding the Solution

The optimal solution typically produces approximately: - **20 chairs** and **40 tables** - **Maximum profit: $140**

This solution is found at the intersection of the two constraint lines, demonstrating a key LP property: optimal solutions occur at vertices of the feasible region.

9.5.3 Tutorial 2: Multi-Resource Problem

Let’s expand to a more complex problem with multiple resources and products.

Problem Statement

A factory produces three products (A, B, C) using three resources (labor, material, machine time). We want to maximize profit while respecting resource limitations.

```
def solve_multi_resource_problem():
    """Solve a multi-resource production problem."""

    # Problem data
    products = [
        {'name': 'Product_A', 'profit': 40, 'labor': 1, 'material': 3, 'machine
↪': 1},
        {'name': 'Product_B', 'profit': 30, 'labor': 2, 'material': 1, 'machine
↪': 2},
        {'name': 'Product_C', 'profit': 20, 'labor': 1, 'material': 2, 'machine
↪': 1}
    ]

    resources = {
        'labor': 100,      # Available labor hours
        'material': 150,   # Available material units
        'machine': 80      # Available machine hours
    }

    # Create problem
    problem = OXLPProblem()

    # Create variables
    for product in products:
        problem.create_decision_variable(
            var_name=f"produce_{product['name']}",
            description=f"Units of {product['name']} to produce",
            lower_bound=0,
            upper_bound=1000,
            variable_type="continuous"
        )

    # Resource constraints
    for resource, capacity in resources.items():
        resource_usage = [product[resource] for product in products]

        problem.create_constraint(
            variables=[var.id for var in problem.variables],
            weights=resource_usage,
            operator=RelationalOperators.LESS_THAN_EQUAL,
            value=capacity,
            description=f"{resource.title()} capacity constraint"
```

(continues on next page)

(continued from previous page)

```
        )

    # Objective function: maximize profit
    profit_coefficients = [product['profit'] for product in products]

    problem.create_objective_function(
        variables=[var.id for var in problem.variables],
        weights=profit_coefficients,
        objective_type=ObjectiveType.MAXIMIZE
    )

    # Solve and analyze
    status, solution = solve(problem, 'ORTools')

    if solution and solution[0].objective_value is not None:
        print(”□  Multi-Resource Production Optimization”)
        print(”=” * 50)
        print(f”Maximum Profit: ${solution[0].objective_value:.2f}”)
        print()

        # Production plan
        print(”Production Plan:”)
        total_profit = 0
        for i, (variable, product) in enumerate(zip(problem.variables,
↪products)):
            quantity = solution[0].variable_values.get(variable.id, 0)
            profit = quantity * product['profit']
            total_profit += profit

            print(f”  {product['name']}: {quantity:.2f} units (${profit:.2f})”)

        print(f”\nTotal Profit: ${total_profit:.2f}”)

        # Resource utilization
        print(”\nResource Utilization:”)
        for resource, capacity in resources.items():
            used = sum(
                solution[0].variable_values.get(problem.variables[i].id, 0) *
↪products[i][resource]
                for i in range(len(products))
            )
            utilization = (used / capacity) * 100
            print(f”  {resource.title()}: {used:.1f}/{capacity} ({utilization:.
↪1f}%)”)

    return problem, solution[0] if solution else None
```

9.5.4 Tutorial 3: Advanced LP Concepts

Sensitivity Analysis

Understanding how changes in parameters affect the optimal solution:

```
def perform_sensitivity_analysis(base_problem, base_solution):
    """Perform sensitivity analysis on problem parameters."""

    print("\n▯  Sensitivity Analysis")
    print("=" * 30)

    # Test profit coefficient changes
    print("Profit Coefficient Sensitivity:")
    original_profits = [40, 30, 20]  # Original profit coefficients

    for i, product_name in enumerate(['Product_A', 'Product_B', 'Product_C']):
        print(f"\n{product_name} profit sensitivity:")

        for change in [-20, -10, 0, 10, 20]:  # Percentage changes
            new_profit = original_profits[i] * (1 + change/100)
            print(f"  {change:+3d}% change (${new_profit:.1f}): ", end="")

            # Create modified problem
            modified_problem = create_modified_problem(base_problem, i, new_
↪profit)
            status, solution = solve(modified_problem, 'ORTools')

            if solution:
                obj_change = ((solution[0].objective_value - base_solution.
↪objective_value)
                            / base_solution.objective_value) * 100
                print(f"Objective {obj_change:+5.1f}%")
            else:
                print("No solution")

def create_modified_problem(base_problem, product_index, new_profit):
    """Create a modified problem with changed profit coefficient."""
    # Implementation would copy base problem and modify specific coefficient
    # This is a simplified version for demonstration
    pass
```

Shadow Prices and Dual Solutions

Understanding the value of additional resources:

```
def analyze_shadow_prices(problem, solution):
    """Analyze shadow prices for resource constraints."""

    print("\n▯  Shadow Price Analysis")
    print("=" * 30)
```

(continues on next page)

(continued from previous page)

```
    # Shadow prices indicate the value of one additional unit of each resource
    # In practice, these would be extracted from the solver's dual solution

    print("Resource shadow prices (value per additional unit):")
    print("  Labor: $X.XX per hour")
    print("  Material: $X.XX per unit")
    print("  Machine: $X.XX per hour")
    print("\nNote: Shadow prices available from solver dual solution")
```

Integer and Binary Variables

Handling discrete decisions:

```
def solve_integer_problem():
    """Solve problem with integer variables."""

    problem = OXLPProblem()

    # Integer variables (can't produce fractional units)
    problem.create_decision_variable(
        var_name="machines_type_A",
        description="Number of Type A machines to buy",
        lower_bound=0,
        upper_bound=10,
        variable_type="integer"  # Must be whole number
    )

    problem.create_decision_variable(
        var_name="machines_type_B",
        description="Number of Type B machines to buy",
        lower_bound=0,
        upper_bound=10,
        variable_type="integer"
    )

    # Binary variables (yes/no decisions)
    problem.create_decision_variable(
        var_name="open_facility",
        description="Whether to open new facility",
        lower_bound=0,
        upper_bound=1,
        variable_type="binary"  # 0 or 1 only
    )

    # Budget constraint
    machine_costs = [50000, 30000]  # Cost per machine
    facility_cost = 100000  # Fixed cost to open facility
```

(continues on next page)

(continued from previous page)

```
problem.create_constraint(
    variables=[var.id for var in problem.variables],
    weights=machine_costs + [facility_cost],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=200000,  # Budget limit
    description=”Budget constraint”
)

# Logical constraint: need facility open to buy machines
# machines_type_A <= 10 * open_facility
problem.create_constraint(
    variables=[problem.variables[0].id, problem.variables[2].id],
    weights=[1, -10],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=0,
    description=”Facility requirement for Type A”
)

# Similar constraint for Type B machines
problem.create_constraint(
    variables=[problem.variables[1].id, problem.variables[2].id],
    weights=[1, -10],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=0,
    description=”Facility requirement for Type B”
)

# Objective: maximize production capacity
capacity_per_machine = [100, 80]  # Units per day per machine
facility_capacity = 50  # Additional capacity from facility

problem.create_objective_function(
    variables=[var.id for var in problem.variables],
    weights=capacity_per_machine + [facility_capacity],
    objective_type=ObjectiveType.MAXIMIZE
)

# Solve
status, solution = solve(problem, 'ORTools')

if solution:
    print(”□  Facility and Equipment Planning”)
    print(”=” * 40)
    print(f”Maximum Capacity: {solution[0].objective_value:.0f} units/day”)
    print()

    for variable in problem.variables:
        value = solution[0].variable_values.get(variable.id, 0)
        print(f”{variable.description}: {value:.0f}”)
```

9.5.5 Tutorial 4: Common LP Patterns

Diet/Nutrition Problems

```python
def solve_nutrition_problem():
    """Standard diet problem pattern."""

    foods = [
        {'name': 'Bread', 'cost': 2.0, 'protein': 4, 'fat': 1, 'carbs': 15},
        {'name': 'Milk', 'cost': 3.5, 'protein': 8, 'fat': 5, 'carbs': 12},
        {'name': 'Cheese', 'cost': 8.0, 'protein': 25, 'fat': 25, 'carbs': 1},
        {'name': 'Potato', 'cost': 1.5, 'protein': 2, 'fat': 0, 'carbs': 17}
    ]

    requirements = {
        'protein': {'min': 55, 'max': 200},
        'fat': {'min': 20, 'max': 100},
        'carbs': {'min': 130, 'max': 300}
    }

    # Implementation pattern for diet problems
    problem = OXLPProblem()

    # Variables: quantity of each food
    for food in foods:
        problem.create_decision_variable(
            var_name=f"quantity_{food['name']}",
            description=f"Quantity of {food['name']} (servings)",
            lower_bound=0,
            upper_bound=10,  # Reasonable upper limit
            variable_type="continuous"
        )

    # Nutritional constraints
    for nutrient, limits in requirements.items():
        nutrient_content = [food[nutrient] for food in foods]

        # Minimum requirement
        problem.create_constraint(
            variables=[var.id for var in problem.variables],
            weights=nutrient_content,
            operator=RelationalOperators.GREATER_THAN_EQUAL,
            value=limits['min'],
            description=f"Minimum {nutrient} requirement"
        )

        # Maximum limit
        problem.create_constraint(
            variables=[var.id for var in problem.variables],
            weights=nutrient_content,
            operator=RelationalOperators.LESS_THAN_EQUAL,
```

(continues on next page)

(continued from previous page)

```
            value=limits['max'],
            description=f"Maximum {nutrient} limit"
        )

    # Objective: minimize cost
    costs = [food['cost'] for food in foods]
    problem.create_objective_function(
        variables=[var.id for var in problem.variables],
        weights=costs,
        objective_type=ObjectiveType.MINIMIZE
    )

    return problem
```

Transportation Problems

```
def solve_transportation_problem():
    """Standard transportation problem pattern."""

    # Supply and demand data
    suppliers = [
        {'name': 'Plant_A', 'supply': 300},
        {'name': 'Plant_B', 'supply': 400},
        {'name': 'Plant_C', 'supply': 500}
    ]

    customers = [
        {'name': 'Customer_1', 'demand': 250},
        {'name': 'Customer_2', 'demand': 350},
        {'name': 'Customer_3', 'demand': 400},
        {'name': 'Customer_4', 'demand': 200}
    ]

    # Transportation costs (supplier x customer)
    costs = [
        [8, 6, 10, 9],    # Plant_A to customers
        [9, 12, 13, 7],   # Plant_B to customers
        [14, 9, 16, 5]    # Plant_C to customers
    ]

    problem = OXLPProblem()

    # Variables: shipment quantities
    for i, supplier in enumerate(suppliers):
        for j, customer in enumerate(customers):
            problem.create_decision_variable(
                var_name=f"ship_{supplier['name']}_{customer['name']}",
                description=f"Shipment from {supplier['name']} to {customer['name
↪']}",
```

(continues on next page)

(continued from previous page)

```python
                lower_bound=0,
                upper_bound=min(supplier['supply'], customer['demand']),
                variable_type="continuous"
            )

    # Supply constraints
    var_index = 0
    for i, supplier in enumerate(suppliers):
        supplier_vars = []
        for j in range(len(customers)):
            supplier_vars.append(problem.variables[var_index].id)
            var_index += 1

        problem.create_constraint(
            variables=supplier_vars,
            weights=[1] * len(customers),
            operator=RelationalOperators.LESS_THAN_EQUAL,
            value=supplier['supply'],
            description=f"Supply constraint for {supplier['name']}"
        )

    # Demand constraints
    for j, customer in enumerate(customers):
        customer_vars = []
        for i in range(len(suppliers)):
            var_idx = i * len(customers) + j
            customer_vars.append(problem.variables[var_idx].id)

        problem.create_constraint(
            variables=customer_vars,
            weights=[1] * len(suppliers),
            operator=RelationalOperators.GREATER_THAN_EQUAL,
            value=customer['demand'],
            description=f"Demand constraint for {customer['name']}"
        )

    # Objective: minimize transportation cost
    flat_costs = [cost for row in costs for cost in row]
    problem.create_objective_function(
        variables=[var.id for var in problem.variables],
        weights=flat_costs,
        objective_type=ObjectiveType.MINIMIZE
    )

    return problem
```

9.5.6 Tutorial 5: Debugging and Validation

Common Issues and Solutions

```
def debug_lp_problem(problem):
    """Debug common LP problem issues."""

    print("▯  LP Problem Debugging")
    print("=" * 30)

    issues = []

    # Check for variables
    if not problem.variables:
        issues.append("No decision variables defined")

    # Check for constraints
    if not problem.constraints:
        issues.append("No constraints defined")

    # Check for objective
    if not hasattr(problem, 'objective_function') or not problem.objective_
↪function:
        issues.append("No objective function defined")

    # Check variable bounds
    for var in problem.variables:
        if var.lower_bound > var.upper_bound:
            issues.append(f"Invalid bounds for {var.name}: [{var.lower_bound},
↪{var.upper_bound}]")

        if var.lower_bound == var.upper_bound:
            issues.append(f"Variable {var.name} is fixed to {var.lower_bound}")

    # Check constraint feasibility (basic checks)
    for i, constraint in enumerate(problem.constraints):
        if len(constraint.variables) != len(constraint.weights):
            issues.append(f"Constraint {i}: variables and weights length mismatch
↪")

        if constraint.value < 0 and constraint.operator in [
            RelationalOperators.GREATER_THAN_EQUAL,
            RelationalOperators.GREATER_THAN
        ]:
            issues.append(f"Constraint {i}: may be infeasible (negative RHS▯
↪with >=)")

    # Report issues
    if issues:
        print("Issues found:")
        for issue in issues:
```

(continues on next page)

(continued from previous page)

```
            print(f"   {issue}")
    else:
        print("  No obvious issues detected")

    # Problem statistics
    print(f"\nProblem Statistics:")
    print(f"  Variables: {len(problem.variables)}")
    print(f"  Constraints: {len(problem.constraints)}")

    if hasattr(problem, 'objective_function') and problem.objective_function:
        print(f"  Objective: {problem.objective_function.objective_type.name}")

    return issues

def validate_solution(problem, solution):
    """Validate solution satisfies all constraints."""

    if not solution:
        print("  No solution to validate")
        return False

    print("\n  Solution Validation")
    print("=" * 25)

    violations = []

    for i, constraint in enumerate(problem.constraints):
        # Calculate left-hand side
        lhs = sum(
            constraint.weights[j] * solution.variable_values.get(constraint.
↪variables[j], 0)
            for j in range(len(constraint.variables))
        )

        # Check constraint satisfaction
        satisfied = False
        tolerance = 1e-6

        if constraint.operator == RelationalOperators.LESS_THAN_EQUAL:
            satisfied = lhs <= constraint.value + tolerance
        elif constraint.operator == RelationalOperators.GREATER_THAN_EQUAL:
            satisfied = lhs >= constraint.value - tolerance
        elif constraint.operator == RelationalOperators.EQUAL:
            satisfied = abs(lhs - constraint.value) <= tolerance

        if not satisfied:
            violations.append({
                'constraint': i,
                'description': constraint.description,
```

(continues on next page)

(continued from previous page)

```
                'lhs': lhs,
                'operator': constraint.operator.name,
                'rhs': constraint.value,
                'violation': abs(lhs - constraint.value)
            })

    if violations:
        print(f"□  Found {len(violations)} constraint violations:")
        for v in violations:
            print(f"  Constraint {v['constraint']}: {v['lhs']:.6f} {v['operator
↪']} {v['rhs']}")
        return False
    else:
        print("□  All constraints satisfied")
        return True
```

9.5.7 Best Practices Summary

Problem Formulation

1. Clearly define decision variables
2. Write objective function first
3. Add constraints systematically
4. Use meaningful names and descriptions
5. Validate mathematical formulation

Implementation Tips

1. Start with simple problems
2. Add constraints incrementally
3. Test with known solutions
4. Use debugging functions
5. Validate all solutions

Performance Optimization

1. Tighten variable bounds
2. Remove redundant constraints
3. Use appropriate variable types
4. Monitor problem size
5. Choose optimal solver

Common Pitfalls to Avoid

1. Infeasible constraint combinations
2. Unbounded objectives

3. Numerical precision issues
4. Missing non-negativity constraints
5. Incorrect constraint directions

9.5.8 Next Steps

After mastering these LP concepts:

1. **Practice**: Implement various LP problems from different domains
2. **Advanced Topics**: Explore integer programming and mixed-integer LP
3. **Goal Programming**: Learn multi-objective optimization techniques
4. **Sensitivity Analysis**: Understand parameter changes and their effects
5. **Large-Scale Problems**: Handle real-world problem sizes and complexity

See also

- *Classic Diet Problem* (page 34) - Complete LP implementation
- ../examples/production_planning - Advanced LP example
- goal_programming - Multi-objective optimization
- *Problem Types* (page 59) - Problem type selection guide

9.6 Problem Module

The problem module provides the core problem type classes for representing different types of optimization problems in the OptiX framework. It implements a hierarchical structure supporting Constraint Satisfaction Problems (CSP), Linear Programming (LP), and Goal Programming (GP).

9.6.1 Problem Types

`class problem.OXCSPProblem(`*id: ~uuid.UUID = <factory>, class_name: str = ”, db: ~data.OXDatabase.OXDatabase = <factory>, variables: ~variables.OXVariableSet.OXVariableSet = <factory>, constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>, specials: list[~constraints.OXSpecialConstraints.OXSpecialConstraint] = <factory>, constraints_in_special_constraints: list[~uuid.UUID] = <factory>*`)`

Bases: `OXObject`

Base class for Constraint Satisfaction Problems (CSP).

This class represents a constraint satisfaction problem where the goal is to find values for variables that satisfy a set of constraints. It provides the fundamental structure for optimization problems in the OptiX framework.

`db`

Database containing data objects used in the problem.

Type

OXDatabase (page 207)

`variables`

Set of decision variables in the problem.

Type

OXVariableSet (page 143)

`constraints`

List of linear constraints.

Type

list[5][*OXConstraint* (page 106)]

`specials`

List of special (non-linear) constraints.

Type

list[6][*OXSpecialConstraint* (page 123)]

`constraints_in_special_constraints`

List of constraint IDs used in special constraints to avoid duplication.

Type

list[7][UUID]

Examples

```
>>> problem = OXCSPProblem()
>>> problem.create_decision_variable("x", lower_bound=0, upper_bound=10)
>>> problem.create_decision_variable("y", lower_bound=0, upper_bound=10)
>>> problem.create_constraint(
...     variables=[problem.variables[0].id, problem.variables[1].id],
...     weights=[1, 1],
...     operator=RelationalOperators.LESS_THAN_EQUAL,
...     value=15
... )
```

See also

`OXLPProblem` (page 88): Linear Programming extension of CSP. `OXGPProblem` (page 92): Goal Programming extension of LP.

`db: OXDatabase (page 207)`

`variables: OXVariableSet (page 143)`

`constraints: OXConstraintSet (page 112)`

`specials: list`[8]`[OXSpecialConstraint (page 123)]` `constraints_in_special_constraints: list`[9]`[UUID`[10]`]`

`create_special_constraint(`** (Keyword-only parameters separator (PEP 3102)), constraint_type: SpecialConstraintType (page 98) = SpecialConstraintType.MultiplicativeEquality, **kwargs*`)`

Create a special (non-linear) constraint for the problem.

This method creates various types of special constraints that handle non-linear operations such as multiplication, division, modulo, summation, and conditional logic.

Parameters

- `constraint_type` (`SpecialConstraintType` (page 98)) – The type of special constraint to create. Defaults to MultiplicativeEquality.
- `**kwargs` – Additional keyword arguments specific to the constraint type. See individual constraint creation functions for details.

Returns

The created special constraint object.

Return type

OXSpecialConstraint (page 123)

Raises

`OXception` – If an unknown constraint type is provided.

Examples

```
>>> # Create a multiplication constraint: z = x * y
>>> constraint = problem.create_special_constraint(
...     constraint_type=SpecialConstraintType.MultiplicativeEquality,
...     input_variables=[var_x, var_y]
... )
>>>
>>> # Create a division constraint: z = x // 3
>>> constraint = problem.create_special_constraint(
...     constraint_type=SpecialConstraintType.DivisionEquality,
...     input_variable=var_x,
...     divisor=3
... )
```

See also

`_create_multiplicative_equality_constraint()`: For multiplication constraints. `_create_division_or_modulus_equality_constraint()`: For division/-modulo constraints. `_create_summation_equality_constraint()`: For summation constraints. `_create_conditional_constraint()`: For conditional constraints.

`create_variables_from_db(`**args, var_name_template: str*[11] *= ”, var_description_template: str*[12] *= ”, upper_bound: float*[13] *| int*[14] *= inf, lower_bound: float*[15] *| int*[16] *= 0*`)`

Create decision variables from database objects using Cartesian product.

This method creates decision variables by taking the Cartesian product of specified database object types. For each combination of objects, a new variable is created with the specified bounds and template-based naming.

Parameters

- `*args` – Variable number of database object types to use for variable creation. Each argument should be a type that exists in the problem's database.
- `var_name_template` (`str`[17], `optional`) – Template string for variable names. Can use database object type names as format keys, as well as individual field values with format "{type_name}_{field_name}". Defaults to "".
- `var_description_template` (`str`[18], `optional`) – Template string for variable descriptions. Can use database object type names as format keys, as well as individual field values with format "{type_name}_{field_name}". Defaults to "".
- `upper_bound` (`float`[19] `|` `int`[20], `optional`) – Upper bound for all created variables. Defaults to positive infinity.
- `lower_bound` (`float`[21] `|` `int`[22], `optional`) – Lower bound for all created variables. Defaults to 0.

Raises

`OXception` – If any of the provided argument types don't exist in the database.

Examples

```
>>> # Create variables for all combinations of buses and routes
>>> problem.create_variables_from_db(
...     Bus, Route,
...     var_name_template="bus_{bus_id}_route_{route_id}",
...     var_description_template="Assignment of bus {bus_name} to
↪route {route_name}",
...     upper_bound=1,
...     lower_bound=0
... )
>>>
>>> # Create variables for single object type using field values
>>> problem.create_variables_from_db(
...     Driver,
...     var_name_template="driver_{driver_id}_active",
...     var_description_template="Driver {driver_name} is active",
...     upper_bound=1
... )
```

Note

The method uses the Cartesian product of all specified object types, so the number of created variables equals the product of the counts of each object type.

Template strings can now access both object type names and individual field values from dataclass objects.

`create_decision_variable`(*var_name: str*[23] *= ", description: str*[24] *= ", upper_bound: float*[25] *| int*[26] *= inf, lower_bound: float*[27] *| int*[28] *= 0, **kwargs*)

Create a decision variable for the optimization problem.

Creates a new decision variable with the specified bounds and properties, and adds it to the problem's variable set. The variable can be linked to database objects through keyword arguments.

Parameters

- `var_name` (`str`[29], `optional`) – Name of the variable. Defaults to "".
- `description` (`str`[30], `optional`) – Description of the variable. Defaults to "".
- `upper_bound` (`float`[31] `|` `int`[32], `optional`) – Upper bound for the variable. Defaults to positive infinity.
- `lower_bound` (`float`[33] `|` `int`[34], `optional`) – Lower bound for the variable. Defaults to 0.
- `**kwargs` – Additional keyword arguments linking the variable to database objects. Keys must match database object types.

Raises

`OXception` – If any keyword argument key doesn't match a database object type.

Examples

```
>>> # Create a simple bounded variable
>>> problem.create_decision_variable(
...     var_name="x",
...     description="Production quantity",
...     upper_bound=100,
...     lower_bound=0
... )
>>>
>>> # Create a variable linked to database objects
>>> problem.create_decision_variable(
...     var_name="assignment_bus_1_route_2",
...     description="Bus 1 assigned to route 2",
...     upper_bound=1,
...     lower_bound=0,
...     bus=bus_1_id,
...     route=route_2_id
... )
```

Note

The variable is automatically added to the problem's variable set and can be accessed through the variables attribute.

`create_constraint`(*variable_search_function: Callable[35][[OXObject], bool[36]] = None*, *weight_calculation_function: Callable[37][[UUID[38], Self[39]], float[40] | int[41] | Fraction[42]] = None*, *variables: list[43][UUID[44]] = None*, *weights: list[45][float[46] | int[47] | Fraction[48]] = None*, *operator: RelationalOperators (page 129) = RelationalOperators.LESS_THAN_EQUAL*, *value: float[49] | int[50] = None*, *name: str[51] = None*)

Create a linear constraint for the optimization problem.

Creates a linear constraint of the form: w1*x1 + w2*x2 + … + wn*xn {operator} value

Variables and weights can be specified either directly or through search and calculation functions.

Parameters

- `variable_search_function` (`Callable[[OXObject], bool[52]], optional`) – Function to search for variables in the problem. If provided, variables parameter must be None.
- `weight_calculation_function` (`Callable[[UUID, Self], float[53] | int[54]], optional`) – Function to calculate weights for each variable. If provided, weights parameter must be None.
- `variables` (`list[55][UUID], optional`) – List of variable IDs to include in the constraint. If provided, variable_search_function must be None.
- `weights` (`list[56][float[57] | int[58]], optional`) – List of weights for each variable. If provided, weight_calculation_function must be None.
- `operator` (`RelationalOperators` (page 129), `optional`) – Relational operator for the constraint. Defaults to LESS_THAN_EQUAL.
- `value` (`float[59] | int[60], optional`) – Right-hand side value of the constraint.
- `name` (`str[61], optional`) – A descriptive name for the constraint. If None, an auto-generated name will be created based on the constraint terms.

Raises

`OXception` – If parameter combinations are invalid (see _check_parameters).

Examples

```
>>> # Using direct variable and weight specification
>>> problem.create_constraint(
...     variables=[var1.id, var2.id, var3.id],
...     weights=[1, 2, 3],
...     operator=RelationalOperators.LESS_THAN_EQUAL,
...     value=100
... )
>>>
>>> # Using search and calculation functions
>>> problem.create_constraint(
...     variable_search_function=lambda v: v.name.startswith("x"),
...     weight_calculation_function=lambda v, p: 1.0,
...     operator=RelationalOperators.EQUAL,
...     value=1
... )
```

Note

The constraint is automatically added to the problem's constraint list. Exactly one of variable_search_function/variables and one of weight_calculation_function/weights must be provided.

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', db: ~data.OXDatabase.OXDatabase = <factory>, variables: ~variables.OXVariableSet.OXVariableSet = <factory>, constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>, specials: list[~constraints.OXSpecialConstraints.OXSpecialConstraint] = <factory>, constraints_in_special_constraints: list[~uuid.UUID] = <factory>*) → None[62]

[5] https://docs.python.org/3/library/stdtypes.html#list
[6] https://docs.python.org/3/library/stdtypes.html#list
[7] https://docs.python.org/3/library/stdtypes.html#list
[8] https://docs.python.org/3/library/stdtypes.html#list
[9] https://docs.python.org/3/library/stdtypes.html#list
[10] https://docs.python.org/3/library/uuid.html#uuid.UUID
[11] https://docs.python.org/3/library/stdtypes.html#str
[12] https://docs.python.org/3/library/stdtypes.html#str
[13] https://docs.python.org/3/library/functions.html#float
[14] https://docs.python.org/3/library/functions.html#int
[15] https://docs.python.org/3/library/functions.html#float
[16] https://docs.python.org/3/library/functions.html#int
[17] https://docs.python.org/3/library/stdtypes.html#str
[18] https://docs.python.org/3/library/stdtypes.html#str
[19] https://docs.python.org/3/library/functions.html#float
[20] https://docs.python.org/3/library/functions.html#int
[21] https://docs.python.org/3/library/functions.html#float
[22] https://docs.python.org/3/library/functions.html#int
[23] https://docs.python.org/3/library/stdtypes.html#str
[24] https://docs.python.org/3/library/stdtypes.html#str
[25] https://docs.python.org/3/library/functions.html#float
[26] https://docs.python.org/3/library/functions.html#int
[27] https://docs.python.org/3/library/functions.html#float
[28] https://docs.python.org/3/library/functions.html#int
[29] https://docs.python.org/3/library/stdtypes.html#str
[30] https://docs.python.org/3/library/stdtypes.html#str
[31] https://docs.python.org/3/library/functions.html#float
[32] https://docs.python.org/3/library/functions.html#int
[33] https://docs.python.org/3/library/functions.html#float
[34] https://docs.python.org/3/library/functions.html#int
[35] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[36] https://docs.python.org/3/library/functions.html#bool
[37] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[38] https://docs.python.org/3/library/uuid.html#uuid.UUID
[39] https://docs.python.org/3/library/typing.html#typing.Self
[40] https://docs.python.org/3/library/functions.html#float
[41] https://docs.python.org/3/library/functions.html#int
[42] https://docs.python.org/3/library/fractions.html#fractions.Fraction
[43] https://docs.python.org/3/library/stdtypes.html#list
[44] https://docs.python.org/3/library/uuid.html#uuid.UUID
[45] https://docs.python.org/3/library/stdtypes.html#list
[46] https://docs.python.org/3/library/functions.html#float
[47] https://docs.python.org/3/library/functions.html#int
[48] https://docs.python.org/3/library/fractions.html#fractions.Fraction
[49] https://docs.python.org/3/library/functions.html#float
[50] https://docs.python.org/3/library/functions.html#int
[51] https://docs.python.org/3/library/stdtypes.html#str
[52] https://docs.python.org/3/library/functions.html#bool
[53] https://docs.python.org/3/library/functions.html#float
[54] https://docs.python.org/3/library/functions.html#int
[55] https://docs.python.org/3/library/stdtypes.html#list
[56] https://docs.python.org/3/library/stdtypes.html#list
[57] https://docs.python.org/3/library/functions.html#float
[58] https://docs.python.org/3/library/functions.html#int
[59] https://docs.python.org/3/library/functions.html#float
[60] https://docs.python.org/3/library/functions.html#int
[61] https://docs.python.org/3/library/stdtypes.html#str
[62] https://docs.python.org/3/library/constants.html#None

`class problem.OXLPProblem(`*id: ~uuid.UUID = <factory>, class_name: str = '', db: ~data.OXDatabase.OXDatabase = <factory>, variables: ~variables.OXVariableSet.OXVariableSet = <factory>, constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>, specials: list[~constraints.OXSpecialConstraints.OXSpecialConstraint] = <factory>, constraints_in_special_constraints: list[~uuid.UUID] = <factory>, objective_function: ~constraints.OXpression.OXpression = <factory>, objective_type: ~problem.OXProblem.ObjectiveType = ObjectiveType.MINIMIZE*`)`

Bases: `OXCSPProblem` (page 81)

Linear Programming Problem class.

This class extends OXCSPProblem to add support for linear programming by introducing an objective function that can be minimized or maximized.

`objective_function`

The objective function to optimize.

Type

OXpression (page 114)

`objective_type`

Whether to minimize or maximize the objective.

Type

ObjectiveType (page 97)

`db`

Inherited from OXCSPProblem.

Type

OXDatabase (page 207)

`variables`

Inherited from OXCSPProblem.

Type

OXVariableSet (page 143)

`constraints`

Inherited from OXCSPProblem.

Type

list[63][*OXConstraint* (page 106)]

`specials`

Inherited from OXCSPProblem.

Type

list[64][*OXSpecialConstraint* (page 123)]

Examples

```
>>> problem = OXLPProblem()
>>> problem.create_decision_variable("x", lower_bound=0, upper_bound=10)
>>> problem.create_decision_variable("y", lower_bound=0, upper_bound=10)
>>> problem.create_constraint(
...     variables=[problem.variables[0].id, problem.variables[1].id],
...     weights=[1, 1],
...     operator=RelationalOperators.LESS_THAN_EQUAL,
...     value=15
... )
>>> problem.create_objective_function(
...     variables=[problem.variables[0].id, problem.variables[1].id],
...     weights=[3, 2],
...     objective_type=ObjectiveType.MAXIMIZE
... )
```

See also

`OXCSPProblem` (page 81): Base constraint satisfaction problem class. `OXGPProblem` (page 92): Goal Programming extension of LP.

`objective_function: OXpression (page 114)`

`objective_type: ObjectiveType (page 97) = 'minimize'`

`create_objective_function`(*variable_search_function: Callable*[65]*[[OXObject], bool*[66]*] = None, weight_calculation_function: Callable*[67]*[[OXVariable (page 134), Self*[68]*], float*[69] | *int*[70]*] = None, variables: list*[71]*[UUID*[72]*] = None, weights: list*[73]*[float*[74] | *int*[75]*] = None, objective_type: ObjectiveType (page 97) = ObjectiveType.MINIMIZE*)

Create an objective function for the linear programming problem.

Creates an objective function of the form: {minimize|maximize} w1*x1 + w2*x2 + … + wn*xn

Variables and weights can be specified either directly or through search and calculation functions.

Parameters

- `variable_search_function` (`Callable[[OXObject], bool`[76]`], optional`) – Function to search for variables in the problem. If provided, variables parameter must be None.
- `weight_calculation_function` (`Callable[[OXVariable` (page 134), `Self], float`[77] `| int`[78]`], optional`) – Function to calculate weights for each variable. If provided, weights parameter must be None.
- `variables` (`list`[79]`[UUID], optional`) – List of variable IDs to include in the objective function. If provided, variable_search_function must be None.

- `weights` (`list`[80][`float`[81] | `int`[82]], `optional`) – List of weights for each variable. If provided, weight_calculation_function must be None.
- `objective_type` (`ObjectiveType` (page 97), `optional`) – Whether to minimize or maximize the objective function. Defaults to MINIMIZE.

Examples

```
>>> # Maximize profit: 3*x + 2*y
>>> problem.create_objective_function(
...     variables=[x_id, y_id],
...     weights=[3, 2],
...     objective_type=ObjectiveType.MAXIMIZE
... )
>>>
>>> # Minimize cost using search function
>>> problem.create_objective_function(
...     variable_search_function=lambda v: "cost" in v.name,
...     weight_calculation_function=lambda v, p: v.get_cost(),
...     objective_type=ObjectiveType.MINIMIZE
... )
```

Note

This method sets both the objective_function and objective_type attributes of the problem.

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', db: ~data.OXDatabase.OXDatabase = <factory>, variables: ~variables.OXVariableSet.OXVariableSet = <factory>, constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>, specials: list[~constraints.OXSpecialConstraints.OXSpecialConstraint] = <factory>, constraints_in_special_constraints: list[~uuid.UUID] = <factory>, objective_function: ~constraints.OXpression.OXpression = <factory>, objective_type: ~problem.OXProblem.ObjectiveType = ObjectiveType.MINIMIZE*) → None[83] `class problem.OXGPProblem(`*id: ~uuid.UUID = <factory>, class_name: str = ”, db: ~data.OXDatabase.OXDatabase = <factory>, variables: ~variables.OXVariableSet.OXVariableSet = <factory>, constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>, specials: list[~constraints.OXSpecialConstraints.OXSpecialConstraint] = <factory>, constraints_in_special_constraints: list[~uuid.UUID] = <factory>, objective_function: ~constraints.OXpression.OXpression = <factory>, objective_type: ~problem.OXProblem.ObjectiveType = ObjectiveType.MINIMIZE, goal_constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>*`)`

Bases: `OXLPProblem` (page 88)

Goal Programming Problem class.

This class extends OXLPProblem to add support for goal programming by introducing goal constraints with deviation variables. Goal programming is used when there are multiple conflicting objectives that cannot be simultaneously satisfied.

`goal_constraints`

List of goal constraints with deviation variables.

Type

list[84][*OXGoalConstraint* (page 110)]

`objective_function`

Inherited from OXLPProblem.

Type

OXpression (page 114)

`objective_type`

Inherited from OXLPProblem.

Type

ObjectiveType (page 97)

[63] https://docs.python.org/3/library/stdtypes.html#list
[64] https://docs.python.org/3/library/stdtypes.html#list
[65] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[66] https://docs.python.org/3/library/functions.html#bool
[67] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[68] https://docs.python.org/3/library/typing.html#typing.Self
[69] https://docs.python.org/3/library/functions.html#float
[70] https://docs.python.org/3/library/functions.html#int
[71] https://docs.python.org/3/library/stdtypes.html#list
[72] https://docs.python.org/3/library/uuid.html#uuid.UUID
[73] https://docs.python.org/3/library/stdtypes.html#list
[74] https://docs.python.org/3/library/functions.html#float
[75] https://docs.python.org/3/library/functions.html#int
[76] https://docs.python.org/3/library/functions.html#bool
[77] https://docs.python.org/3/library/functions.html#float
[78] https://docs.python.org/3/library/functions.html#int
[79] https://docs.python.org/3/library/stdtypes.html#list
[80] https://docs.python.org/3/library/stdtypes.html#list
[81] https://docs.python.org/3/library/functions.html#float
[82] https://docs.python.org/3/library/functions.html#int
[83] https://docs.python.org/3/library/constants.html#None

`db`

Inherited from OXCSPProblem.

Type
OXDatabase (page 207)

`variables`

Inherited from OXCSPProblem.

Type
OXVariableSet (page 143)

`constraints`

Inherited from OXCSPProblem.

Type
list[85][*OXConstraint* (page 106)]

`specials`

Inherited from OXCSPProblem.

Type
list[86][*OXSpecialConstraint* (page 123)]

Examples

```
>>> problem = OXGPProblem()
>>> problem.create_decision_variable("x", lower_bound=0, upper_bound=10)
>>> problem.create_decision_variable("y", lower_bound=0, upper_bound=10)
>>> # Create a goal constraint: aim for x + y = 8
>>> problem.create_goal_constraint(
...     variables=[problem.variables[0].id, problem.variables[1].id],
...     weights=[1, 1],
...     operator=RelationalOperators.EQUAL,
...     value=8
... )
>>> # The objective function minimizes undesired deviations
>>> problem.create_objective_function()
```

See also

`OXLPProblem` (page 88): Base linear programming problem class. `OXCSPProblem` (page 81): Base constraint satisfaction problem class. `constraints.OXConstraint.OXGoalConstraint` (page 110): Goal constraint with deviation variables.

`goal_constraints: OXConstraintSet (page 112)`

`create_goal_constraint`(*variable_search_function: Callable[87][[OXObject], bool[88]] = None, weight_calculation_function: Callable[89][[UUID[90], Self[91]], float[92] | int[93] | Fraction[94]] = None, variables: list[95][UUID[96]] = None, weights: list[97][float[98] | int[99]] = None, operator: RelationalOperators (page 129) = RelationalOperators.LESS_THAN_EQUAL, value: float[100] | int[101] = None, name: str[102] = None*)

Create a goal constraint for the goal programming problem.

Creates a goal constraint with associated positive and negative deviation variables. Goal constraints represent targets that the problem should try to achieve, but may not be strictly satisfied.

The constraint is of the form: w1*x1 + w2*x2 + … + wn*xn + d- - d+ {operator} value

Where d- and d+ are negative and positive deviation variables respectively.

Parameters

- `variable_search_function` (`Callable[[OXObject], bool`[103]`], optional`) – Function to search for variables in the problem. If provided, variables parameter must be None.
- `weight_calculation_function` (`Callable[[OXVariable` (page 134), `Self], float`[104] `|` `int`[105]`], optional`) – Function to calculate weights for each variable. If provided, weights parameter must be None.
- `variables` (`list`[106]`[UUID], optional`) – List of variable IDs to include in the constraint. If provided, variable_search_function must be None.
- `weights` (`list`[107]`[float`[108] `|` `int`[109]`], optional`) – List of weights for each variable. If provided, weight_calculation_function must be None.
- `operator` (`RelationalOperators` (page 129), `optional`) – Relational operator for the constraint. Defaults to LESS_THAN_EQUAL.
- `value` (`float`[110] `|` `int`[111]`, optional`) – Target value for the goal constraint.

Examples

```
>>> # Goal: total production should be around 1000 units
>>> problem.create_goal_constraint(
...     variables=[prod1_id, prod2_id, prod3_id],
...     weights=[1, 1, 1],
...     operator=RelationalOperators.EQUAL,
...     value=1000
... )
>>>
>>> # Goal: minimize resource usage below 500
>>> problem.create_goal_constraint(
...     variable_search_function=lambda v: ”resource” in v.name,
...     weight_calculation_function=lambda v, p: v.usage_rate,
...     operator=RelationalOperators.LESS_THAN_EQUAL,
...     value=500
... )
```

Note

This method creates a regular constraint first, then converts it to a goal constraint with deviation variables. The goal constraint is added to the goal_constraints list.

`create_objective_function`(*variable_search_function: Callable[112][[OXObject], bool[113]] = None*, *weight_calculation_function: Callable[114][[OXVariable (page 134), Self[115]], float[116] | int[117]] = None*, *variables: list[118][UUID[119]] = None*, *weights: list[120][float[121] | int[122]] = None*, *objective_type: ObjectiveType (page 97) = ObjectiveType.MINIMIZE*)

Create an objective function for the goal programming problem.

Creates an objective function that minimizes the sum of undesired deviation variables from all goal constraints. This is the standard approach in goal programming where the objective is to minimize deviations from goals.

Parameters

- `variable_search_function` (`Callable[[OXObject], bool[123]], optional`) – Not used in goal programming. Included for interface consistency.
- `weight_calculation_function` (`Callable[[OXVariable` (page 134), `Self], float[124] | int[125], optional`) – Not used in goal programming. Included for interface consistency.
- `variables` (`list[126][UUID], optional`) – Not used in goal programming. Included for interface consistency.
- `weights` (`list[127][float[128] | int[129]], optional`) – Not used in goal programming. Included for interface consistency.
- `objective_type` (`ObjectiveType` (page 97), `optional`) – Not used in goal programming. Always set to MINIMIZE. Included for interface consistency.

Examples

```
>>> # Create goal constraints first
>>> problem.create_goal_constraint(
...     variables=[x_id, y_id],
...     weights=[1, 1],
...     operator=RelationalOperators.EQUAL,
...     value=100
... )
>>>
>>> # Create objective function to minimize deviations
>>> problem.create_objective_function()
```

Note

This method automatically collects all undesired deviation variables from existing goal constraints and creates an objective function that minimizes their sum. The objective is always set to MINIMIZE. Parameters are ignored as the objective is automatically determined from goal constraints.

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = ”, db: ~data.OXDatabase.OXDatabase = <factory>, variables: ~variables.OXVariableSet.OXVariableSet = <factory>, constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>, specials: list[~constraints.OXSpecialConstraints.OXSpecialConstraint] = <factory>, constraints_in_special_constraints: list[~uuid.UUID] = <factory>, objective_function: ~constraints.OXpression.OXpression = <factory>, objective_type: ~problem.OXProblem.ObjectiveType = ObjectiveType.MINIMIZE, goal_constraints: ~constraints.OXConstraintSet.OXConstraintSet = <factory>*) → None[130]

9.6.2 Enumerations

`class problem.ObjectiveType(*values)`

Enumeration of objective types for optimization problems.

`MINIMIZE`

Minimize the objective function.

Type

str[131]

`MAXIMIZE`

Maximize the objective function.

[84] https://docs.python.org/3/library/stdtypes.html#list
[85] https://docs.python.org/3/library/stdtypes.html#list
[86] https://docs.python.org/3/library/stdtypes.html#list
[87] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[88] https://docs.python.org/3/library/functions.html#bool
[89] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[90] https://docs.python.org/3/library/uuid.html#uuid.UUID
[91] https://docs.python.org/3/library/typing.html#typing.Self
[92] https://docs.python.org/3/library/functions.html#float
[93] https://docs.python.org/3/library/functions.html#int
[94] https://docs.python.org/3/library/fractions.html#fractions.Fraction
[95] https://docs.python.org/3/library/stdtypes.html#list
[96] https://docs.python.org/3/library/uuid.html#uuid.UUID
[97] https://docs.python.org/3/library/stdtypes.html#list
[98] https://docs.python.org/3/library/functions.html#float
[99] https://docs.python.org/3/library/functions.html#int
[100] https://docs.python.org/3/library/functions.html#float
[101] https://docs.python.org/3/library/functions.html#int
[102] https://docs.python.org/3/library/stdtypes.html#str
[103] https://docs.python.org/3/library/functions.html#bool
[104] https://docs.python.org/3/library/functions.html#float
[105] https://docs.python.org/3/library/functions.html#int
[106] https://docs.python.org/3/library/stdtypes.html#list
[107] https://docs.python.org/3/library/stdtypes.html#list
[108] https://docs.python.org/3/library/functions.html#float
[109] https://docs.python.org/3/library/functions.html#int
[110] https://docs.python.org/3/library/functions.html#float
[111] https://docs.python.org/3/library/functions.html#int
[112] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[113] https://docs.python.org/3/library/functions.html#bool
[114] https://docs.python.org/3/library/collections.abc.html#collections.abc.Callable
[115] https://docs.python.org/3/library/typing.html#typing.Self
[116] https://docs.python.org/3/library/functions.html#float
[117] https://docs.python.org/3/library/functions.html#int
[118] https://docs.python.org/3/library/stdtypes.html#list
[119] https://docs.python.org/3/library/uuid.html#uuid.UUID
[120] https://docs.python.org/3/library/stdtypes.html#list
[121] https://docs.python.org/3/library/functions.html#float
[122] https://docs.python.org/3/library/functions.html#int
[123] https://docs.python.org/3/library/functions.html#bool
[124] https://docs.python.org/3/library/functions.html#float
[125] https://docs.python.org/3/library/functions.html#int
[126] https://docs.python.org/3/library/stdtypes.html#list
[127] https://docs.python.org/3/library/stdtypes.html#list
[128] https://docs.python.org/3/library/functions.html#float
[129] https://docs.python.org/3/library/functions.html#int
[130] https://docs.python.org/3/library/constants.html#None

Type
str[132]

Available objective types:

- `MINIMIZE` - Minimize the objective function
- `MAXIMIZE` - Maximize the objective function

`MINIMIZE = 'minimize'`

`MAXIMIZE = 'maximize'`

`class problem.SpecialConstraintType(*values)`

Enumeration of special constraint types supported by the framework.

This enumeration defines the types of special (non-linear) constraints that can be created in optimization problems.

`MultiplicativeEquality`

Constraint for variable multiplication.

Type
str[133]

`DivisionEquality`

Constraint for integer division operations.

Type
str[134]

`ModulusEquality`

Constraint for modulo operations.

Type
str[135]

`SummationEquality`

Constraint for variable summation.

Type
str[136]

`ConditionalConstraint`

Constraint for conditional logic.

Type
str[137]

Available special constraint types:

- `MultiplicativeEquality` - Multiplication: result = var1 * var2 * … * varN
- `DivisionEquality` - Integer division: result = var // divisor
- `ModulusEquality` - Modulo operation: result = var % divisor
- `SummationEquality` - Summation: result = var1 + var2 + … + varN

[131] https://docs.python.org/3/library/stdtypes.html#str
[132] https://docs.python.org/3/library/stdtypes.html#str

- `ConditionalConstraint` - Conditional logic: if condition then constraint1 else constraint2

`MultiplicativeEquality = 'MultiplicativeEquality'`

`DivisionEquality = 'DivisionEquality'`

`ModulusEquality = 'ModulusEquality'`

`SummationEquality = 'SummationEquality'`

`ConditionalConstraint = 'ConditionalConstraint'`

9.6.3 Examples

Creating a Constraint Satisfaction Problem

```
from problem import OXCSPProblem
from constraints import RelationalOperators

# Create CSP
csp = OXCSPProblem()

# Add variables
csp.create_decision_variable("x1", lower_bound=0, upper_bound=10)
csp.create_decision_variable("x2", lower_bound=0, upper_bound=10)

# Add constraint: x1 + x2 <= 15
csp.create_constraint(
    variables=[csp.variables[0].id, csp.variables[1].id],
    weights=[1, 1],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=15
)
```

Creating a Linear Programming Problem

```
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators

# Create LP problem
lp = OXLPProblem()

# Add variables
lp.create_decision_variable("production_a", lower_bound=0, upper_bound=1000)
lp.create_decision_variable("production_b", lower_bound=0, upper_bound=1000)
```

(continues on next page)

[133] https://docs.python.org/3/library/stdtypes.html#str
[134] https://docs.python.org/3/library/stdtypes.html#str
[135] https://docs.python.org/3/library/stdtypes.html#str
[136] https://docs.python.org/3/library/stdtypes.html#str
[137] https://docs.python.org/3/library/stdtypes.html#str

(continued from previous page)

```
# Add resource constraint
lp.create_constraint(
    variables=[lp.variables[0].id, lp.variables[1].id],
    weights=[2, 3],  # Resource consumption per unit
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=500,  # Available resources
    name="Resource limitation"
)

# Set objective: maximize profit
lp.create_objective_function(
    variables=[lp.variables[0].id, lp.variables[1].id],
    weights=[10, 15],  # Profit per unit
    objective_type=ObjectiveType.MAXIMIZE
)
```

Creating a Goal Programming Problem

```
from problem import OXGPProblem
from constraints import RelationalOperators

# Create GP problem
gp = OXGPProblem()

# Add variables
gp.create_decision_variable("workers_day", lower_bound=0, upper_bound=100)
gp.create_decision_variable("workers_night", lower_bound=0, upper_bound=100)

# Add goal constraint: target 80 total workers
gp.create_goal_constraint(
    variables=[gp.variables[0].id, gp.variables[1].id],
    weights=[1, 1],
    operator=RelationalOperators.EQUAL,
    value=80,
    name="Target workforce size"
)

# Add goal constraint: prefer balanced shifts
gp.create_goal_constraint(
    variables=[gp.variables[0].id, gp.variables[1].id],
    weights=[1, -1],  # Difference between shifts
    operator=RelationalOperators.EQUAL,
    value=0,   # Equal shifts
    name="Balanced shift allocation"
)

# Create objective function to minimize deviations
gp.create_objective_function()
```

Working with Database Objects

```python
from problem import OXLPProblem
from data import OXData, OXDatabase

# Create data objects
bus1 = OXData()
bus1.capacity = 50
bus1.cost_per_km = 2.5

bus2 = OXData()
bus2.capacity = 40
bus2.cost_per_km = 2.0

route1 = OXData()
route1.distance = 25
route1.demand = 35

route2 = OXData()
route2.distance = 30
route2.demand = 45

# Create problem
problem = OXLPProblem()

# Add data to database
problem.db.add_object(bus1)
problem.db.add_object(bus2)
problem.db.add_object(route1)
problem.db.add_object(route2)

# Create variables from database objects using custom types
class Bus:
    pass

class Route:
    pass

# Create variables for all bus-route combinations
problem.create_variables_from_db(
    Bus, Route,
    var_name_template="assign_{bus_id}_{route_id}",
    var_description_template="Assign bus {bus_id} to route {route_id}",
    upper_bound=1,
    lower_bound=0
)
```

Special Constraints

```python
from problem import OXLPProblem, SpecialConstraintType

problem = OXLPProblem()

# Create variables
problem.create_decision_variable("x", lower_bound=0, upper_bound=100)
problem.create_decision_variable("y", lower_bound=0, upper_bound=100)

# Create multiplication constraint: z = x * y
mult_constraint = problem.create_special_constraint(
    constraint_type=SpecialConstraintType.MultiplicativeEquality,
    input_variables=[problem.variables[0], problem.variables[1]]
)

# Create division constraint: w = x // 5
div_constraint = problem.create_special_constraint(
    constraint_type=SpecialConstraintType.DivisionEquality,
    input_variable=[problem.variables[0]],
    divisor=5
)

# Create modulo constraint: r = x % 7
mod_constraint = problem.create_special_constraint(
    constraint_type=SpecialConstraintType.ModulusEquality,
    input_variable=[problem.variables[0]],
    divisor=7
)
```

Functional Constraint Creation

```python
from problem import OXLPProblem
from constraints import RelationalOperators

problem = OXLPProblem()

# Create variables with meaningful names
for i in range(5):
    problem.create_decision_variable(
        f"production_{i}",
        lower_bound=0,
        upper_bound=100
    )

# Create constraint using search function
problem.create_constraint(
    variable_search_function=lambda v: v.name.startswith("production"),
    weight_calculation_function=lambda var_id, prob: 1.0,  # Equal weights
    operator=RelationalOperators.LESS_THAN_EQUAL,
```

(continues on next page)

(continued from previous page)

```
    value=300,
    name="Total production capacity"
)

# Create objective using search function
problem.create_objective_function(
    variable_search_function=lambda v: v.name.startswith("production"),
    weight_calculation_function=lambda var_id, prob: 10.0,  # Profit per unit
    objective_type=ObjectiveType.MAXIMIZE
)
```

Advanced Goal Programming

```
from problem import OXGPProblem
from constraints import RelationalOperators

# Multi-criteria optimization problem
problem = OXGPProblem()

# Create variables for different departments
problem.create_decision_variable("dept_a_budget", lower_bound=0, upper_
↪bound=1000000)
problem.create_decision_variable("dept_b_budget", lower_bound=0, upper_
↪bound=1000000)
problem.create_decision_variable("dept_c_budget", lower_bound=0, upper_
↪bound=1000000)

# Goal 1: Total budget should be around $2M
problem.create_goal_constraint(
    variables=[v.id for v in problem.variables],
    weights=[1, 1, 1],
    operator=RelationalOperators.EQUAL,
    value=2000000,
    name="Total budget target"
)

# Goal 2: Department A should get at least 40% of total budget
problem.create_goal_constraint(
    variables=[v.id for v in problem.variables],
    weights=[1, -0.4, -0.4],  # dept_a - 0.4*(dept_b + dept_c)
    operator=RelationalOperators.GREATER_THAN_EQUAL,
    value=0,
    name="Department A minimum share"
)

# Goal 3: Departments B and C should have similar budgets
problem.create_goal_constraint(
    variables=[problem.variables[1].id, problem.variables[2].id],
    weights=[1, -1],  # dept_b - dept_c
```

(continues on next page)

(continued from previous page)

```
    operator=RelationalOperators.EQUAL,
    value=0,
    name="Balanced B and C budgets"
)

# Create objective to minimize undesired deviations
problem.create_objective_function()
```

Complex Special Constraints

```
from problem import OXCSPProblem, SpecialConstraintType

problem = OXCSPProblem()

# Create variables for a scheduling problem
problem.create_decision_variable("task_duration", lower_bound=1, upper_bound=10)
problem.create_decision_variable("num_workers", lower_bound=1, upper_bound=5)
problem.create_decision_variable("efficiency_factor", lower_bound=1, upper_
↪bound=3)

# Total work = duration * workers * efficiency
work_constraint = problem.create_special_constraint(
    constraint_type=SpecialConstraintType.MultiplicativeEquality,
    input_variables=problem.variables.objects  # All variables
)

# Create summation constraint for resource allocation
for i in range(3):
    problem.create_decision_variable(f"resource_{i}", lower_bound=0, upper_
↪bound=100)

# Total resources used
resource_sum = problem.create_special_constraint(
    constraint_type=SpecialConstraintType.SummationEquality,
    input_variables=lambda v: v.name.startswith("resource")
)
```

Working with Variable Templates

```
from problem import OXLPProblem
from data import OXData, OXDatabase

# Create a transportation problem
problem = OXLPProblem()

# Create supply and demand data
supply_points = []
for i in range(3):
```

(continues on next page)

(continued from previous page)

```
    point = OXData()
    point.location = f"Factory_{i}"
    point.supply_capacity = (i + 1) * 100
    problem.db.add_object(point)
    supply_points.append(point)

demand_points = []
for i in range(4):
    point = OXData()
    point.location = f"Customer_{i}"
    point.demand = (i + 1) * 50
    problem.db.add_object(point)
    demand_points.append(point)

# Create classes for database query
class SupplyPoint:
    pass

class DemandPoint:
    pass

# Create transportation variables
problem.create_variables_from_db(
    SupplyPoint, DemandPoint,
    var_name_template="ship_{supplypoint_location}_{demandpoint_location}",
    var_description_template="Shipment from {supplypoint_location} to
↪{demandpoint_location}",
    upper_bound=1000,
    lower_bound=0
)

print(f"Created {len(problem.variables)} transportation variables")
```

9.6.4 See Also

- *Constraints Module* (page 105) - Constraint definitions and operators
- *Variables Module* (page 134) - Variable management and types
- *Data Module* (page 205) - Database objects and scenario management
- *Solvers Module* (page 163) - Solver interfaces and implementations
- *Examples* (page 32) - Complete example problems

9.7 Constraints Module

The constraints module provides comprehensive constraint definition capabilities for optimization problems. It includes linear constraints, goal programming constraints, special non-linear constraints, and mathematical expressions.

9.7.1 Constraint Classes

Linear Constraints

`class constraints.OXConstraint(`*id: ~uuid.UUID = <factory>, class_name: str = ”, expression: ~constraints.OXpression.OXpression = <factory>, relational_operator: ~constraints.OXConstraint.RelationalOperators = RelationalOperators.EQUAL, rhs: float | int = 0, name: str = ”, active_scenario: str = ’Default’, scenarios: dict[str, dict[str, ~typing.Any]] = <factory>*)

Bases: `OXObject`

A constraint in an optimization problem with scenario support.

A constraint represents a relationship between an expression and a value, such as “2x + 3y <= 10”. This class supports multiple scenarios, allowing different constraint parameters (RHS values, names, operators) to be defined for different optimization scenarios.

The scenario system enables sensitivity analysis and what-if modeling by maintaining multiple constraint configurations within the same constraint object.

`expression`

The left-hand side of the constraint.

Type

OXpression (page 114)

`relational_operator`

The operator (>, >=, =, <, <=).

Type

RelationalOperators (page 129)

`rhs`

The right-hand side value.

Type

float[138] | int[139]

`name`

A descriptive name for the constraint.

Type

str[140]

`active_scenario`

The name of the currently active scenario.

Type

str[141]

`scenarios`

Dictionary mapping scenario names to dictionaries of attribute values for that scenario.

Type
dict[142][str[143], dict[144][str[145], Any]]

Examples

Basic constraint creation:

```
>>> from constraints.OXpression import OXpression
>>> expr = OXpression(variables=[x.id, y.id], weights=[2, 3])
>>> constraint = OXConstraint(
...     expression=expr,
...     relational_operator=RelationalOperators.LESS_THAN_EQUAL,
...     rhs=10,
...     name="Capacity constraint"
... )
```

Scenario-based constraint management:

```
>>> # Create constraint with base values
>>> constraint = OXConstraint(
...     expression=expr,
...     relational_operator=RelationalOperators.LESS_THAN_EQUAL,
...     rhs=100,
...     name="Production capacity"
... )
>>>
>>> # Create scenarios with different RHS values
>>> constraint.create_scenario("High_Capacity", rhs=150, name="High
↪capacity scenario")
>>> constraint.create_scenario("Low_Capacity", rhs=80, name="Reduced
↪capacity scenario")
>>>
>>> # Switch between scenarios
>>> print(constraint.rhs)  # 100 (Default scenario)
>>>
>>> constraint.active_scenario = "High_Capacity"
>>> print(constraint.rhs)  # 150
>>> print(constraint.name)  # "High capacity scenario"
>>>
>>> constraint.active_scenario = "Low_Capacity"
>>> print(constraint.rhs)  # 80
```

`expression: OXpression (page 114)`

`relational_operator: RelationalOperators (page 129) = '='`

`rhs: float`[146] `| int`[147] `= 0`

`name: str`[148] `= ''`

`active_scenario: str`[149] `= 'Default'`

`scenarios: dict`[150]`[str`[151]`, dict`[152]`[str`[153]`, Any`[154]`]]` `__getattribute__`(*item*)

Custom attribute access that checks the active scenario first.

When an attribute is accessed, this method first checks if it exists in the active scenario, and if not, falls back to the object's own attribute. This enables transparent scenario switching for constraint parameters.

Parameters

`item` (`str`[155]) – The name of the attribute to access.

Returns

The value of the attribute in the active scenario, or the object's own attribute if not found in the active scenario.

Return type

Any

Examples

```
>>> constraint = OXConstraint(rhs=100)
>>> constraint.create_scenario("High_RHS", rhs=150)
>>> print(constraint.rhs)  # 100 (Default)
>>> constraint.active_scenario = "High_RHS"
>>> print(constraint.rhs)  # 150 (from scenario)
```

`create_scenario`(*scenario_name: str*[156], ***kwargs*)

Create a new scenario with the specified constraint attribute values.

If the "Default" scenario doesn't exist yet, it is created first, capturing the constraint's current attribute values. This enables systematic scenario-based analysis while preserving the original constraint configuration.

Parameters

- `scenario_name` (`str`[157]) – The name of the new scenario.
- `**kwargs` – Constraint attribute-value pairs for the new scenario. Common attributes include: - rhs (float | int): Right-hand side value - name (str): Constraint name for this scenario - relational_operator (RelationalOperators): Constraint operator

Raises

`OXception` – If an attribute in kwargs doesn't exist in the constraint object.

Examples

Creating RHS scenarios for sensitivity analysis:

```
>>> constraint = OXConstraint(
...     expression=expr,
...     relational_operator=RelationalOperators.LESS_THAN_EQUAL,
...     rhs=100,
...     name="Base capacity"
... )
```

(continues on next page)

(continued from previous page)

```
>>>
>>> # Create scenarios with different RHS values
>>> constraint.create_scenario("High_Demand", rhs=150, name="Peak␣
↪capacity")
>>> constraint.create_scenario("Low_Demand", rhs=75, name="Reduced␣
↪capacity")
>>> constraint.create_scenario("Critical", rhs=200, name="Emergency␣
↪capacity")
>>>
>>> # Switch scenarios and access values
>>> constraint.active_scenario = "High_Demand"
>>> print(f"RHS: {constraint.rhs}, Name: {constraint.name}")
>>> # Output: RHS: 150, Name: Peak capacity
```

Creating operator scenarios for constraint type analysis:

```
>>> constraint.create_scenario("Equality",
...     relational_operator=RelationalOperators.EQUAL,
...     name="Exact capacity requirement"
... )
>>> constraint.create_scenario("Lower_Bound",
...     relational_operator=RelationalOperators.GREATER_THAN_EQUAL,
...     name="Minimum capacity requirement"
... )
```

`reverse()`

Reverse the relational operator of the constraint.

This method changes the relational operator to its opposite: - GREATER_THAN becomes LESS_THAN - GREATER_THAN_EQUAL becomes LESS_THAN_EQUAL - EQUAL remains EQUAL - LESS_THAN becomes GREATER_THAN - LESS_THAN_EQUAL becomes GREATER_THAN_EQUAL

Returns
A new constraint with the reversed operator.

Return type
OXConstraint (page 106)

`property rhs_numerator`

Get the numerator of the right-hand side as a fraction.

Returns
The numerator of the right-hand side.

Return type
int[158]

`property rhs_denominator`

Get the denominator of the right-hand side as a fraction.

Returns
The denominator of the right-hand side.

Return type
int[159]

`to_goal`(*upper_bound: int*[160] | *float*[161] | *Fraction*[162] *= 100*) → *OXGoalConstraint* (page 110)

Convert this constraint to a goal constraint for goal programming.

The conversion sets the relational operator to EQUAL and sets the desired deviation variables based on the original operator.

Returns
A new goal constraint based on this constraint.

Return type
OXGoalConstraint (page 110)

See also

`OXGoalConstraint` (page 110)

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = ”, expression: ~constraints.OXpression.OXpression = <factory>, relational_operator: ~constraints.OXConstraint.RelationalOperators = RelationalOperators.EQUAL, rhs: float | int = 0, name: str = ”, active_scenario: str = 'Default', scenarios: dict[str, dict[str, ~typing.Any]] = <factory>*) → None[163]

[138] https://docs.python.org/3/library/functions.html#float
[139] https://docs.python.org/3/library/functions.html#int
[140] https://docs.python.org/3/library/stdtypes.html#str
[141] https://docs.python.org/3/library/stdtypes.html#str
[142] https://docs.python.org/3/library/stdtypes.html#dict
[143] https://docs.python.org/3/library/stdtypes.html#str
[144] https://docs.python.org/3/library/stdtypes.html#dict
[145] https://docs.python.org/3/library/stdtypes.html#str
[146] https://docs.python.org/3/library/functions.html#float
[147] https://docs.python.org/3/library/functions.html#int
[148] https://docs.python.org/3/library/stdtypes.html#str
[149] https://docs.python.org/3/library/stdtypes.html#str
[150] https://docs.python.org/3/library/stdtypes.html#dict
[151] https://docs.python.org/3/library/stdtypes.html#str
[152] https://docs.python.org/3/library/stdtypes.html#dict
[153] https://docs.python.org/3/library/stdtypes.html#str
[154] https://docs.python.org/3/library/typing.html#typing.Any
[155] https://docs.python.org/3/library/stdtypes.html#str
[156] https://docs.python.org/3/library/stdtypes.html#str
[157] https://docs.python.org/3/library/stdtypes.html#str
[158] https://docs.python.org/3/library/functions.html#int
[159] https://docs.python.org/3/library/functions.html#int
[160] https://docs.python.org/3/library/functions.html#int
[161] https://docs.python.org/3/library/functions.html#float
[162] https://docs.python.org/3/library/fractions.html#fractions.Fraction
[163] https://docs.python.org/3/library/constants.html#None

`class constraints.OXGoalConstraint(`*id: ~uuid.UUID = <factory>, class_name: str = '', expression: ~constraints.OXpression.OXpression = <factory>, relational_operator: ~constraints.OXConstraint.RelationalOperators = RelationalOperators.EQUAL, rhs: float | int = 0, name: str = '', active_scenario: str = 'Default', scenarios: dict[str, dict[str, ~typing.Any]] = <factory>, positive_deviation_variable: ~variables.OXDeviationVar.OXDeviationVar = <factory>, negative_deviation_variable: ~variables.OXDeviationVar.OXDeviationVar = <factory>*`)`

Bases: `OXConstraint` (page 106)

A goal constraint for goal programming.

A goal constraint extends a regular constraint by adding deviation variables that measure how much the constraint is violated. In goal programming, the objective is typically to minimize undesired deviations.

`positive_deviation_variable`

The variable representing positive deviation from the goal.

Type

OXDeviationVar (page 139)

`negative_deviation_variable`

The variable representing negative deviation from the goal.

Type

OXDeviationVar (page 139)

Examples

```
>>> goal = constraint.to_goal()
>>> print(goal.positive_deviation_variable.desired)
False
>>> print(goal.negative_deviation_variable.desired)
True
```

See also

`OXConstraint` (page 106) `variables.OXDeviationVar.OXDeviationVar` (page 139)

`positive_deviation_variable: OXDeviationVar (page 139)`

`negative_deviation_variable: OXDeviationVar (page 139)`

`property desired_variables: list`[164]`[OXDeviationVar (page 139)]`

Get the list of desired deviation variables.

Returns

A list of deviation variables marked as desired.

Return type
list[165][*OXDeviationVar* (page 139)]

`property undesired_variables: list[166][OXDeviationVar (page 139)]`

Get the list of undesired deviation variables.

Returns
A list of deviation variables not marked as desired.

Return type
list[167][*OXDeviationVar* (page 139)]

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', expression: ~constraints.OXpression.OXpression = <factory>, relational_operator: ~constraints.OXConstraint.RelationalOperators = RelationalOperators.EQUAL, rhs: float | int = 0, name: str = '', active_scenario: str = 'Default', scenarios: dict[str, dict[str, ~typing.Any]] = <factory>, positive_deviation_variable: ~variables.OXDeviationVar.OXDeviationVar = <factory>, negative_deviation_variable: ~variables.OXDeviationVar.OXDeviationVar = <factory>*) → None[168]

Constraint Collections

`class constraints.OXConstraintSet`(*id: ~uuid.UUID = <factory>, class_name: str = '', objects: list[~base.OXObject.OXObject] = <factory>*)

Bases: `OXObjectPot`

A specialized container for managing OXConstraint objects.

OXConstraintSet extends OXObjectPot to provide a type-safe container specifically designed for managing collections of OXConstraint objects. It ensures that only OXConstraint instances can be added or removed from the set, and provides specialized query functionality for finding constraints based on their metadata.

This class is particularly useful for organizing constraints in optimization problems by category, type, or other metadata attributes stored in the constraint's related_data dictionary.

```
Inherits all attributes from OXObjectPot
```

`- objects`

List of constraint objects in the set

Type
list[169][OXObject]

`- id`

Unique identifier for the constraint set

Type
str[170]

[164] https://docs.python.org/3/library/stdtypes.html#list
[165] https://docs.python.org/3/library/stdtypes.html#list
[166] https://docs.python.org/3/library/stdtypes.html#list
[167] https://docs.python.org/3/library/stdtypes.html#list
[168] https://docs.python.org/3/library/constants.html#None

- name

  Human-readable name for the constraint set

  Type
  str[171]

`add_object(obj)`

Add an OXConstraint to the set

`remove_object(obj)`

Remove an OXConstraint from the set

`query(**kwargs)`

Find constraints by metadata attributes

Raises
`OXception` – When attempting to add/remove non-OXConstraint objects

Examples

```
>>> # Create a constraint set for capacity constraints
>>> capacity_set = OXConstraintSet(name="Capacity Constraints")
>>>
>>> # Add constraints with metadata
>>> for i, constraint in enumerate(capacity_constraints):
...     constraint.related_data["category"] = "capacity"
...     constraint.related_data["priority"] = "high"
...     capacity_set.add_object(constraint)
>>>
>>> # Query by metadata
>>> high_priority = capacity_set.query(priority="high")
>>> capacity_constraints = capacity_set.query(category="capacity")
>>>
>>> # Check set size
>>> print(f"Total constraints: {len(capacity_set)}")
>>>
>>> # Iterate through constraints
>>> for constraint in capacity_set:
...     print(f"Constraint: {constraint.name}")
```

`add_object(obj: OXObject)`

Add an OXConstraint object to the constraint set.

This method provides type-safe addition of constraint objects to the set. Only OXConstraint instances are allowed to be added to maintain the integrity of the constraint set.

Parameters
`obj` (`OXObject`) – The constraint object to add. Must be an instance of OXConstraint.

Raises
`OXception` – If the object is not an instance of OXConstraint.

Examples

```
>>> constraint_set = OXConstraintSet()
>>> constraint = OXConstraint(...)
>>> constraint_set.add_object(constraint)
>>> print(len(constraint_set))  # 1
```

`remove_object`(*obj: OXObject*)

Remove an OXConstraint object from the constraint set.

This method provides type-safe removal of constraint objects from the set. Only OXConstraint instances are allowed to be removed to maintain the integrity of the constraint set.

Parameters

`obj` (`OXObject`) – The constraint object to remove. Must be an instance of OXConstraint.

Raises

`OXception` – If the object is not an instance of OXConstraint.

Examples

```
>>> constraint_set = OXConstraintSet()
>>> constraint = OXConstraint(...)
>>> constraint_set.add_object(constraint)
>>> constraint_set.remove_object(constraint)
>>> print(len(constraint_set))  # 0
```

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', objects: list[~base.OXObject.OXObject] = <factory>*) → None[172]

Mathematical Expressions

class `constraints.OXpression`(*id: ~uuid.UUID = <factory>, class_name: str = '', variables: list[~uuid.UUID] = <factory>, weights: list[float | int | ~fractions.Fraction] = <factory>*)

Bases: `OXObject`

Mathematical expression representing linear combinations of optimization variables.

OXpression is a fundamental component of the OptiX optimization framework that represents linear mathematical expressions in the form: $c_1x_1 + c_2x_2 + \ldots + c_n x_n$, where c_i are coefficients (weights) and x_i are decision variables.

This class is designed to handle expressions used in both constraint definitions and objective functions within optimization problems. It provides precise arithmetic handling through fraction-based calculations to avoid floating-point precision errors that can occur in mathematical optimization.

[169] https://docs.python.org/3/library/stdtypes.html#list
[170] https://docs.python.org/3/library/stdtypes.html#str
[171] https://docs.python.org/3/library/stdtypes.html#str
[172] https://docs.python.org/3/library/constants.html#None

The class maintains variable references using UUIDs rather than direct object references, enabling serialization, persistence, and cross-system compatibility. This design pattern supports distributed optimization scenarios and model persistence.

Key Features:

- UUID-based variable referencing for serialization safety
- Automatic conversion between floating-point and integer coefficient representations
- Fraction-based arithmetic for mathematical precision
- Iterator support for easy traversal of variable-coefficient pairs
- Integration with OptiX constraint and objective function systems
- Support for multiple numeric types (int, float, Fraction, Decimal)

`variables`

Ordered list of variable UUIDs that participate in this expression. The order corresponds to the order of coefficients in the weights list.

Type

list[173][UUID]

`weights`

Ordered list of coefficients (weights) for each variable. Supports mixed numeric types with automatic conversion.

Type

list[174][float[175] | int[176] | Fraction]

Type Parameters:

The class inherits from OXObject, providing UUID-based identity and serialization capabilities.

Example

Basic usage of OXpression for creating mathematical expressions:

```
from uuid import UUID
from constraints import OXpression
from variables import OXVariable

# Create some optimization variables
x = OXVariable(name="production_x", lower_bound=0, upper_bound=100)
y = OXVariable(name="production_y", lower_bound=0, upper_bound=50)
z = OXVariable(name="production_z", lower_bound=0)

# Create expression: 2.5x + 1.75y + 3z (production cost function)
cost_expr = OXpression(
    variables=[x.id, y.id, z.id],
    weights=[2.5, 1.75, 3.0]
)
```

(continues on next page)

(continued from previous page)

```
# Access expression properties
print(f"Variables in expression: {cost_expr.number_of_variables}")  # 3
print(f"Integer weights: {cost_expr.integer_weights}")              # [10,
↪7, 12]
print(f"Common denominator: {cost_expr.integer_denominator}")       # 4

# Iterate through variable-coefficient pairs
for var_uuid, coefficient in cost_expr:
    print(f"Variable {var_uuid}: coefficient = {coefficient}")

# Example with mixed coefficient types
mixed_expr = OXpression(
    variables=[x.id, y.id],
    weights=[Fraction(1, 3), 0.75]  # Mixed Fraction and float
)
```

Note

- Variables are referenced by UUID to support serialization and persistence
- The weights list must have the same length as the variables list
- Automatic fraction conversion ensures mathematical precision for optimization solvers
- The class supports empty expressions (no variables/weights) for initialization
- All weight types are converted to fractions internally for consistent arithmetic

Warning

Ensure that the variables and weights lists maintain corresponding order and equal length. Mismatched lengths will result in undefined behavior during iteration and calculations.

See also

OXVariable: Decision variables used in expressions OXConstraint: Constraints that use OXpression for left-hand sides calculate_fraction: Internal function for precise fraction conversion get_integer_numerator_and_denominators: Utility for solver-compatible representations

`variables: list`[177]`[UUID`[178]`]`

`weights: list`[179]`[float`[180]` | int`[181]` | Fraction`[182]`]`

`property number_of_variables: int`[183]

Get the total count of variables participating in this mathematical expression.

This property provides a convenient way to determine the dimensionality of the linear expression, which is useful for validation, debugging, and solver setup. The count represents the number of decision variables that have non-zero coefficients in this expression.

Returns

The total number of variables in the expression. Returns 0 for empty
expressions (expressions with no variables or coefficients).

Return type
int[184]

Note

- The count is based on the length of the variables list
- Empty expressions return 0, which is valid for initialization scenarios
- The count should match the length of the weights list for consistency

Example

```
# Create expression with three variables
expr = OXpression(
    variables=[var1_id, var2_id, var3_id],
    weights=[1.0, 2.5, 0.75]
)
print(expr.number_of_variables)  # Output: 3

# Empty expression
empty_expr = OXpression()
print(empty_expr.number_of_variables)  # Output: 0
```

`property integer_weights: list[185][int[186]]`

Convert expression coefficients to integer representations with common denominator.

This property transforms all variable coefficients from their original numeric types (float, int, Fraction, Decimal) into integer values by finding a common denominator and scaling appropriately. This conversion is essential for optimization solvers that require integer coefficients while maintaining mathematical precision.

The conversion process: 1. Converts each weight to its exact fractional representation 2. Finds the least common multiple (LCM) of all denominators 3. Scales all numerators to use the common denominator 4. Returns the scaled integer numerators

Returns

Integer representations of all coefficients, scaled by the common
denominator. The order corresponds to the variables list order. Returns empty list if no weights are present.

Return type

list[187][int[188]]

Note

- Maintains exact mathematical precision through fraction arithmetic
- The integer values represent numerators when using the common denominator
- Use integer_denominator property to get the corresponding denominator
- Essential for solvers like CPLEX or Gurobi that prefer integer coefficients

Example

```
# Expression with decimal coefficients
expr = OXpression(
    variables=[x_id, y_id, z_id],
    weights=[0.5, 1.25, 2.0]
)

print(expr.integer_weights)      # [2, 5, 8]
print(expr.integer_denominator)  # 4

# Verification: 2/4 = 0.5, 5/4 = 1.25, 8/4 = 2.0
```

See also

integer_denominator: Get the common denominator for these integer weights
get_integer_numerator_and_denominators: The underlying conversion function

`property integer_denominator: int`[189]

Get the common denominator used for integer weight representation.

This property returns the least common multiple (LCM) of all denominators in the fractional representations of the expression coefficients. When combined with the integer_weights property, it allows for exact reconstruction of the original coefficient values while providing integer representations suitable for optimization solvers.

The denominator represents the scaling factor applied to convert floating-point or fractional coefficients into integers. This approach maintains mathematical precision and avoids floating-point arithmetic errors in optimization calculations.

Returns

The common denominator for all coefficients in the expression.
Returns 1 if all weights are integers, or the LCM of all fractional denominators if floating-point weights are present. Returns 1 for empty expressions.

Return type

int[190]

Note

- Always returns a positive integer value
- The LCM approach ensures the smallest possible common denominator
- Combined with integer_weights, provides exact coefficient representation
- Essential for maintaining precision in constraint and objective definitions

Example

```
# Expression with fractional coefficients
expr = OXpression(
    variables=[x_id, y_id, z_id],
    weights=[0.5, 0.25, 1.75]  # 1/2, 1/4, 7/4
)

print(expr.integer_denominator)  # 4 (LCM of 2, 4, 4)
print(expr.integer_weights)      # [2, 1, 7]

# Verification: 2/4 = 0.5, 1/4 = 0.25, 7/4 = 1.75

# Expression with integer coefficients
int_expr = OXpression(
    variables=[x_id, y_id],
    weights=[2, 3]
)
print(int_expr.integer_denominator)  # 1
```

See also

integer_weights: Get the integer numerators for these coefficients get_integer_numerator_and_denominators: The underlying conversion function

`__iter__()`

Enable iteration over variable-coefficient pairs in the mathematical expression.

This method implements the iterator protocol, allowing the OXpression object to be used in for-loops and other iteration contexts. It yields tuples of (variable_uuid, coefficient) pairs, maintaining the order defined in the variables and weights lists.

The iterator is particularly useful for: - Traversing expression terms for solver setup - Debugging and validation of expression contents - Serialization and persistence operations - Constructing string representations of expressions

Yields

tuple[UUID, float | int | Fraction] –

Each iteration yields a tuple containing:

- UUID: The unique identifier of the variable

- float | int | Fraction: The coefficient (weight) for that variable

Note

- Iteration order matches the order of variables and weights lists
- Empty expressions will not yield any items
- The yielded coefficients maintain their original numeric types
- Supports standard Python iteration protocols (for loops, list comprehension, etc.)

Example

```
from uuid import uuid4
from fractions import Fraction

# Create expression with mixed coefficient types
expr = OXpression(
    variables=[uuid4(), uuid4(), uuid4()],
    weights=[2.5, 3, Fraction(1, 2)]
)

# Iterate over variable-coefficient pairs
for var_uuid, coefficient in expr:
    print(f"Variable {var_uuid}: coefficient = {coefficient}")

# Use in list comprehension
terms = [(str(var_id)[:8], coef) for var_id, coef in expr]
print(f"Expression terms: {terms}")

# Convert to dictionary
expr_dict = dict(expr)

# Count terms
term_count = len(list(expr))
```

Raises

`ValueError`[191] – If variables and weights lists have different lengths (this would indicate a malformed expression)

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', variables: list[~uuid.UUID] = <factory>, weights: list[float | int | ~fractions.Fraction] = <factory>*) → None[192] `constraints.get_integer_numerator_and_denominators(`*numbers: list[193][float[194] | int[195]]*`)` → tuple[196][int[197], list[198][int[199]]]

Convert a list of floating-point or integer weights to integer representations.

This function takes a collection of numeric values (which may include floating-point numbers and integers) and converts them to exact integer representations by finding a common denominator. This is essential for optimization solvers that require integer coefficients while maintaining mathematical precision.

The function works by: 1. Converting each number to its fractional representation using calculate_fraction() 2. Finding the least common multiple (LCM) of all denominators 3. Scaling all numerators by appropriate factors to use the common denominator 4. Returning both the common denominator and the scaled integer numerators

Parameters

`numbers` (`list`[200][`float`[201] | `int`[202]]) – A list of numeric values to convert to integer representations. Can contain floating-point numbers, integers, or a mix of both types.

Returns

A tuple containing:

- int: The common denominator for all converted values
- **list[int]: List of integer numerators corresponding to each input number**
  when expressed with the common denominator

Return type

tuple[203][int[204], list[205][int[206]]]

Raises

- `ValueError`[207] – If the input list is empty or contains non-numeric values.
- `ZeroDivisionError`[208] – If any input number results in a zero denominator.

[173] https://docs.python.org/3/library/stdtypes.html#list
[174] https://docs.python.org/3/library/stdtypes.html#list
[175] https://docs.python.org/3/library/functions.html#float
[176] https://docs.python.org/3/library/functions.html#int
[177] https://docs.python.org/3/library/stdtypes.html#list
[178] https://docs.python.org/3/library/uuid.html#uuid.UUID
[179] https://docs.python.org/3/library/stdtypes.html#list
[180] https://docs.python.org/3/library/functions.html#float
[181] https://docs.python.org/3/library/functions.html#int
[182] https://docs.python.org/3/library/fractions.html#fractions.Fraction
[183] https://docs.python.org/3/library/functions.html#int
[184] https://docs.python.org/3/library/functions.html#int
[185] https://docs.python.org/3/library/stdtypes.html#list
[186] https://docs.python.org/3/library/functions.html#int
[187] https://docs.python.org/3/library/stdtypes.html#list
[188] https://docs.python.org/3/library/functions.html#int
[189] https://docs.python.org/3/library/functions.html#int
[190] https://docs.python.org/3/library/functions.html#int
[191] https://docs.python.org/3/library/exceptions.html#ValueError
[192] https://docs.python.org/3/library/constants.html#None

Note

- All calculations maintain exact precision through Fraction arithmetic
- The LCM approach ensures the smallest possible common denominator
- Integer inputs are handled efficiently as Fraction(value, 1)
- Useful for preparing coefficients for linear programming solvers

Example

```python
# Convert mixed numeric types
numbers = [0.5, 1.5, 2, 0.25]
denominator, numerators = get_integer_numerator_and_denominators(numbers)
print(f"Common denominator: {denominator}")  # 4
print(f"Integer numerators: {numerators}")   # [2, 6, 8, 1]

# Verify the conversion
for i, num in enumerate(numbers):
    converted = numerators[i] / denominator
    print(f"{num} = {numerators[i]}/{denominator} = {converted}")

# Example with simple fractions
simple_fractions = [0.5, 1.5, 2.0]
denom, nums = get_integer_numerator_and_denominators(simple_fractions)
# Returns: (2, [1, 3, 4]) representing [1/2, 3/2, 4/2]
```

See also

calculate_fraction: Used internally to convert individual numbers to fractions math.lcm: Used to find the least common multiple of denominators

193 https://docs.python.org/3/library/stdtypes.html#list
194 https://docs.python.org/3/library/functions.html#float
195 https://docs.python.org/3/library/functions.html#int
196 https://docs.python.org/3/library/stdtypes.html#tuple
197 https://docs.python.org/3/library/functions.html#int
198 https://docs.python.org/3/library/stdtypes.html#list
199 https://docs.python.org/3/library/functions.html#int
200 https://docs.python.org/3/library/stdtypes.html#list
201 https://docs.python.org/3/library/functions.html#float
202 https://docs.python.org/3/library/functions.html#int
203 https://docs.python.org/3/library/stdtypes.html#tuple
204 https://docs.python.org/3/library/functions.html#int
205 https://docs.python.org/3/library/stdtypes.html#list
206 https://docs.python.org/3/library/functions.html#int
207 https://docs.python.org/3/library/exceptions.html#ValueError
208 https://docs.python.org/3/library/exceptions.html#ZeroDivisionError

Special Constraints

`class constraints.OXSpecialConstraint`(*id: ~uuid.UUID = <factory>*, *class_name: str = ”*)

Bases: `OXObject`

Base class for special constraints in optimization problems.

Special constraints are non-linear constraints that cannot be expressed as simple linear relationships. They require special handling by solvers and often involve complex relationships between variables.

This class serves as a base for all special constraint types including multiplicative, division, modulo, summation, and conditional constraints.

See also

`OXNonLinearEqualityConstraint` (page 123) `OXMultiplicativeEqualityConstraint` (page 124) `OXDivisionEqualityConstraint` (page 124) `OXModuloEqualityConstraint` (page 126) `OXSummationEqualityConstraint` (page 127) `OXConditionalConstraint` (page 127)

`__init__`(*id: ~uuid.UUID = <factory>*, *class_name: str = ”*) → None[209]

`class constraints.OXNonLinearEqualityConstraint`(*id: ~uuid.UUID = <factory>*, *class_name: str = ”*, *output_variable: ~uuid.UUID = <factory>*)

Bases: `OXSpecialConstraint` (page 123)

Base class for non-linear equality constraints.

Non-linear equality constraints represent relationships that cannot be expressed as linear combinations of variables. They typically have the form f(x1, x2, …, xn) = output_variable, where f is a non-linear function.

`output_variable`

The UUID of the variable that stores the result of the non-linear operation.

Type

UUID

Examples

This is a base class and should not be instantiated directly. Use specific subclasses like OXMultiplicativeEqualityConstraint.

See also

`OXMultiplicativeEqualityConstraint` (page 124) `OXDivisionEqualityConstraint` (page 124) `OXModuloEqualityConstraint` (page 126)

`output_variable: UUID`[210]

[209] https://docs.python.org/3/library/constants.html#None

`__init__`(*id: ~uuid.UUID = <factory>*, *class_name: str = ”*, *output_variable: ~uuid.UUID = <factory>*) → None[211]

`class constraints.OXMultiplicativeEqualityConstraint`(*id: ~uuid.UUID = <factory>*, *class_name: str = ”*, *output_variable: ~uuid.UUID = <factory>*, *input_variables: list[~uuid.UUID] = <factory>*)

Bases: `OXNonLinearEqualityConstraint` (page 123)

A constraint representing multiplication of variables.

This constraint enforces that the output variable equals the product of all input variables: output_variable = input_variable_1 * input_variable_2 * … * input_variable_n

`input_variables`

The list of variable UUIDs to multiply.

Type

list[212][UUID]

`output_variable`

The UUID of the variable that stores the product. Inherited from OXNonLinearEqualityConstraint.

Type

UUID

Examples

```
>>> # Create a constraint: z = x * y
>>> constraint = OXMultiplicativeEqualityConstraint(
...     input_variables=[x.id, y.id],
...     output_variable=z.id
... )
```

Note

This constraint is typically handled by constraint programming solvers that support non-linear operations.

`input_variables: list[213][UUID[214]]`

`__init__`(*id: ~uuid.UUID = <factory>*, *class_name: str = ”*, *output_variable: ~uuid.UUID = <factory>*, *input_variables: list[~uuid.UUID] = <factory>*) → None[215]

[210] https://docs.python.org/3/library/uuid.html#uuid.UUID
[211] https://docs.python.org/3/library/constants.html#None
[212] https://docs.python.org/3/library/stdtypes.html#list
[213] https://docs.python.org/3/library/stdtypes.html#list
[214] https://docs.python.org/3/library/uuid.html#uuid.UUID
[215] https://docs.python.org/3/library/constants.html#None

class constraints.OXDivisionEqualityConstraint(*id: ~uuid.UUID = <factory>, class_name: str = ”, output_variable: ~uuid.UUID = <factory>, input_variable: ~uuid.UUID = <factory>, denominator: int = 1*)

Bases: OXNonLinearEqualityConstraint (page 123)

A constraint representing integer division of a variable.

This constraint enforces that the output variable equals the integer division of the input variable by the denominator: output_variable = input_variable // denominator

input_variable

The UUID of the variable to divide.

Type

UUID

denominator

The divisor for the division operation.

Type

int[216]

output_variable

The UUID of the variable that stores the quotient. Inherited from OXNonLinearEqualityConstraint.

Type

UUID

Examples

```
>>> # Create a constraint: z = x // 3
>>> constraint = OXDivisionEqualityConstraint(
...     input_variable=x.id,
...     denominator=3,
...     output_variable=z.id
... )
```

Note

This constraint performs integer division (floor division), not floating-point division.

input_variable: UUID[217]

denominator: int[218] = 1

__init__(*id: ~uuid.UUID = <factory>, class_name: str = ”, output_variable: ~uuid.UUID = <factory>, input_variable: ~uuid.UUID = <factory>, denominator: int = 1*) → None[219] class constraints.OXModuloEqualityConstraint(*id: ~uuid.UUID = <factory>, class_name: str = ", output_variable: ~uuid.UUID = <factory>, input_variable: ~uuid.UUID = <factory>, denominator: int = 1*)

Bases: OXNonLinearEqualityConstraint (page 123)

A constraint representing modulo operation on a variable.

This constraint enforces that the output variable equals the remainder of the input variable divided by the denominator: output_variable = input_variable % denominator

input_variable

The UUID of the variable to apply modulo to.

Type

UUID

denominator

The divisor for the modulo operation.

Type

int[220]

output_variable

The UUID of the variable that stores the remainder. Inherited from OXNonLinearEqualityConstraint.

Type

UUID

Examples

```
>>> # Create a constraint: z = x % 5
>>> constraint = OXModuloEqualityConstraint(
...     input_variable=x.id,
...     denominator=5,
...     output_variable=z.id
... )
```

Note

The result is always non-negative and less than the denominator.

input_variable: UUID[221]

denominator: int[222] = 1

__init__(*id: ~uuid.UUID = <factory>, class_name: str = ", output_variable: ~uuid.UUID = <factory>, input_variable: ~uuid.UUID = <factory>, denominator: int = 1*) → None[223]

[216] https://docs.python.org/3/library/functions.html#int
[217] https://docs.python.org/3/library/uuid.html#uuid.UUID
[218] https://docs.python.org/3/library/functions.html#int
[219] https://docs.python.org/3/library/constants.html#None

class constraints.OXSummationEqualityConstraint(*id: ~uuid.UUID = <factory>, class_name: str = ”, input_variables: list[~uuid.UUID] = <factory>, output_variable: ~uuid.UUID = <factory>*)

Bases: OXSpecialConstraint (page 123)

A constraint representing summation of variables.

This constraint enforces that the output variable equals the sum of all input variables: output_variable = input_variable_1 + input_variable_2 + … + input_variable_n

input_variables

The list of variable UUIDs to sum.

Type

list[224][UUID]

output_variable

The UUID of the variable that stores the sum.

Type

UUID

Examples

```
>>> # Create a constraint: z = x + y + w
>>> constraint = OXSummationEqualityConstraint(
...     input_variables=[x.id, y.id, w.id],
...     output_variable=z.id
... )
```

Note

While this could be expressed as a linear constraint, it’s included as a special constraint for consistency and solver optimization.

input_variables: list[225][UUID[226]]

output_variable: UUID[227]

__init__(*id: ~uuid.UUID = <factory>, class_name: str = ”, input_variables: list[~uuid.UUID] = <factory>, output_variable: ~uuid.UUID = <factory>*) → None[228]

[220] https://docs.python.org/3/library/functions.html#int
[221] https://docs.python.org/3/library/uuid.html#uuid.UUID
[222] https://docs.python.org/3/library/functions.html#int
[223] https://docs.python.org/3/library/constants.html#None
[224] https://docs.python.org/3/library/stdtypes.html#list
[225] https://docs.python.org/3/library/stdtypes.html#list
[226] https://docs.python.org/3/library/uuid.html#uuid.UUID
[227] https://docs.python.org/3/library/uuid.html#uuid.UUID
[228] https://docs.python.org/3/library/constants.html#None

`class constraints.OXConditionalConstraint`(*id: ~uuid.UUID = <factory>, class_name: str = '', indicator_variable: ~uuid.UUID = <factory>, input_constraint: ~uuid.UUID = <factory>, constraint_if_true: ~uuid.UUID = <factory>, constraint_if_false: ~uuid.UUID = <factory>*)

Bases: `OXSpecialConstraint` (page 123)

A constraint representing conditional logic.

This constraint enforces different constraints based on the value of an indicator variable. If the indicator variable is true, constraint_if_true is enforced; otherwise, constraint_if_false is enforced.

`indicator_variable`

The UUID of the boolean variable that determines which constraint to enforce.

Type
UUID

`input_constraint`

The UUID of the base constraint to evaluate.

Type
UUID

`constraint_if_true`

The UUID of the constraint to enforce if the indicator variable is true.

Type
UUID

`constraint_if_false`

The UUID of the constraint to enforce if the indicator variable is false.

Type
UUID

Examples

```
>>> # Create a conditional constraint: if flag then x >= 5 else x <= 3
>>> constraint = OXConditionalConstraint(
...     indicator_variable=flag.id,
...     input_constraint=base_constraint.id,
...     constraint_if_true=upper_bound_constraint.id,
...     constraint_if_false=lower_bound_constraint.id
... )
```

Note

This constraint is used for modeling logical implications and conditional relationships in optimization problems.

`indicator_variable`: UUID[229]

`input_constraint`: UUID[230]

`constraint_if_true`: UUID[231]

`constraint_if_false`: UUID[232]

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = ", indicator_variable: ~uuid.UUID = <factory>, input_constraint: ~uuid.UUID = <factory>, constraint_if_true: ~uuid.UUID = <factory>, constraint_if_false: ~uuid.UUID = <factory>*) → None[233]

9.7.2 Enumerations

class `constraints.RelationalOperators`(**values*)

Enumeration of relational operators for constraints.

These operators define the relationship between the left-hand side (expression) and right-hand side (rhs) of a constraint.

`GREATER_THAN`

The ">" operator.

Type
str[234]

`GREATER_THAN_EQUAL`

The ">=" operator.

Type
str[235]

`EQUAL`

The "=" operator.

Type
str[236]

`LESS_THAN`

The "<" operator.

Type
str[237]

`LESS_THAN_EQUAL`

The "<=" operator.

Type
str[238]

Available operators:

[229] https://docs.python.org/3/library/uuid.html#uuid.UUID
[230] https://docs.python.org/3/library/uuid.html#uuid.UUID
[231] https://docs.python.org/3/library/uuid.html#uuid.UUID
[232] https://docs.python.org/3/library/uuid.html#uuid.UUID
[233] https://docs.python.org/3/library/constants.html#None

- GREATER_THAN - Greater than (>)
- GREATER_THAN_EQUAL - Greater than or equal (>=)
- EQUAL - Equal (=)
- LESS_THAN - Less than (<)
- LESS_THAN_EQUAL - Less than or equal (<=)

GREATER_THAN = '>'

GREATER_THAN_EQUAL = '>='

EQUAL = '='

LESS_THAN = '<'

LESS_THAN_EQUAL = '<='

9.7.3 Examples

Basic Linear Constraints

```
from constraints import OXConstraint, OXpression, RelationalOperators
from variables import OXVariable

# Create variables
x = OXVariable(name="x", lower_bound=0, upper_bound=100)
y = OXVariable(name="y", lower_bound=0, upper_bound=100)

# Create expression: 2x + 3y
expr = OXpression(variables=[x.id, y.id], weights=[2, 3])

# Create constraint: 2x + 3y <= 500
constraint = OXConstraint(
    expression=expr,
    relational_operator=RelationalOperators.LESS_THAN_EQUAL,
    rhs=500,
    name="Resource capacity constraint"
)

print(f"Constraint: {constraint}")
```

Goal Programming Constraints

```
from constraints import OXConstraint, RelationalOperators

# Create a regular constraint
```

(continues on next page)

[234] https://docs.python.org/3/library/stdtypes.html#str
[235] https://docs.python.org/3/library/stdtypes.html#str
[236] https://docs.python.org/3/library/stdtypes.html#str
[237] https://docs.python.org/3/library/stdtypes.html#str
[238] https://docs.python.org/3/library/stdtypes.html#str

(continued from previous page)

```
constraint = OXConstraint(
    expression=expr,
    relational_operator=RelationalOperators.LESS_THAN_EQUAL,
    rhs=100,
    name="Production target"
)

# Convert to goal constraint
goal_constraint = constraint.to_goal()

# Check deviation variables
print(f"Positive deviation desired: {goal_constraint.positive_deviation_variable.
↪desired}")
print(f"Negative deviation desired: {goal_constraint.negative_deviation_variable.
↪desired}")
```

Constraint Sets

```
from constraints import OXConstraintSet, OXConstraint

# Create constraint set
constraint_set = OXConstraintSet(name="Production Constraints")

# Add constraints to the set
for i, constraint in enumerate(production_constraints):
    # Add metadata for querying
    constraint.related_data["category"] = "production"
    constraint.related_data["priority"] = "high" if i < 3 else "medium"
    constraint_set.add_object(constraint)

print(f"Total constraints: {len(constraint_set)}")

# Query constraints by metadata
high_priority = constraint_set.query(priority="high")
production_constraints = constraint_set.query(category="production")
```

Mathematical Expressions

```
from constraints import OXpression
from variables import OXVariable

# Create variables
x = OXVariable(name="x", lower_bound=0)
y = OXVariable(name="y", lower_bound=0)
z = OXVariable(name="z", lower_bound=0)

# Create expression: 2.5x + 1.5y + 3z
expr = OXpression(
```

(continues on next page)

(continued from previous page)

```
    variables=[x.id, y.id, z.id],
    weights=[2.5, 1.5, 3]
)

# Access variable coefficients
for var_id, weight in zip(expr.variables, expr.weights):
    print(f"Variable {var_id}: coefficient {weight}")

# Convert to integer coefficients
int_weights, denominator = expr.get_integer_weights()
print(f"Integer weights: {int_weights}, common denominator: {denominator}")
```

Special Constraints

```
from constraints import OXMultiplicativeEqualityConstraint,
↪OXDivisionEqualityConstraint
from variables import OXVariable

# Create variables for multiplication
x = OXVariable(name="x", lower_bound=0, upper_bound=50)
y = OXVariable(name="y", lower_bound=0, upper_bound=50)
product = OXVariable(name="product", lower_bound=0, upper_bound=2500)

# Multiplication constraint: x * y = product
mult_constraint = OXMultiplicativeEqualityConstraint(
    left_variable_id=x.id,
    right_variable_id=y.id,
    result_variable_id=product.id,
    name="Product calculation"
)

# Division constraint: x / y = quotient (integer division)
quotient = OXVariable(name="quotient", lower_bound=0, upper_bound=100)
div_constraint = OXDivisionEqualityConstraint(
    left_variable_id=x.id,
    right_variable_id=y.id,
    result_variable_id=quotient.id,
    name="Division calculation"
)
```

Conditional Constraints

```
from constraints import OXConditionalConstraint
from variables import OXVariable

# Create binary variables
condition = OXVariable(name="use_machine", lower_bound=0, upper_bound=1,
↪variable_type="binary")
```

(continues on next page)

(continued from previous page)

```
production = OXVariable(name="production", lower_bound=0, upper_bound=100)
cost = OXVariable(name="cost", lower_bound=0, upper_bound=1000)

# Conditional constraint: if use_machine then production >= 10
conditional = OXConditionalConstraint(
    condition_variable_id=condition.id,
    implication_variable_id=production.id,
    threshold_value=10,
    name="Minimum production when machine is used"
)
```

Constraint Validation

```
def validate_constraint_satisfaction(constraint, variable_values):
    """Check if a constraint is satisfied by given variable values."""

    # Calculate left-hand side value
    lhs_value = 0
    for var_id, weight in zip(constraint.expression.variables, constraint.
↪expression.weights):
        lhs_value += weight * variable_values.get(var_id, 0)

    # Check constraint satisfaction
    tolerance = 1e-6

    if constraint.relational_operator == RelationalOperators.LESS_THAN_EQUAL:
        return lhs_value <= constraint.rhs + tolerance
    elif constraint.relational_operator == RelationalOperators.GREATER_THAN_
↪EQUAL:
        return lhs_value >= constraint.rhs - tolerance
    elif constraint.relational_operator == RelationalOperators.EQUAL:
        return abs(lhs_value - constraint.rhs) <= tolerance
    elif constraint.relational_operator == RelationalOperators.LESS_THAN:
        return lhs_value < constraint.rhs + tolerance
    elif constraint.relational_operator == RelationalOperators.GREATER_THAN:
        return lhs_value > constraint.rhs - tolerance

    return False

# Usage
is_satisfied = validate_constraint_satisfaction(constraint, solution_values)
print(f"Constraint satisfied: {is_satisfied}")
```

Precise Arithmetic with Fractions

```
from constraints import OXpression, get_integer_numerator_and_denominators

# Create expression with decimal coefficients
```

(continues on next page)

(continued from previous page)

```
expr = OXpression(
    variables=[x.id, y.id, z.id],
    weights=[0.333, 0.667, 1.5]  # These will be converted to fractions
)

# Convert to integer representation for solver compatibility
int_weights, common_denom = get_integer_numerator_and_denominators(expr.weights)
print(f"Integer weights: {int_weights}")
print(f"Common denominator: {common_denom}")

# Access fraction properties
for i, weight in enumerate(expr.weights):
    frac_weight = expr.get_fraction_weight(i)
    print(f"Weight {i}: {frac_weight.numerator}/{frac_weight.denominator}")
```

9.7.4 See Also

- *Problem Module* (page 81) - Problem classes that use constraints
- *Variables Module* (page 134) - Variable definitions and management
- *Solvers Module* (page 163) - Solver implementations that handle constraints
- ../user_guide/constraints - Advanced constraint modeling guide

9.8 Variables Module

The variables module provides comprehensive decision variable management for the OptiX optimization framework. It implements a complete variable system supporting linear programming (LP), goal programming (GP), and constraint satisfaction problems (CSP) with advanced features for bounds management, relationship tracking, and specialized variable types.

9.8.1 Core Variable Classes

Base Decision Variable

class variables.OXVariable(*id: ~uuid.UUID = <factory>, class_name: str = ", name: str = ", description: str = ", value: float | int | bool = None, upper_bound: float | int = inf, lower_bound: float | int = 0, related_data: dict[str, ~uuid.UUID] = <factory>*)

Bases: `OXObject`

Fundamental decision variable class for mathematical optimization problems.

This class provides a comprehensive representation of decision variables used in optimization modeling within the OptiX framework. It extends the base OXObject class to include domain-specific features such as bounds management, value tracking, and relationship linking essential for complex optimization scenarios.

The class implements automatic validation, intelligent naming, and flexible data relationships to support various optimization paradigms including linear programming, integer programming, and goal programming applications.

Key Capabilities:

- Automatic bounds validation with infinity support for unbounded variables
- UUID-based relationship tracking for linking variables to data entities
- Intelligent automatic naming using UUID when names are not provided
- Type-safe value assignment with comprehensive validation
- Integration with solver interfaces through standardized attributes

`name`

The human-readable identifier for the variable. If empty or whitespace, automatically generated as "var_<uuid>" to ensure uniqueness and traceability throughout the optimization process.

Type

str[239]

`description`

Detailed description of the variable's purpose and meaning within the optimization context. Used for documentation and model interpretation purposes.

Type

str[240]

`value`

The current assigned value of the variable. Can be None for unassigned variables. Should respect the defined bounds when set by optimization solvers.

Type

float[241] | int[242] | bool[243]

`upper_bound`

The maximum allowable value for the variable. Defaults to positive infinity for unbounded variables. Must be greater than or equal to lower_bound.

Type

float[244] | int[245]

`lower_bound`

The minimum allowable value for the variable. Defaults to 0 for non-negative variables. Must be less than or equal to upper_bound.

Type

float[246] | int[247]

`related_data`

Dictionary mapping relationship type names to UUID identifiers of related objects. Enables complex data modeling and constraint relationships.

Type

dict[248][str[249], UUID]

Raises

OXception – If lower_bound is greater than upper_bound during initialization. This validation ensures mathematical consistency of the variable domain.

Performance:

- Variable creation is optimized for large-scale problems with minimal overhead
- Bounds checking is performed only during initialization and explicit validation
- String representation is cached for efficient display in large variable sets

Thread Safety:

- Individual variable instances are thread-safe for read operations
- Modification operations should be synchronized in multi-threaded environments
- The related_data dictionary requires external synchronization for concurrent access

Examples

Create variables for different optimization scenarios:

```
# Production planning variable with finite bounds
production = OXVariable(
    name="daily_production",
    description="Daily production quantity in units",
    lower_bound=0,
    upper_bound=1000,
    value=500
)

# Binary decision variable for facility location
facility_open = OXVariable(
    name="facility_open",
    description="Whether to open the facility (0=closed, 1=open)",
    lower_bound=0,
    upper_bound=1,
    value=0
)

# Unbounded variable for inventory surplus/deficit
inventory_delta = OXVariable(
    name="inventory_change",
    description="Change in inventory level (positive=surplus,
↪negative=deficit)",
    lower_bound=float('-inf'),
    upper_bound=float('inf')
)
```

(continues on next page)

(continued from previous page)

```
# Link variable to related business entities
from uuid import uuid4
customer_id = uuid4()
production.related_data["customer"] = customer_id
production.related_data["facility"] = facility_open.id
```

Note

- Variables are immutable after solver assignment to maintain solution integrity
- The bounds validation is strict and prevents invalid domain specifications
- Automatic naming ensures no variable is left without a unique identifier
- Related data relationships support complex constraint modeling patterns

See also

`variables.OXDeviationVar.OXDeviationVar` (page 139): Specialized deviation variables for goal programming. `variables.OXVariableSet.OXVariableSet` (page 143): Container for managing variable collections. `base.OXObject`: Base class providing UUID and serialization capabilities.

`name: str`[250] `= ''`

`description: str`[251] `= ''`

`value: float`[252] `| int`[253] `| bool`[254] `= None`

`upper_bound: float`[255] `| int`[256] `= inf`

`lower_bound: float`[257] `| int`[258] `= 0`

`related_data: dict`[259]`[str`[260]`, UUID`[261]`]`

`__post_init__()`

Initialize and validate the variable after dataclass construction.

This method is automatically invoked by the dataclass mechanism after all field assignments are complete. It performs critical validation and setup operations to ensure the variable is in a consistent and valid state.

The initialization process includes: 1. Calling the parent OXObject initialization for UUID and class name setup 2. Validating that bounds are mathematically consistent (lower $\leq$ upper) 3. Generating an automatic name if none was provided or if empty/whitespace 4. Ensuring all internal state is properly configured for optimization use

Validation Rules:

- Lower bound must be less than or equal to upper bound
- Infinite bounds are permitted and properly handled

- Empty or whitespace-only names trigger automatic UUID-based naming

Raises

`OXception` – If lower_bound is greater than upper_bound. This ensures mathematical validity of the variable's domain and prevents optimization solver errors that would occur with invalid bounds.

Note

- This method is called automatically and should not be invoked manually
- The automatic naming scheme uses format "var_<uuid>" for traceability
- All validation occurs during object creation, not during value assignment
- Parent initialization must complete successfully for proper inheritance

Examples

The validation prevents invalid variable creation:

```
# This will raise OXception due to invalid bounds
try:
    invalid_var = OXVariable(lower_bound=10, upper_bound=5)
except OXception as e:
    print("Invalid bounds detected:", e)

# This creates a variable with automatic naming
auto_named = OXVariable(name="   ")  # Whitespace triggers auto-naming
print(auto_named.name)  # Output: "var_<some-uuid>"
```

`__str__()`

Return a string representation of the variable.

Provides a concise, human-readable identifier for the variable that is suitable for display in optimization model summaries, debug output, and solver interfaces.

Returns

The variable's name, which serves as its primary identifier in the optimization context. This will be either the user-provided name or the automatically generated "var_<uuid>" format.

Return type

str[262]

Note

- The string representation is optimized for readability and brevity
- Used extensively by solvers and constraint display mechanisms
- Guaranteed to be unique due to UUID-based automatic naming fallback

Examples

```
named_var = OXVariable(name="production_rate")
print(str(named_var))  # Output: "production_rate"

auto_var = OXVariable()  # No name provided
print(str(auto_var))    # Output: "var_<uuid>"
```

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', name: str = '', description: str = '', value: float | int | bool = None, upper_bound: float | int = inf, lower_bound: float | int = 0, related_data: dict[str, ~uuid.UUID] = <factory>*) → None[263]

Goal Programming Variables

class `variables.OXDeviationVar`(*id: ~uuid.UUID = <factory>, class_name: str = '', name: str = '', description: str = '', value: float | int | bool = None, upper_bound: float | int = inf, lower_bound: float | int = 0, related_data: dict[str, ~uuid.UUID] = <factory>, desired: bool = False*)

Bases: `OXVariable` (page 134)

Specialized decision variable for goal programming deviation measurement.

This class extends the base OXVariable to provide goal programming-specific functionality for measuring and tracking deviations from target goals. Deviation variables are essential components of goal programming models where multiple objectives are balanced through the minimization of unwanted deviations.

In goal programming, deviation variables come in pairs (positive and negative) to measure over-achievement and under-achievement relative to goal targets. The desirability

239 https://docs.python.org/3/library/stdtypes.html#str
240 https://docs.python.org/3/library/stdtypes.html#str
241 https://docs.python.org/3/library/functions.html#float
242 https://docs.python.org/3/library/functions.html#int
243 https://docs.python.org/3/library/functions.html#bool
244 https://docs.python.org/3/library/functions.html#float
245 https://docs.python.org/3/library/functions.html#int
246 https://docs.python.org/3/library/functions.html#float
247 https://docs.python.org/3/library/functions.html#int
248 https://docs.python.org/3/library/stdtypes.html#dict
249 https://docs.python.org/3/library/stdtypes.html#str
250 https://docs.python.org/3/library/stdtypes.html#str
251 https://docs.python.org/3/library/stdtypes.html#str
252 https://docs.python.org/3/library/functions.html#float
253 https://docs.python.org/3/library/functions.html#int
254 https://docs.python.org/3/library/functions.html#bool
255 https://docs.python.org/3/library/functions.html#float
256 https://docs.python.org/3/library/functions.html#int
257 https://docs.python.org/3/library/functions.html#float
258 https://docs.python.org/3/library/functions.html#int
259 https://docs.python.org/3/library/stdtypes.html#dict
260 https://docs.python.org/3/library/stdtypes.html#str
261 https://docs.python.org/3/library/uuid.html#uuid.UUID
262 https://docs.python.org/3/library/stdtypes.html#str
263 https://docs.python.org/3/library/constants.html#None

flag helps optimization algorithms prioritize which deviations to minimize, supporting complex multi-objective decision making scenarios.

Key Capabilities:

- Goal programming deviation measurement with directional semantics
- Desirability tracking for optimization objective formulation
- Full integration with standard optimization variable operations
- Enhanced string representation showing goal programming characteristics
- Seamless compatibility with OXVariable-based constraint systems

Mathematical Context:

In goal programming, a typical goal constraint has the form: achievement_level + negative_deviation - positive_deviation = target_value

Where: - achievement_level: actual performance or resource usage - negative_deviation: shortfall below target (under-achievement) - positive_deviation: excess above target (over-achievement) - target_value: desired goal level

`desired`

Flag indicating whether this deviation is desirable in the optimization context. False (default) means the deviation should be minimized, while True means it may be acceptable or even beneficial. This flag influences objective function formulation in goal programming models.

Type

bool[264]

Performance:

- Inherits all performance characteristics from OXVariable
- Minimal overhead for the additional boolean flag
- String representation includes desirability information with minimal cost

Thread Safety:

- Same thread safety characteristics as OXVariable
- The desired flag is immutable after initialization for consistency

Examples

Create deviation variables for different goal programming scenarios:

```
from variables.OXDeviationVar import OXDeviationVar

# Budget constraint - over-spending is undesirable
budget_overrun = OXDeviationVar(
    name="budget_positive_deviation",
    description="Amount by which spending exceeds budget",
    lower_bound=0,
    upper_bound=float('inf'),
    desired=False  # Minimize over-spending
```

(continues on next page)

(continued from previous page)

```
)

# Production target - under-production is undesirable
production_shortfall = OXDeviationVar(
    name="production_negative_deviation",
    description="Amount by which production falls short of target",
    lower_bound=0,
    upper_bound=float('inf'),
    desired=False  # Minimize under-production
)

# Quality improvement - exceeding quality targets may be desired
quality_improvement = OXDeviationVar(
    name="quality_positive_deviation",
    description="Amount by which quality exceeds minimum standards",
    lower_bound=0,
    upper_bound=float('inf'),
    desired=True  # Exceeding quality standards is good
)

# Capacity utilization - some under-utilization might be acceptable
capacity_slack = OXDeviationVar(
    name="capacity_negative_deviation",
    description="Unused capacity below target utilization",
    lower_bound=0,
    upper_bound=100,  # Maximum 100% under-utilization
    desired=False  # Generally want to minimize unused capacity
)

# Display deviation characteristics
print(budget_overrun)  # Shows desired status in string representation

# Goal programming objective formulation example
undesired_deviations = [budget_overrun, production_shortfall, capacity_
↪slack]
objective_terms = [dev for dev in undesired_deviations if not dev.desired]
```

Note

- Deviation variables are typically non-negative in goal programming models
- Pairs of positive/negative deviation variables are common for each goal
- The desired flag influences objective function coefficient assignment
- String representation clearly shows both variable name and desirability status

See also

`variables.OXVariable.OXVariable` (page 134): Base variable class with bounds and relationships. `constraints.OXConstraint.OXGoalConstraint` (page 110): Goal constraints that use deviation variables. `variables.OXVariableSet.OXVariableSet` (page 143): Container for managing deviation variable collections.

`desired: bool`[265] `= False`

`__str__()`

Return enhanced string representation including desirability status.

Provides a comprehensive string representation that includes both the variable name from the parent class and the goal programming-specific desirability flag. This enhanced representation is useful for debugging, model visualization, and optimization solver interfaces.

Returns

String in format "variable_name (desired: boolean_value)" that clearly indicates both the variable identifier and its role in the goal programming objective function.

Return type

str[266]

Examples

```
desired_dev = OXDeviationVar(name="quality_surplus", desired=True)
undesired_dev = OXDeviationVar(name="cost_overrun", desired=False)

print(desired_dev)    # Output: "quality_surplus (desired: True)"
print(undesired_dev)  # Output: "cost_overrun (desired: False)"
```

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = ", name: str = ", description: str = ", value: float | int | bool = None, upper_bound: float | int = inf, lower_bound: float | int = 0, related_data: dict[str, ~uuid.UUID] = <factory>, desired: bool = False*) → None[267]

Goal Programming Examples

```
from variables import OXDeviationVar, OXVariableSet

# Create deviation variables for goal programming
goal_vars = OXVariableSet()

# Positive deviation (over-achievement) - undesirable
budget_overrun = OXDeviationVar(
```

(continues on next page)

[264] https://docs.python.org/3/library/functions.html#bool
[265] https://docs.python.org/3/library/functions.html#bool
[266] https://docs.python.org/3/library/stdtypes.html#str
[267] https://docs.python.org/3/library/constants.html#None

(continued from previous page)

```
    name="budget_deviation_positive",
    description="Amount exceeding budget target",
    lower_bound=0,
    upper_bound=float('inf'),
    desired=False  # We want to minimize this
)
goal_vars.add_object(budget_overrun)

# Negative deviation (under-achievement) - sometimes desirable
cost_savings = OXDeviationVar(
    name="cost_deviation_negative",
    description="Amount below cost target",
    lower_bound=0,
    upper_bound=float('inf'),
    desired=True  # We want to maximize savings
)
goal_vars.add_object(cost_savings)

# Quality deviation - minimize any deviation from target
quality_deviation = OXDeviationVar(
    name="quality_deviation",
    description="Deviation from quality target",
    lower_bound=0,
    upper_bound=float('inf'),
    desired=False  # Any deviation is undesirable
)
goal_vars.add_object(quality_deviation)

# Print deviation variable details
for var in goal_vars:
    print(f"{var.name}: desired={var.desired}")
    print(f"  String representation: {str(var)}")
```

Variable Collections

`class variables.OXVariableSet(`*id: ~uuid.UUID = <factory>, class_name: str = ", objects: list[~base.OXObject.OXObject] = <factory>*`)`

Bases: `OXObjectPot`

Type-safe container for managing collections of optimization variables.

This specialized container class extends OXObjectPot to provide comprehensive management of OXVariable instances with strict type enforcement and advanced querying capabilities. It serves as the primary collection mechanism for organizing decision variables in complex optimization models.

The class implements robust validation, efficient storage, and relationship-based querying to support large-scale optimization scenarios where variables need to be organized, searched, and managed based on their business relationships and mathematical properties.

Key Capabilities:

- Strict type enforcement ensuring only OXVariable instances are stored
- Relationship-based querying using variable related_data attributes
- Full iteration support with Python's standard collection protocols
- Memory-efficient storage optimized for large variable collections
- Thread-safe read operations for concurrent optimization environments

Architecture:
The container inherits from OXObjectPot to leverage proven collection management patterns while adding variable-specific functionality such as relationship querying and type validation. All operations maintain the mathematical integrity required for optimization model consistency.

Performance Characteristics:

- Variable addition: O(1) average case with type validation overhead
- Variable removal: O(n) linear search with type validation
- Relationship queries: O(n) linear scan with predicate evaluation
- Iteration: O(n) with minimal memory overhead for large collections

Thread Safety:

- Read operations (iteration, querying, length) are thread-safe
- Write operations (add, remove) require external synchronization
- Related_data modifications on contained variables need coordination

Examples

Build and query variable collections for optimization models:

```
from variables.OXVariableSet import OXVariableSet
from variables.OXVariable import OXVariable
from uuid import uuid4

# Create variable set for production planning
production_vars = OXVariableSet()

# Create variables for different products and facilities
facility1_id = uuid4()
facility2_id = uuid4()
product_a_id = uuid4()
product_b_id = uuid4()

# Production variable for Product A at Facility 1
var_a1 = OXVariable(
    name="prod_A_facility1",
    description="Production of Product A at Facility 1",
    lower_bound=0,
    upper_bound=1000
)
```

(continues on next page)

(continued from previous page)

```
var_a1.related_data["facility"] = facility1_id
var_a1.related_data["product"] = product_a_id

# Production variable for Product B at Facility 2
var_b2 = OXVariable(
    name="prod_B_facility2",
    description="Production of Product B at Facility 2",
    lower_bound=0,
    upper_bound=800
)
var_b2.related_data["facility"] = facility2_id
var_b2.related_data["product"] = product_b_id

# Add variables to the set
production_vars.add_object(var_a1)
production_vars.add_object(var_b2)

# Query variables by facility
facility1_vars = production_vars.query(facility=facility1_id)
print(f"Facility 1 variables: {[v.name for v in facility1_vars]}")

# Query variables by product type
product_a_vars = production_vars.query(product=product_a_id)
print(f"Product A variables: {[v.name for v in product_a_vars]}")

# Iterate through all variables
total_capacity = sum(var.upper_bound for var in production_vars)
print(f"Total production capacity: {total_capacity}")
```

Note

- Type validation occurs at runtime during add/remove operations
- Query operations scan all variables for matching relationships
- Container operations maintain the same semantics as OXObjectPot
- Variables can be queried by any combination of related_data attributes

See also

`base.OXObjectPot.OXObjectPot`: Base container class with collection operations. `variables.OXVariable.OXVariable` (page 134): Variable type managed by this container. `base.OXObject`: Base object type for UUID and serialization support.

`add_object`(*obj: OXObject*)

Add an OXVariable instance to the variable collection.

This method performs type validation to ensure only OXVariable instances are added to the set, maintaining type safety and collection integrity. The validation occurs before delegation to the parent container's add operation, preventing invalid state and ensuring optimization model consistency.

The method enforces the container's type invariant that all contained objects must be optimization variables, which is essential for the specialized querying and management operations provided by this class.

Parameters

`obj` (`OXObject`) – The variable object to add to the collection. Must be an instance of OXVariable or its subclasses. The object will be stored and can be retrieved through iteration or relationship-based queries.

Raises

`OXception` – If the provided object is not an instance of OXVariable. This strict type checking prevents runtime errors and maintains the mathematical integrity of variable collections.

Performance:

- Time complexity: O(1) average case for the type check and container addition
- Space complexity: O(1) additional memory for the new variable reference
- Type validation adds minimal overhead compared to container operations

Note

- Duplicate variables (same UUID) will be handled by the parent container
- The variable's related_data can be modified after addition for querying
- Type validation is strict and does not allow duck-typing or coercion

Examples

Add variables with proper type validation:

```
var_set = OXVariableSet()

# Valid addition - OXVariable instance
production_var = OXVariable(name="production", lower_bound=0)
var_set.add_object(production_var)  # Success

# Valid addition - OXVariable subclass
deviation_var = OXDeviationVar(name="deviation")
var_set.add_object(deviation_var)  # Success

# Invalid addition - wrong type
from base.OXObject import OXObject
generic_obj = OXObject()
try:
```

(continues on next page)

(continued from previous page)

```
    var_set.add_object(generic_obj)  # Raises OXception
except OXception as e:
    print("Type validation prevented invalid addition")
```

See also

`remove_object()` (page 147): Type-safe variable removal from the collection.
`query()` (page 148): Relationship-based variable querying capabilities.

`remove_object(`*obj: OXObject*`)`

Remove an OXVariable instance from the variable collection.

This method performs type validation to ensure only OXVariable instances are removed from the set, maintaining type safety and preventing invalid removal operations. The validation occurs before delegation to the parent container's removal operation.

Parameters

`obj` (`OXObject`) – The variable object to remove from the collection. Must be an instance of OXVariable that is currently stored in the set. The object will be completely removed from the collection.

Raises

- `OXception` – If the provided object is not an instance of OXVariable. This maintains the type safety invariant of the container.
- `ValueError`[268] – If the object is not currently in the set. This is raised by the parent container when attempting to remove a non-existent object.

Performance:

- Time complexity: O(n) where n is the number of variables (linear search)
- Space complexity: O(1) as removal only deallocates the reference
- Type validation overhead is minimal compared to the search operation

Note

- Removal is based on object identity (UUID), not value equality
- After removal, the variable can no longer be queried or iterated
- Related data relationships are not automatically cleaned up

Examples

Remove variables with proper validation:

```python
var_set = OXVariableSet()
production_var = OXVariable(name="production")
var_set.add_object(production_var)

# Valid removal - variable exists in set
var_set.remove_object(production_var)  # Success

# Invalid removal - wrong type
from base.OXObject import OXObject
generic_obj = OXObject()
try:
    var_set.remove_object(generic_obj)  # Raises OXception
except OXception as e:
    print("Type validation prevented invalid removal")

# Invalid removal - variable not in set
try:
    var_set.remove_object(production_var)  # Raises ValueError
except ValueError as e:
    print("Variable not found in set")
```

See also

`add_object()` (page 145): Type-safe variable addition to the collection. `query()` (page 148): Find variables before removal operations.

`query(**kwargs)` → list[269][OXObject]

Search for variables based on their relationship data attributes.

This method provides powerful relationship-based querying capabilities by searching through all variables in the collection and returning those that match the specified related_data criteria. Variables are included in the result only if they contain ALL specified relationship key-value pairs.

The query system enables complex filtering scenarios essential for large-scale optimization models where variables need to be organized and accessed based on their business relationships, such as customers, facilities, products, time periods, or other domain-specific entities.

Query Logic:

- Variables must have ALL specified keys in their related_data dictionary
- Values must match exactly (no partial matching or type coercion)
- Variables without any matching keys are excluded from results
- Empty queries (no kwargs) return no results for safety

Parameters

`**kwargs` – Key-value pairs to match against variables' related_data dictionaries. Keys represent relationship types (e.g., 'customer', 'facility') and values are the corresponding UUID identifiers. A variable is included in results only if its related_data contains ALL specified key-value pairs with exact matches.

Returns

A list of OXVariable instances that match ALL query criteria.
The list is empty if no variables match or if no query parameters are provided. Variables are returned in the order they are stored in the container.

Return type

list[270][OXObject]

Raises

`OXception` – If a non-OXVariable object is encountered during the search. This should never occur due to type validation but provides a safety check against container corruption.

Performance:

- Time complexity: O(n × k) where n is variables count and k is query criteria count
- Space complexity: O(m) where m is the number of matching variables
- Linear scan through all variables makes this suitable for moderate-sized collections

Note

- Query parameters are case-sensitive and require exact key matches
- UUID values are compared for exact equality (no fuzzy matching)
- Variables can be queried by any combination of related_data attributes
- Results maintain references to original variables (not copies)

Examples

Query variables using relationship-based filtering:

```
from variables.OXVariableSet import OXVariableSet
from variables.OXVariable import OXVariable
from uuid import uuid4

# Set up variables with relationships
var_set = OXVariableSet()

customer1_id = uuid4()
customer2_id = uuid4()
```

(continues on next page)

(continued from previous page)

```
facility1_id = uuid4()
facility2_id = uuid4()

# Create variables for different customer-facility combinations
var1 = OXVariable(name="prod_c1_f1")
var1.related_data["customer"] = customer1_id
var1.related_data["facility"] = facility1_id

var2 = OXVariable(name="prod_c1_f2")
var2.related_data["customer"] = customer1_id
var2.related_data["facility"] = facility2_id

var3 = OXVariable(name="prod_c2_f1")
var3.related_data["customer"] = customer2_id
var3.related_data["facility"] = facility1_id

# Add to set
for var in [var1, var2, var3]:
    var_set.add_object(var)

# Query by single criterion
customer1_vars = var_set.query(customer=customer1_id)
print(f"Customer 1 variables: {len(customer1_vars)}")  # Output: 2

# Query by multiple criteria (AND operation)
specific_vars = var_set.query(customer=customer1_id,
↪facility=facility1_id)
print(f"Customer 1 at Facility 1: {len(specific_vars)}")  # Output: 1

# Empty query returns no results
empty_result = var_set.query()
print(f"Empty query result: {len(empty_result)}")  # Output: 0
```

See also

`add_object()` (page 145): Add variables with relationship data for querying. `search_by_function()`: Lower-level search functionality from parent class.

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', objects: list[~base.OXObject.OXObject] = <factory>*) → None[271]

268 https://docs.python.org/3/library/exceptions.html#ValueError
269 https://docs.python.org/3/library/stdtypes.html#list
270 https://docs.python.org/3/library/stdtypes.html#list
271 https://docs.python.org/3/library/constants.html#None

9.8.2 Variable Architecture

OptiX variables are designed for optimization problems with the following key features:

- **Bounds Management**: All variables support lower and upper bounds with automatic validation
- **Relationship Tracking**: UUID-based linking to business entities through related_data
- **Value Storage**: Optional value assignment for initial solutions or fixed variables
- **Goal Programming**: Specialized deviation variables with desirability indicators

9.8.3 Examples

Creating Variables

```
from variables import OXVariable
from uuid import uuid4

# Create a basic decision variable
production_rate = OXVariable(
    name="production_rate",
    description="Daily production rate (units/day)",
    lower_bound=0.0,
    upper_bound=1000.0,
    value=500.0  # Optional initial value
)

# Create a variable with entity relationships
facility_id = uuid4()
machine_hours = OXVariable(
    name="machine_hours",
    description="Machine operating hours per day",
    lower_bound=0,
    upper_bound=24,
    related_data={"facility": facility_id}
)

# Variables auto-generate names if not provided
auto_var = OXVariable(
    description="Automatically named variable",
    lower_bound=0,
    upper_bound=100
)
print(f"Auto-generated name: {auto_var.name}")  # Will be "var_<uuid>"

print(f"Production rate bounds: [{production_rate.lower_bound}, {production_rate.
↪upper_bound}]")
print(f"Machine hours facility: {machine_hours.related_data.get('facility')}")
```

Variable Sets and Collections

```
from variables import OXVariableSet, OXVariable

# Create variable set
variables = OXVariableSet()

# Add variables for production planning
products = ["A", "B", "C"]
factories = ["Factory_1", "Factory_2"]

for product in products:
    for factory in factories:
        var = OXVariable(
            name=f"production_{product}_{factory}",
            description=f"Production of {product} at {factory}",
            lower_bound=0,
            upper_bound=500,
            variable_type="continuous"
        )
        variables.add_variable(var)

print(f"Total variables: {len(variables)}")

# Search for specific variables
product_a_vars = variables.search_by_function(
    lambda v: "product_A" in v.name
)
print(f"Product A variables: {len(product_a_vars)}")

# Search by name pattern
factory_1_vars = variables.search_by_name("Factory_1")
print(f"Factory 1 variables: {len(factory_1_vars)}")

# Search by type
continuous_vars = variables.search_by_type("continuous")
print(f"Continuous variables: {len(continuous_vars)}")
```

Advanced Variable Management

```
def create_transportation_variables(origins, destinations, products):
    """Create variables for a transportation problem."""

    variables = OXVariableSet()

    for origin in origins:
        for destination in destinations:
            for product in products:
                # Flow variable
                flow_var = OXVariable(
```

(continues on next page)

(continued from previous page)

```
                    name=f"flow_{origin.id}_{destination.id}_{product.id}",
                    description=f"Flow of {product.name} from {origin.name} to
↪{destination.name}",
                    lower_bound=0,
                    upper_bound=min(origin.capacity, destination.demand[product.
↪id]),
                    related_data={
                        "origin": origin.id,
                        "destination": destination.id,
                        "product": product.id
                    }
                )
                variables.add_object(flow_var)

                # Binary variable for route activation
                route_var = OXVariable(
                    name=f"route_{origin.id}_{destination.id}_{product.id}",
                    description=f"Route activation for {product.name} from
↪{origin.name} to {destination.name}",
                    lower_bound=0,
                    upper_bound=1,
                    related_data={
                        "origin": origin.id,
                        "destination": destination.id,
                        "product": product.id,
                        "is_binary": True
                    }
                )
                variables.add_object(route_var)

    return variables

# Usage example
# variables = create_transportation_variables(warehouses, customers, products)
```

Working with Variable Values

```
from variables import OXVariable

# Create variable with initial value
machine_capacity = OXVariable(
    name="machine_capacity",
    description="Machine processing capacity",
    lower_bound=0,
    upper_bound=1000,
    value=500  # Initial value
)

# Update value
```

(continues on next page)

(continued from previous page)

```
machine_capacity.value = 800
print(f"Current value: {machine_capacity.value}")

# Update bounds directly
machine_capacity.lower_bound = 100
machine_capacity.upper_bound = 1200
print(f"New bounds: [{machine_capacity.lower_bound}, {machine_capacity.upper_
↪bound}]")

# Manual validation of values
test_values = [50, 150, 1100, 1300]
for test_value in test_values:
    is_valid = machine_capacity.lower_bound <= test_value <= machine_capacity.
↪upper_bound
    print(f"Value {test_value} is within bounds: {is_valid}")
```

Variable Search and Filtering

```
def demonstrate_variable_search(variable_set):
    """Demonstrate various variable search capabilities."""
    from uuid import uuid4

    print("Variable Search Examples")
    print("=" * 40)

    # Search by name pattern using list comprehension
    production_vars = [v for v in variable_set if "production" in v.name.lower()]
    print(f"Production variables: {len(production_vars)}")

    # Search by bounds
    bounded_vars = [v for v in variable_set if 0 <= v.lower_bound and v.upper_
↪bound <= 100]
    print(f"Variables bounded between 0 and 100: {len(bounded_vars)}")

    # Search by description keywords
    cost_vars = [v for v in variable_set if "cost" in v.description.lower()]
    print(f"Cost-related variables: {len(cost_vars)}")

    # Query by relationships (if you've set up related_data)
    # Example: Find all variables related to a specific facility
    facility_id = uuid4()  # Example facility ID
    facility_vars = variable_set.query(facility=facility_id)
    print(f"Variables for facility {facility_id}: {len(facility_vars)}")

    # Find variables by checking related_data for binary indicators
    binary_vars = [v for v in variable_set if v.related_data.get("is_binary",
↪False)]
    print(f"Binary variables: {len(binary_vars)}")
```

(continues on next page)

(continued from previous page)

```
    # Get specific variables by name
    target_names = ["production_A_Factory_1", "production_B_Factory_1"]
    specific_vars = [v for v in variable_set if v.name in target_names]
    print(f"Specific named variables found: {len(specific_vars)}")
```

Dynamic Variable Creation

```
def create_scheduling_variables(tasks, time_periods, resources):
    """Create variables for a scheduling problem with dynamic structure."""

    variables = OXVariableSet()

    # Task assignment variables (binary)
    for task in tasks:
        for period in time_periods:
            if task.can_start_in_period(period):
                var = OXVariable(
                    name=f"start_{task.id}_period_{period}",
                    description=f"Task {task.name} starts in period {period}",
                    lower_bound=0,
                    upper_bound=1,
                    related_data={"task": task.id, "period": period, "is_binary
↪": True}
                )
                variables.add_object(var)

    # Resource allocation variables (continuous)
    for task in tasks:
        for resource in resources:
            if task.can_use_resource(resource):
                var = OXVariable(
                    name=f"allocation_{task.id}_{resource.id}",
                    description=f"Allocation of {resource.name} to {task.name}",
                    lower_bound=0,
                    upper_bound=resource.capacity,
                    related_data={"task": task.id, "resource": resource.id}
                )
                variables.add_object(var)

    # Completion time variables (continuous)
    for task in tasks:
        var = OXVariable(
            name=f"completion_{task.id}",
            description=f"Completion time of {task.name}",
            lower_bound=task.earliest_start,
            upper_bound=task.latest_finish,
            related_data={"task": task.id}
        )
        variables.add_object(var)
```

(continues on next page)

(continued from previous page)

```
    return variables
```

Variable Validation and Analysis

```
def validate_variable_set(variable_set):
    """Validate a variable set for common issues."""

    issues = []

    # Check for duplicate names
    names = [var.name for var in variable_set]
    duplicate_names = set([name for name in names if names.count(name) > 1])

    if duplicate_names:
        issues.append(f"Duplicate variable names: {duplicate_names}")

    # Check for invalid bounds
    invalid_bounds_vars = []
    for var in variable_set:
        if var.lower_bound > var.upper_bound:
            invalid_bounds_vars.append(var.name)

    if invalid_bounds_vars:
        issues.append(f"Variables with invalid bounds: {invalid_bounds_vars}")

    # Check for extremely large bounds (potential numerical issues)
    large_bound_vars = []
    for var in variable_set:
        if abs(var.lower_bound) > 1e6 or abs(var.upper_bound) > 1e6:
            large_bound_vars.append(var.name)

    if large_bound_vars:
        issues.append(f"Variables with very large bounds: {large_bound_vars}")

    # Check for missing descriptions
    no_description_vars = [var.name for var in variable_set if not var.
↪description]

    if no_description_vars:
        issues.append(f"Variables without descriptions: {no_description_vars}")

    return issues

def analyze_variable_structure(variable_set):
    """Analyze the structure and properties of a variable set."""

    analysis = {
        'total_variables': len(variable_set),
```

(continues on next page)

(continued from previous page)

```
        'variable_types': {},
        'bound_statistics': {
            'min_lower_bound': float('inf'),
            'max_lower_bound': float('-inf'),
            'min_upper_bound': float('inf'),
            'max_upper_bound': float('-inf'),
            'unbounded_variables': 0
        },
        'fixed_variables': 0,
        'name_patterns': {}
    }

    # Since OXVariable doesn't have variable_type attribute, we can infer from
↪bounds
    for var in variable_set:
        # Infer type based on bounds and related_data
        if var.related_data.get("is_binary", False) or (var.lower_bound == 0
↪and var.upper_bound == 1):
            var_type = "binary"
        elif var.lower_bound == int(var.lower_bound) and var.upper_bound ==
↪int(var.upper_bound):
            var_type = "integer"
        else:
            var_type = "continuous"
        analysis['variable_types'][var_type] = analysis['variable_types'].
↪get(var_type, 0) + 1

        # Bound statistics
        if var.lower_bound != float('-inf'):
            analysis['bound_statistics']['min_lower_bound'] = min(
                analysis['bound_statistics']['min_lower_bound'], var.lower_bound
            )
            analysis['bound_statistics']['max_lower_bound'] = max(
                analysis['bound_statistics']['max_lower_bound'], var.lower_bound
            )

        if var.upper_bound != float('inf'):
            analysis['bound_statistics']['min_upper_bound'] = min(
                analysis['bound_statistics']['min_upper_bound'], var.upper_bound
            )
            analysis['bound_statistics']['max_upper_bound'] = max(
                analysis['bound_statistics']['max_upper_bound'], var.upper_bound
            )

        if (var.lower_bound == float('-inf') or var.upper_bound == float('inf')):
            analysis['bound_statistics']['unbounded_variables'] += 1

        # Check if variable has a fixed value
        if var.value is not None:
```

(continues on next page)

(continued from previous page)

```
            analysis['fixed_variables'] += 1

        # Name patterns
        name_parts = var.name.split('_')
        if len(name_parts) > 1:
            pattern = name_parts[0]
            analysis['name_patterns'][pattern] = analysis['name_patterns'].
↪get(pattern, 0) + 1

    return analysis

def print_variable_analysis(analysis):
    """Print formatted variable analysis."""

    print("Variable Set Analysis")
    print("=" * 50)
    print(f"Total Variables: {analysis['total_variables']}")
    print(f"Fixed Variables: {analysis['fixed_variables']}")

    print("\nVariable Types:")
    for var_type, count in analysis['variable_types'].items():
        percentage = (count / analysis['total_variables']) * 100
        print(f"  {var_type}: {count} ({percentage:.1f}%)")

    print("\nBound Statistics:")
    bounds = analysis['bound_statistics']
    if bounds['min_lower_bound'] != float('inf'):
        print(f"  Lower bounds range: [{bounds['min_lower_bound']}, {bounds['max_
↪lower_bound']}]")
    if bounds['min_upper_bound'] != float('inf'):
        print(f"  Upper bounds range: [{bounds['min_upper_bound']}, {bounds['max_
↪upper_bound']}]")
    print(f"  Unbounded variables: {bounds['unbounded_variables']}")

    print("\nName Patterns:")
    sorted_patterns = sorted(analysis['name_patterns'].items(), key=lambda x:
↪x[1], reverse=True)
    for pattern, count in sorted_patterns[:10]:  # Top 10 patterns
        print(f"  {pattern}_*: {count} variables")

# Usage examples
issues = validate_variable_set(variable_set)
if issues:
    print("Variable Set Issues:")
    for issue in issues:
        print(f"  - {issue}")
else:
    print("Variable set validation passed!")
```

(continues on next page)

(continued from previous page)

```
analysis = analyze_variable_structure(variable_set)
print_variable_analysis(analysis)
```

Variable Transformation and Scaling

```
def scale_variables(variable_set, scaling_factor=1.0):
    """Scale variable bounds by a given factor."""

    scaled_variables = OXVariableSet()

    for var in variable_set:
        scaled_var = OXVariable(
            name=var.name,
            description=var.description,
            lower_bound=var.lower_bound * scaling_factor,
            upper_bound=var.upper_bound * scaling_factor,
            value=var.value * scaling_factor if var.value is not None else None,
            related_data=var.related_data.copy()
        )
        scaled_variables.add_object(scaled_var)

    return scaled_variables

def normalize_variables(variable_set):
    """Normalize variables to [0, 1] range."""

    normalized_variables = OXVariableSet()

    for var in variable_set:
        # Check if variable appears to be binary
        is_binary = var.related_data.get("is_binary", False) or (var.lower_
↪bound == 0 and var.upper_bound == 1)

        if not is_binary:
            # Normalize to [0, 1] range
            normalized_var = OXVariable(
                name=f"norm_{var.name}",
                description=f"Normalized {var.description}",
                lower_bound=0.0,
                upper_bound=1.0
            )
            normalized_variables.add_object(normalized_var)
        else:
            # Keep binary variables as is
            normalized_variables.add_object(var)

    return normalized_variables

def create_auxiliary_variables(base_variables, auxiliary_type="slack"):
```

(continues on next page)

(continued from previous page)

```
    """Create auxiliary variables (slack, surplus, artificial) for constraints.""
↪"

    auxiliary_variables = OXVariableSet()

    for i, base_var in enumerate(base_variables):
        if auxiliary_type == "slack":
            aux_var = OXVariable(
                name=f"slack_{i}",
                description=f"Slack variable for constraint {i}",
                lower_bound=0,
                upper_bound=float('inf')
            )
        elif auxiliary_type == "surplus":
            aux_var = OXVariable(
                name=f"surplus_{i}",
                description=f"Surplus variable for constraint {i}",
                lower_bound=0,
                upper_bound=float('inf')
            )
        elif auxiliary_type == "artificial":
            aux_var = OXVariable(
                name=f"artificial_{i}",
                description=f"Artificial variable for constraint {i}",
                lower_bound=0,
                upper_bound=float('inf')
            )

        auxiliary_variables.add_object(aux_var)

    return auxiliary_variables
```

Performance Optimization

```
def optimize_variable_access(variable_set):
    """Optimize variable set for faster access patterns."""

    # Create lookup dictionaries for O(1) access
    name_lookup = {var.name: var for var in variable_set}
    id_lookup = {var.id: var for var in variable_set}
    type_lookup = {}

    # Group variables by inferred type
    for var in variable_set:
        # Infer type from bounds and related_data
        if var.related_data.get("is_binary", False) or (var.lower_bound == 0
↪and var.upper_bound == 1):
            var_type = "binary"
        elif var.lower_bound == int(var.lower_bound) and var.upper_bound ==
```

(continues on next page)

(continued from previous page)

```
↪int(var.upper_bound):
                var_type = "integer"
            else:
                var_type = "continuous"

            if var_type not in type_lookup:
                type_lookup[var_type] = []
            type_lookup[var_type].append(var)

    # Create optimized variable set with fast lookups
    class OptimizedVariableSet(OXVariableSet):
        def __init__(self, variables):
            super().__init__()
            self.data = list(variables)
            self._name_lookup = name_lookup
            self._id_lookup = id_lookup
            self._type_lookup = type_lookup

        def get_variable_by_name_fast(self, name):
            return self._name_lookup.get(name)

        def get_variable_by_id_fast(self, var_id):
            return self._id_lookup.get(var_id)

        def get_variables_by_type_fast(self, var_type):
            return self._type_lookup.get(var_type, [])

    return OptimizedVariableSet(variable_set)

# Usage
optimized_vars = optimize_variable_access(variable_set)
var = optimized_vars.get_variable_by_name_fast("production_A_Factory_1")
```

Variable Export and Import

```
import json
from datetime import datetime

def export_variables_to_json(variable_set, filename=None):
    """Export variable set to JSON format."""

    if filename is None:
        filename = f"variables_{datetime.now().strftime('%Y%m%d_%H%M%S')}.json"

    export_data = {
        'metadata': {
            'export_date': datetime.now().isoformat(),
            'total_variables': len(variable_set),
            'optix_version': '1.0.0'
```

(continues on next page)

(continued from previous page)

```
        },
        'variables': []
    }

    for var in variable_set:
        var_data = {
            'id': str(var.id),
            'name': var.name,
            'description': var.description,
            'lower_bound': var.lower_bound if var.lower_bound != float('-inf')␣
↪else None,
            'upper_bound': var.upper_bound if var.upper_bound != float('inf')␣
↪else None,
            'value': var.value,
            'related_data': {k: str(v) for k, v in var.related_data.items()}
        }
        export_data['variables'].append(var_data)

    with open(filename, 'w') as f:
        json.dump(export_data, f, indent=2)

    return filename

def import_variables_from_json(filename):
    """Import variable set from JSON format."""

    with open(filename, 'r') as f:
        import_data = json.load(f)

    variable_set = OXVariableSet()

    for var_data in import_data['variables']:
        # Reconstruct related_data with UUIDs
        from uuid import UUID
        related_data = {}
        for k, v in var_data.get('related_data', {}).items():
            try:
                related_data[k] = UUID(v)
            except:
                related_data[k] = v

        var = OXVariable(
            name=var_data['name'],
            description=var_data['description'],
            lower_bound=var_data.get('lower_bound', 0),
            upper_bound=var_data.get('upper_bound', float('inf')),
            value=var_data.get('value'),
            related_data=related_data
        )
```

(continues on next page)

(continued from previous page)

```
        variable_set.add_object(var)

    return variable_set

# Usage
filename = export_variables_to_json(variable_set)
print(f"Variables exported to {filename}")

imported_variables = import_variables_from_json(filename)
print(f"Imported {len(imported_variables)} variables")
```

9.8.4 See Also

- *Problem Module* (page 81) - Problem classes that use variables
- *Constraints Module* (page 105) - Constraint definitions that reference variables
- *Data Module* (page 205) - Database integration for variable creation
- ../user_guide/variables - Advanced variable management guide

9.9 Solvers Module

The solvers module provides solver interfaces and implementations for solving optimization problems. OptiX supports multiple solvers through a unified interface, allowing easy switching between different optimization engines.

9.9.1 Solver Factory

`solvers.solve`(*problem:* OXCSPProblem *(page 81)*, *solver:* *str*[272], ***kwargs*)

Unified optimization problem solving interface with multi-solver support.

This function serves as the primary entry point for solving optimization problems within the OptiX framework, providing a standardized interface that abstracts away solver-specific implementation details while ensuring consistent problem setup, solving workflows, and solution extraction across different optimization engines and algorithmic approaches.

The function implements a comprehensive solving pipeline that automatically handles variable creation, constraint translation, objective function configuration, and solution extraction, enabling users to focus on problem modeling rather than solver-specific integration complexities.

Solving Pipeline:

The function orchestrates a standardized solving workflow:

1. **Solver Validation**: Verifies solver availability and compatibility with the specified problem type and configuration parameters

2. **Solver Instantiation**: Creates solver instance with custom parameters and configuration options for performance tuning and behavior control
3. **Variable Setup**: Translates OptiX decision variables to solver-specific variable representations with proper bounds, types, and naming conventions
4. **Constraint Translation**: Converts OptiX constraints to native solver constraint formats with accurate coefficient handling and operator mapping
5. **Special Constraint Handling**: Processes advanced constraint types including multiplicative, division, modulo, and conditional constraints using solver-specific implementation strategies
6. **Objective Configuration**: Sets up optimization objectives for linear and goal programming problems with proper minimization/maximization handling
7. **Solution Execution**: Executes the core solving algorithm with progress monitoring and early termination capabilities
8. **Result Extraction**: Retrieves optimization results and translates them to standardized OptiX solution formats for consistent analysis

Parameters

- `problem` (`OXCSPProblem` (page 81)) – The optimization problem instance to solve. Must be a properly configured OptiX problem with defined variables, constraints, and (for LP/GP problems) objective functions. Supports constraint satisfaction problems (CSP), linear programming (LP), and goal programming (GP) formulations.
- `solver` (`str`[273]) – The identifier of the optimization solver to use for problem solving. Must match a key in the _available_solvers registry. Supported values include: - 'ORTools': Google's open-source constraint programming solver - 'Gurobi': Commercial high-performance optimization solver Additional solvers may be available through plugin extensions.
- `**kwargs` – Arbitrary keyword arguments passed directly to the solver constructor for custom parameter configuration. Enables solver-specific performance tuning, algorithmic customization, and behavior control. Common parameters include: - maxTime (int): Maximum solving time in seconds - solutionCount (int): Maximum number of solutions to enumerate - equalizeDenominators (bool): Enable fractional coefficient handling - use_continuous (bool): Enable continuous variable optimization - Additional solver-specific parameters as documented by each solver

Returns

A two-element tuple containing comprehensive solving results:

- **status** (OXSolutionStatus): The termination status of the optimization process indicating solution quality and solver performance. Possible values: * OXSolutionStatus.OPTIMAL: Globally optimal solution found * OXSolutionStatus.FEASIBLE: Feasible solution found, optimality not guaranteed * OXSolutionStatus.INFEASIBLE: No feasible solution exists * OXSolutionStatus.UNBOUNDED: Problem is unbounded * OX-

SolutionStatus.TIMEOUT: Solver reached time limit * OXSolutionStatus.ERROR: Solving error occurred * OXSolutionStatus.UNKNOWN: Status cannot be determined

- **solver_obj** (OXSolverInterface): The configured solver instance used for problem solving. Provides access to all found solutions through iteration protocols, individual solution access through indexing, and solver-specific diagnostic information through logging methods. The solver maintains complete solution history and enables detailed post-solving analysis and validation.

Return type
tuple[274]

Raises

- `OXception` – Raised when the specified solver is not available in the solver registry. This typically occurs when: - The solver name is misspelled or incorrect - The solver backend is not installed or properly configured - Required dependencies for the solver are missing - The solver registration failed during framework initialization
- `Additional solver-specific exceptions may be raised during the solving process` –
- `and should be handled appropriately by calling code for robust error management.` –

Example

Basic problem solving with default parameters:

```
from problem.OXProblem import OXCSPProblem
from solvers.OXSolverFactory import solve

# Create and configure problem
problem = OXCSPProblem()
x = problem.create_decision_variable("x", 0, 10)
y = problem.create_decision_variable("y", 0, 10)
problem.create_constraint([x, y], [1, 1], "<=", 15)

# Solve with default OR-Tools configuration
status, solver = solve(problem, 'ORTools')

# Analyze results
if status == OXSolutionStatus.OPTIMAL:
    print("Found optimal solution")
    for solution in solver:
        print(f"Variables: {solution.decision_variable_values}")
elif status == OXSolutionStatus.INFEASIBLE:
    print("Problem has no feasible solution")
```

Advanced solving with custom parameters:

```python
from problem.OXProblem import OXLPProblem
from solvers.OXSolverFactory import solve

# Create linear programming problem
problem = OXLPProblem()
x = problem.create_decision_variable("x", 0, 100)
y = problem.create_decision_variable("y", 0, 100)
problem.create_constraint([x, y], [1, 1], "<=", 150)
problem.create_objective_function([x, y], [2, 3], "maximize")

# Solve with custom Gurobi parameters
status, solver = solve(
    problem,
    'Gurobi',
    use_continuous=True,
    maxTime=3600,
    optimality_gap=0.01
)

# Access optimal solution
if status == OXSolutionStatus.OPTIMAL:
    solution = solver[0]
    print(f"Optimal value: {solution.objective_function_value}")
    print(f"Variables: {solution.decision_variable_values}")
```

Performance Considerations:

- Solver instantiation overhead is minimized through efficient registry lookup
- Problem setup is optimized for large-scale problems with thousands of variables
- Memory usage scales linearly with problem size and solution enumeration
- Parallel solving capabilities depend on individual solver implementations

Solver Selection Guidelines:

- **OR-Tools**: Recommended for constraint satisfaction, discrete optimization, and problems requiring advanced constraint types with good open-source support
- **Gurobi**: Optimal for large-scale linear/quadratic programming requiring commercial-grade performance and advanced optimization algorithms
- **Custom Solvers**: Consider for specialized problem domains or when specific algorithmic approaches are required for particular optimization scenarios

Thread Safety:

The solve function creates independent solver instances for each call, ensuring thread safety for concurrent optimization operations. However, individual solver implementations may have their own thread safety considerations that should be reviewed for multi-threaded optimization scenarios.

9.9.2 Solver Interfaces

Base Interface

`class solvers.OXSolverInterface(**kwargs)`

Bases: `object`[275]

Abstract base class defining the standard interface for all optimization solver implementations.

This class establishes the fundamental contract that all concrete solver implementations must adhere to within the OptiX optimization framework. It provides a comprehensive template for integrating diverse optimization engines while maintaining consistent behavior, standardized method signatures, and uniform solution handling patterns.

The interface design follows the Template Method pattern, defining the overall algorithm structure for solving optimization problems while allowing subclasses to implement solver-specific details. This ensures consistent problem setup, solving workflows, and solution extraction across different optimization engines.

Core Responsibilities:

- **Variable Management**: Standardized creation and mapping of decision variables from OptiX problem formulations to solver-specific representations
- **Constraint Translation**: Systematic conversion of OptiX constraints to native solver constraint formats with proper operator and coefficient handling
- **Objective Configuration**: Setup of optimization objectives for linear and goal programming problems with support for minimization and maximization
- **Solution Extraction**: Comprehensive retrieval of optimization results including variable values, constraint evaluations, and solver statistics
- **Parameter Management**: Flexible configuration of solver-specific parameters for performance tuning and algorithmic customization

Interface Methods:

The class defines both abstract methods (must be implemented by subclasses) and concrete methods (provide common functionality):

Abstract Methods (require implementation): - *_create_single_variable()*: Variable creation in solver-specific format - *_create_single_constraint()*: Constraint creation with proper translation - *create_special_constraints()*: Advanced constraint type handling - *create_objective()*: Objective function setup for optimization problems - *solve()*: Core solving algorithm execution and solution extraction - *get_solver_logs()*: Diagnostic and debugging information retrieval

Concrete Methods (provided by base class): - *create_variable()*: Orchestrates creation of all problem variables - *create_constraints()*: Manages setup of all standard constraints - Collection access methods for solution enumeration and analysis

`_parameters`

Comprehensive dictionary storing solver-specific configuration parameters including algorithmic settings, performance tuning options, and behavioral controls. This

[272] https://docs.python.org/3/library/stdtypes.html#str
[273] https://docs.python.org/3/library/stdtypes.html#str
[274] https://docs.python.org/3/library/stdtypes.html#tuple

enables flexible customization of solver behavior without modifying core implementation code.

Type
Parameters

`_solutions`

Ordered collection of optimization solutions found during the solving process. Supports multiple solution enumeration for problems with multiple optimal or feasible solutions. Provides efficient access through indexing and iteration protocols.

Type
List[OXSolverSolution]

Solving Workflow:

The standard solving process follows a well-defined sequence:

1. **Problem Setup**: Variable and constraint creation from OptiX problem definition
2. **Solver Configuration**: Parameter application and algorithmic customization
3. **Objective Setup**: Optimization direction and objective function configuration
4. **Solution Process**: Core solving algorithm execution with progress monitoring
5. **Result Extraction**: Solution data retrieval and status determination
6. **Validation**: Solution verification and constraint satisfaction checking

```
# Standard workflow implementation
solver = ConcretesolverInterface(**parameters)
solver.create_variable(problem)
solver.create_constraints(problem)
solver.create_special_constraints(problem)

if isinstance(problem, OXLPProblem):
    solver.create_objective(problem)

status = solver.solve(problem)

# Access results
for solution in solver:
    analyze_solution(solution)
```

Extensibility Design:

The interface is designed to accommodate diverse optimization paradigms:

- **Linear Programming**: Continuous optimization with linear constraints
- **Integer Programming**: Discrete optimization with integer variables
- **Constraint Programming**: Logical constraint satisfaction and enumeration
- **Goal Programming**: Multi-objective optimization with priority levels
- **Heuristic Algorithms**: Approximate optimization with custom algorithms

Parameter Management:

Solver parameters enable fine-grained control over optimization behavior:

- **Algorithmic Parameters**: Solver-specific algorithm selection and tuning
- **Performance Parameters**: Time limits, memory limits, and precision settings
- **Output Parameters**: Logging levels, solution enumeration, and debugging options
- **Problem-Specific Parameters**: Customization for particular problem characteristics

Solution Management:
The interface provides comprehensive solution handling capabilities:

- **Multiple Solutions**: Support for enumeration of alternative optimal solutions
- **Solution Quality**: Status tracking and optimality verification
- **Incremental Results**: Progressive solution improvement tracking
- **Solution Comparison**: Utilities for comparing and ranking multiple solutions

Error Handling:
The interface defines consistent error handling patterns:

- **Implementation Errors**: NotImplementedError for missing abstract methods
- **Parameter Validation**: Custom exceptions for invalid solver parameters
- **Numerical Issues**: Graceful handling of solver-specific numerical problems
- **Resource Limitations**: Proper handling of memory and time limit violations

Performance Considerations:

- Solution storage uses efficient data structures for large solution sets
- Parameter dictionaries provide O(1) configuration access
- Iterator protocols enable memory-efficient solution enumeration
- Abstract method design minimizes overhead in concrete implementations

Example Implementation:
Basic structure for implementing a custom solver interface:

```python
class CustomSolverInterface(OXSolverInterface):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self._native_solver = initialize_custom_solver()
        self._var_mapping = {}

    def _create_single_variable(self, var: OXVariable):
        native_var = self._native_solver.add_variable(
            name=var.name,
            lower_bound=var.lower_bound,
            upper_bound=var.upper_bound
        )
        self._var_mapping[var.id] = native_var

    def solve(self, prb: OXCSPProblem) -> OXSolutionStatus:
```

(continues on next page)

(continued from previous page)

```
        status = self._native_solver.solve()
        if status == 'optimal':
            solution = self._extract_solution()
            self._solutions.append(solution)
            return OXSolutionStatus.OPTIMAL
        return OXSolutionStatus.UNKNOWN
```

Note

Concrete implementations should carefully handle solver-specific exceptions and translate them to appropriate OXSolutionStatus values for consistent error reporting across the framework.

`__init__`(***kwargs*)

Initialize the solver interface with optional parameters.

Parameters

`**kwargs` – Solver-specific parameters passed as keyword arguments.

`create_variable`(*prb: OXCSPProblem (page 81)*)

Create all variables from the problem in the solver.

Parameters

`prb` (`OXCSPProblem` (page 81)) – The problem containing variables to create.

`create_constraints`(*prb: OXCSPProblem (page 81)*)

Create all regular constraints from the problem in the solver.

This method creates all constraints except those that are part of special constraints (which are handled separately).

Parameters

`prb` (`OXCSPProblem` (page 81)) – The problem containing constraints to create.

`create_special_constraints`(*prb: OXCSPProblem (page 81)*)

Create all special constraints from the problem in the solver.

Parameters

`prb` (`OXCSPProblem` (page 81)) – The problem containing special constraints to create.

Raises

`NotImplementedError`[276] – Must be implemented by subclasses.

`create_objective`(*prb: OXLPProblem (page 88)*)

Create the objective function in the solver.

Parameters

`prb` (`OXLPProblem` (page 88)) – The linear programming problem containing the objective function.

Raises
`NotImplementedError`[277] – Must be implemented by subclasses.

`solve`(*prb: OXCSPProblem (page 81)*) → OXSolutionStatus

Solve the optimization problem.

Parameters
`prb` (`OXCSPProblem` (page 81)) – The problem to solve.

Returns
The status of the solution process.

Return type
OXSolutionStatus

Raises
`NotImplementedError`[278] – Must be implemented by subclasses.

`get_solver_logs`() → List[279][str[280] | List[281][str[282]] | None[283]

Get solver-specific logs and debugging information.

Returns
Solver logs if available, None otherwise.

Return type
Optional[LogsType]

Raises
`NotImplementedError`[284] – Must be implemented by subclasses.

`__getitem__`(*item*) → OXSolverSolution

Get a solution by index.

Parameters
`item` – The index of the solution to retrieve.

Returns
The solution at the specified index.

Return type
OXSolverSolution

`__len__`() → int[285]

Get the number of solutions found.

Returns
The number of solutions in the solution list.

Return type
int[286]

`__iter__`() → Iterator[287][OXSolverSolution]

Iterate over all solutions.

Returns
An iterator over the solutions.

Return type
Iterator[OXSolverSolution]

`property parameters: Dict[288][str[289], Any[290]]`

Get the solver parameters.

Returns
Dictionary of solver parameters.

Return type
Parameters

Warning

Users can modify these parameters, but validation mechanisms should be implemented to ensure parameters are valid for the specific solver.

OR-Tools Solver

OR-Tools Solver Integration Module

This module provides comprehensive integration between the OptiX optimization framework and Google's OR-Tools constraint programming solver. It implements the OptiX solver interface using OR-Tools' CP-SAT engine to enable solving complex discrete optimization problems including constraint satisfaction, integer programming, and goal programming.

The module serves as a critical component of OptiX's multi-solver architecture, offering high-performance constraint programming capabilities alongside other solver backends like Gurobi for different optimization scenarios.

Architecture:

- **Solver Interface**: Complete implementation of OXSolverInterface for OR-Tools
- **Constraint Programming**: Leverages CP-SAT for discrete optimization excellence
- **Multi-Problem Support**: Handles CSP, LP, and GP problem types seamlessly
- **Advanced Constraints**: Supports complex non-linear constraint relationships

Key Components:

- **OXORToolsSolverInterface**: Primary solver implementation class providing complete integration with OR-Tools CP-SAT solver including variable management, constraint translation, objective handling, and solution extraction capabilities

[275] https://docs.python.org/3/library/functions.html#object
[276] https://docs.python.org/3/library/exceptions.html#NotImplementedError
[277] https://docs.python.org/3/library/exceptions.html#NotImplementedError
[278] https://docs.python.org/3/library/exceptions.html#NotImplementedError
[279] https://docs.python.org/3/library/typing.html#typing.List
[280] https://docs.python.org/3/library/stdtypes.html#str
[281] https://docs.python.org/3/library/typing.html#typing.List
[282] https://docs.python.org/3/library/stdtypes.html#str
[283] https://docs.python.org/3/library/constants.html#None
[284] https://docs.python.org/3/library/exceptions.html#NotImplementedError
[285] https://docs.python.org/3/library/functions.html#int
[286] https://docs.python.org/3/library/functions.html#int
[287] https://docs.python.org/3/library/typing.html#typing.Iterator
[288] https://docs.python.org/3/library/typing.html#typing.Dict
[289] https://docs.python.org/3/library/stdtypes.html#str
[290] https://docs.python.org/3/library/typing.html#typing.Any

Solver Capabilities:

- **Variable Types**: Boolean and bounded integer decision variables with automatic type detection based on variable bounds and mathematical properties
- **Linear Constraints**: Full support for relational operators (=, <=, >=, <, >) with efficient constraint expression evaluation and validation
- **Special Constraints**: Advanced non-linear relationships including:
  - **Multiplicative**: Product relationships between multiple variables
  - **Division/Modulo**: Integer division and remainder operations for discrete math
  - **Summation**: Explicit sum constraints for complex variable relationships
  - **Conditional**: If-then-else logic with indicator variables for decision modeling
- **Objective Functions**: Optimization support for minimization and maximization with linear and goal programming objective types
- **Multi-Solution Enumeration**: Configurable solution collection with callback mechanisms for exploring solution spaces and alternative optima
- **Performance Tuning**: Comprehensive parameter configuration for time limits, solution counts, and algorithmic behavior customization

Mathematical Features:

- **Float Coefficient Handling**: Automatic denominator equalization for fractional weights enabling seamless integration of real-valued problem formulations
- **Integer Programming**: Native support for discrete optimization with advanced branching and cutting plane algorithms from OR-Tools
- **Constraint Propagation**: Sophisticated constraint propagation techniques for efficient problem space reduction and faster solving

Configuration Parameters:

The solver accepts multiple parameters for fine-tuning performance and behavior:

- **equalizeDenominators** (bool): Enables automatic conversion of float coefficients to integers using common denominator calculation, allowing OR-Tools to handle fractional weights in constraints and objectives. Default: False
- **solutionCount** (int): Maximum number of solutions to enumerate during solving. Higher values enable comprehensive solution space exploration but increase computational overhead. Default: 1
- **maxTime** (int): Maximum solving time in seconds before automatic termination. Prevents indefinite solving on computationally difficult problem instances. Default: 600 seconds (10 minutes)

Integration Patterns:

The module follows OptiX's standardized solver integration patterns for consistent usage across different solver backends:

```
from problem.OXProblem import OXCSPProblem, OXLPProblem
from solvers.ortools import OXORToolsSolverInterface
from solvers.OXSolverFactory import solve
```

(continues on next page)

(continued from previous page)

```
# Direct solver instantiation approach
problem = OXCSPProblem()
# ... configure problem variables and constraints ...

solver = OXORToolsSolverInterface(
    equalizeDenominators=True,
    solutionCount=5,
    maxTime=300
)

solver.create_variables(problem)
solver.create_constraints(problem)
solver.create_special_constraints(problem)
status = solver.solve(problem)

# Factory pattern approach (recommended)
problem = OXLPProblem()
# ... configure problem ...

status, solutions = solve(problem, "ORTools",
                         equalizeDenominators=True,
                         solutionCount=10,
                         maxTime=600)
```

Performance Considerations:

- OR-Tools CP-SAT excels at discrete optimization problems with complex constraints
- Integer variable domains should be bounded for optimal performance
- Large solution enumeration (>100 solutions) may require increased time limits
- Float coefficient conversion adds preprocessing overhead but enables broader compatibility
- Special constraints leverage native CP-SAT primitives for efficient solving

Compatibility:

- **Python Version**: Requires Python 3.7 or higher for full feature support
- **OR-Tools Version**: Compatible with OR-Tools 9.0+ constraint programming library
- **OptiX Framework**: Fully integrated with OptiX problem modeling and solving architecture
- **Operating Systems**: Cross-platform support on Windows, macOS, and Linux

Use Cases:

This solver implementation is particularly well-suited for:

- Scheduling and resource allocation problems with discrete time slots
- Combinatorial optimization problems with complex constraint relationships

- Integer programming formulations requiring advanced constraint types
- Multi-objective optimization with goal programming approaches
- Constraint satisfaction problems with large solution spaces requiring enumeration

Notes

- For continuous optimization problems, consider using the Gurobi solver interface
- Large-scale linear programming may benefit from specialized LP solver backends
- Memory usage scales with problem size and solution enumeration requirements
- Solver logs and debugging information available through get_solver_logs() method

`class solvers.ortools.OXORToolsSolverInterface(`*`**kwargs`*`)`

Bases: `OXSolverInterface` (page 167)

Concrete implementation of OptiX solver interface using Google OR-Tools CP-SAT solver.

This class provides a comprehensive bridge between OptiX's problem modeling framework and Google's OR-Tools Constraint Programming solver. It handles the complete lifecycle of problem solving from variable and constraint creation through solution extraction and analysis.

The implementation leverages OR-Tools' CP-SAT solver, which excels at discrete optimization problems including constraint satisfaction, integer programming, and mixed-integer programming. The class automatically handles type conversions, constraint translations, and solution callbacks to provide seamless integration with OptiX workflows.

Key Capabilities:

- **Variable Management**: Automatic creation and mapping of boolean and integer variables
- **Constraint Translation**: Comprehensive support for linear and special constraint types
- **Multi-Solution Handling**: Configurable solution enumeration with callback system
- **Parameter Configuration**: Flexible solver parameter management for performance tuning
- **Solution Analysis**: Complete solution data extraction including constraint violations

Solver Parameters:

The class accepts various initialization parameters to customize solver behavior:

- **equalizeDenominators** (bool): When True, enables automatic conversion of float coefficients to integers using common denominator calculation. This allows OR-Tools to handle fractional weights that would otherwise be rejected. Default: False

- **solutionCount** (int): Maximum number of solutions to collect during enumeration. Higher values enable finding multiple feasible solutions but increase solving time. Default: 1
- **maxTime** (int): Maximum solving time in seconds before termination. Prevents infinite solving on difficult instances. Default: 600 seconds (10 minutes)

`_model`

The underlying OR-Tools CP-SAT model instance that stores all variables, constraints, and objectives for the optimization problem.

Type
CpModel

`_var_mapping`

Bidirectional mapping from OptiX variable UUIDs to their corresponding OR-Tools variable objects for efficient lookup during solving.

Type
Dict[str[291], IntVar|BoolVar]

`_constraint_mapping`

Mapping from OptiX constraint UUIDs to OR-Tools constraint objects for tracking and solution analysis purposes.

Type
Dict[str[292], Constraint]

`_constraint_expr_mapping`

Mapping from constraint UUIDs to their mathematical expressions for solution value calculation.

Type
Dict[str[293], LinearExpr]

Type Support:

- **Boolean Variables**: Automatically detected from 0-1 bounds, mapped to BoolVar
- **Integer Variables**: Bounded integer variables with custom ranges, mapped to IntVar
- **Linear Expressions**: Sum of variables with integer or float coefficients
- **Special Constraints**: Non-linear relationships handled through CP-SAT primitives

Example

Comprehensive solver setup and configuration:

```python
# Create solver with advanced configuration
solver = OXORToolsSolverInterface(
    equalizeDenominators=True,  # Handle fractional coefficients
    solutionCount=10,           # Find up to 10 solutions
    maxTime=1800               # 30-minute time limit
```

(continues on next page)

(continued from previous page)

```
)

# Setup problem
solver.create_variables(problem)
solver.create_constraints(problem)
solver.create_special_constraints(problem)

if isinstance(problem, OXLPProblem):
    solver.create_objective(problem)

# Solve and analyze
status = solver.solve(problem)

if status == OXSolutionStatus.OPTIMAL:
    for i, solution in enumerate(solver):
        print(f"Solution {i+1}: {solution.decision_variable_values}")
        print(f"Objective: {solution.objective_function_value}")

# Access solver statistics
logs = solver.get_solver_logs()
```

Warning

OR-Tools CP-SAT requires integer coefficients for all constraints and objectives. When using float coefficients, the equalizeDenominators parameter must be enabled to perform automatic conversion, or an OXception will be raised during constraint creation.

Note

This implementation is optimized for discrete optimization problems. For continuous optimization or large-scale linear programming, consider using the Gurobi solver interface which may provide better performance for those problem types.

`class SolutionLimiter(`*max_solution_count: int*[294]*, solver:* OXORToolsSolverInterface *(page 179), prb:* OXCSPProblem *(page 81)*`)`

Bases: `CpSolverSolutionCallback`

Callback class to limit the number of solutions found.

This class extends CpSolverSolutionCallback to control the number of solutions collected during the solving process.

`_solution_count`

Current number of solutions found.

Type

int[295]

`_max_solution_count`

Maximum number of solutions to collect.

Type

int[296]

`_solver`

Reference to the solver interface.

Type

OXORToolsSolverInterface (page 179)

`_problem`

The problem being solved.

Type

OXCSPProblem (page 81)

`__init__`(*max_solution_count: int*[297], *solver:* OXORToolsSolverInterface *(page 179)*, *prb:* OXCSPProblem *(page 81)*)

Initialize the solution limiter callback.

Parameters

- `max_solution_count` (`int`[298]) – Maximum number of solutions to collect.
- `solver` (`OXORToolsSolverInterface` (page 179)) – Reference to the solver interface.
- `prb` (`OXCSPProblem` (page 81)) – The problem being solved.

`on_solution_callback`()

Callback method called when a solution is found.

This method creates an OXSolverSolution object with the current solution values and adds it to the solver's solution list.

Raises

`OXception` – If an unsupported special constraint type is encountered.

`__init__`(***kwargs*)

Initialize the OR-Tools solver interface.

Parameters

`**kwargs` – Solver parameters. Supported parameters: - equalizeDenominators (bool): Use denominator equalization for float handling. - solutionCount (int): Maximum number of solutions to find. - maxTime (int): Maximum solving time in seconds.

`create_objective`(*prb:* OXLPProblem *(page 88)*)

Create the objective function in the OR-Tools model.

Parameters

`prb` (`OXLPProblem` (page 88)) – The linear programming problem containing the objective function.

Raises

`OXception` – If no objective function is specified or if float weights are used without denominator equalization enabled.

`create_special_constraints`(*prb:* OXCSPProblem *(page 81)*)

Create all special constraints from the problem.

Parameters
prb (OXCSPProblem (page 81)) – The problem containing special constraints.

Raises
OXception – If an unsupported special constraint type is encountered.

`get_solver_logs()` → List[299][str[300] | List[301][str[302]] | None[303]

Get solver logs and debugging information.

Returns
Currently not implemented, returns None.

Return type
Optional[LogsType]

`solve(`*prb: OXCSPProblem (page 81)*`)` → OXSolutionStatus

Solve the optimization problem using OR-Tools CP-SAT solver.

Parameters
prb (OXCSPProblem (page 81)) – The problem to solve.

Returns
The status of the solution process.

Return type
OXSolutionStatus

Raises
OXception – If the solver returns an unexpected status.

`class solvers.ortools.OXORToolsSolverInterface(`***kwargs*`)`

Bases: OXSolverInterface (page 167)

Concrete implementation of OptiX solver interface using Google OR-Tools CP-SAT solver.

This class provides a comprehensive bridge between OptiX's problem modeling framework and Google's OR-Tools Constraint Programming solver. It handles the complete lifecycle of problem solving from variable and constraint creation through solution extraction and analysis.

The implementation leverages OR-Tools' CP-SAT solver, which excels at discrete optimization problems including constraint satisfaction, integer programming, and mixed-integer programming. The class automatically handles type conversions, constraint

291 https://docs.python.org/3/library/stdtypes.html#str
292 https://docs.python.org/3/library/stdtypes.html#str
293 https://docs.python.org/3/library/stdtypes.html#str
294 https://docs.python.org/3/library/functions.html#int
295 https://docs.python.org/3/library/functions.html#int
296 https://docs.python.org/3/library/functions.html#int
297 https://docs.python.org/3/library/functions.html#int
298 https://docs.python.org/3/library/functions.html#int
299 https://docs.python.org/3/library/typing.html#typing.List
300 https://docs.python.org/3/library/stdtypes.html#str
301 https://docs.python.org/3/library/typing.html#typing.List
302 https://docs.python.org/3/library/stdtypes.html#str
303 https://docs.python.org/3/library/constants.html#None

translations, and solution callbacks to provide seamless integration with OptiX workflows.

Key Capabilities:

- **Variable Management**: Automatic creation and mapping of boolean and integer variables
- **Constraint Translation**: Comprehensive support for linear and special constraint types
- **Multi-Solution Handling**: Configurable solution enumeration with callback system
- **Parameter Configuration**: Flexible solver parameter management for performance tuning
- **Solution Analysis**: Complete solution data extraction including constraint violations

Solver Parameters:

The class accepts various initialization parameters to customize solver behavior:

- **equalizeDenominators** (bool): When True, enables automatic conversion of float coefficients to integers using common denominator calculation. This allows OR-Tools to handle fractional weights that would otherwise be rejected. Default: False
- **solutionCount** (int): Maximum number of solutions to collect during enumeration. Higher values enable finding multiple feasible solutions but increase solving time. Default: 1
- **maxTime** (int): Maximum solving time in seconds before termination. Prevents infinite solving on difficult instances. Default: 600 seconds (10 minutes)

`_model`

The underlying OR-Tools CP-SAT model instance that stores all variables, constraints, and objectives for the optimization problem.

Type
CpModel

`_var_mapping`

Bidirectional mapping from OptiX variable UUIDs to their corresponding OR-Tools variable objects for efficient lookup during solving.

Type
Dict[str[304], IntVar|BoolVar]

`_constraint_mapping`

Mapping from OptiX constraint UUIDs to OR-Tools constraint objects for tracking and solution analysis purposes.

Type
Dict[str[305], Constraint]

`_constraint_expr_mapping`

Mapping from constraint UUIDs to their mathematical expressions for solution value calculation.

Type
Dict[str[306], LinearExpr]

Type Support:

- **Boolean Variables**: Automatically detected from 0-1 bounds, mapped to BoolVar
- **Integer Variables**: Bounded integer variables with custom ranges, mapped to IntVar
- **Linear Expressions**: Sum of variables with integer or float coefficients
- **Special Constraints**: Non-linear relationships handled through CP-SAT primitives

Example

Comprehensive solver setup and configuration:

```
# Create solver with advanced configuration
solver = OXORToolsSolverInterface(
    equalizeDenominators=True,  # Handle fractional coefficients
    solutionCount=10,           # Find up to 10 solutions
    maxTime=1800               # 30-minute time limit
)

# Setup problem
solver.create_variables(problem)
solver.create_constraints(problem)
solver.create_special_constraints(problem)

if isinstance(problem, OXLPProblem):
    solver.create_objective(problem)

# Solve and analyze
status = solver.solve(problem)

if status == OXSolutionStatus.OPTIMAL:
    for i, solution in enumerate(solver):
        print(f"Solution {i+1}: {solution.decision_variable_values}")
        print(f"Objective: {solution.objective_function_value}")

# Access solver statistics
logs = solver.get_solver_logs()
```

Warning

OR-Tools CP-SAT requires integer coefficients for all constraints and objectives. When using float coefficients, the equalizeDenominators parameter must be enabled to perform automatic conversion, or an OXception will be raised during constraint creation.

Note

This implementation is optimized for discrete optimization problems. For continuous optimization or large-scale linear programming, consider using the Gurobi solver interface which may provide better performance for those problem types.

`__init__`(***kwargs*)

Initialize the OR-Tools solver interface.

Parameters
`**kwargs` – Solver parameters. Supported parameters: - equalizeDenominators (bool): Use denominator equalization for float handling. - solutionCount (int): Maximum number of solutions to find. - maxTime (int): Maximum solving time in seconds.

`create_special_constraints`(*prb: OXCSPProblem (page 81)*)

Create all special constraints from the problem.

Parameters
`prb` (`OXCSPProblem` (page 81)) – The problem containing special constraints.

Raises
`OXception` – If an unsupported special constraint type is encountered.

`create_objective`(*prb: OXLPProblem (page 88)*)

Create the objective function in the OR-Tools model.

Parameters
`prb` (`OXLPProblem` (page 88)) – The linear programming problem containing the objective function.

Raises
`OXception` – If no objective function is specified or if float weights are used without denominator equalization enabled.

`class SolutionLimiter`(*max_solution_count: int*[307], *solver: OXORToolsSolverInterface (page 179)*, *prb: OXCSPProblem (page 81)*)

Bases: `CpSolverSolutionCallback`

Callback class to limit the number of solutions found.

This class extends CpSolverSolutionCallback to control the number of solutions collected during the solving process.

`_solution_count`

Current number of solutions found.

Type
int[308]

`_max_solution_count`

Maximum number of solutions to collect.

Type
int[309]

`_solver`

Reference to the solver interface.

Type

OXORToolsSolverInterface (page 179)

`_problem`

The problem being solved.

Type

OXCSPProblem (page 81)

`__init__`(*max_solution_count: int*[310], *solver:* OXORToolsSolverInterface *(page 179)*, *prb:* OXCSPProblem *(page 81)*)

Initialize the solution limiter callback.

Parameters

- `max_solution_count` (`int`[311]) – Maximum number of solutions to collect.
- `solver` (`OXORToolsSolverInterface` (page 179)) – Reference to the solver interface.
- `prb` (`OXCSPProblem` (page 81)) – The problem being solved.

`on_solution_callback()`

Callback method called when a solution is found.

This method creates an OXSolverSolution object with the current solution values and adds it to the solver's solution list.

Raises

`OXception` – If an unsupported special constraint type is encountered.

`solve`(*prb:* OXCSPProblem *(page 81)*) → OXSolutionStatus

Solve the optimization problem using OR-Tools CP-SAT solver.

Parameters

`prb` (`OXCSPProblem` (page 81)) – The problem to solve.

Returns

The status of the solution process.

Return type

OXSolutionStatus

Raises

`OXception` – If the solver returns an unexpected status.

`get_solver_logs()` → List[312][str[313] | List[314][str[315]]] | None[316]

Get solver logs and debugging information.

Returns

Currently not implemented, returns None.

Return type

Optional[LogsType]

Gurobi Solver

Gurobi Solver Integration Module

This module provides Gurobi commercial solver integration for the OptiX mathematical optimization framework. It implements the Gurobi-specific solver interface that enables high-performance optimization for linear programming, goal programming, and constraint satisfaction problems using Gurobi's advanced optimization engine.

The module is organized around the following key components:

Architecture:

- **Solver Interface**: Gurobi-specific implementation of OptiX solver interface
- **Variable Translation**: Automatic conversion of OptiX variables to Gurobi format
- **Constraint Handling**: Support for all OptiX constraint types and operators
- **Solution Extraction**: Comprehensive solution status and value retrieval

Key Features:

- High-performance commercial optimization engine integration
- Support for binary, integer, and continuous variable types
- Advanced constraint handling including goal programming
- Configurable solver parameters and optimization settings
- Robust solution status detection and error handling

Solver Capabilities:

- **Linear Programming (LP)**: Standard optimization with linear constraints
- **Goal Programming (GP)**: Multi-objective optimization with deviation variables
- **Constraint Satisfaction (CSP)**: Feasibility problems without optimization
- **Mixed-Integer Programming**: Support for both continuous and integer variables

Usage:

The Gurobi solver is typically accessed through OptiX's unified solver factory:

```
from solvers.OXSolverFactory import solve
from problem import OXLPProblem

```

(continues on next page)

[304] https://docs.python.org/3/library/stdtypes.html#str
[305] https://docs.python.org/3/library/stdtypes.html#str
[306] https://docs.python.org/3/library/stdtypes.html#str
[307] https://docs.python.org/3/library/functions.html#int
[308] https://docs.python.org/3/library/functions.html#int
[309] https://docs.python.org/3/library/functions.html#int
[310] https://docs.python.org/3/library/functions.html#int
[311] https://docs.python.org/3/library/functions.html#int
[312] https://docs.python.org/3/library/typing.html#typing.List
[313] https://docs.python.org/3/library/stdtypes.html#str
[314] https://docs.python.org/3/library/typing.html#typing.List
[315] https://docs.python.org/3/library/stdtypes.html#str
[316] https://docs.python.org/3/library/constants.html#None

(continued from previous page)

```
# Create your optimization problem
problem = OXLPProblem()
# ... configure variables, constraints, objective ...

# Solve using Gurobi
status = solve(problem, 'Gurobi', use_continuous=True)
```

Requirements:

- Gurobi optimization software and valid license
- gurobipy Python package
- OptiX framework core components

Notes

- Gurobi requires a valid license for operation
- Performance characteristics may vary based on problem size and type
- Advanced Gurobi parameters can be configured through solver settings

`class solvers.gurobi.OXGurobiSolverInterface(**kwargs)`

Bases: `OXSolverInterface` (page 167)

Gurobi-specific implementation of the OptiX solver interface.

This class provides a concrete implementation of the OXSolverInterface for the Gurobi optimization solver. It handles the translation between OptiX's abstract problem representation and Gurobi's specific API calls, variable types, and constraint formats.

The interface supports both continuous and integer optimization modes, with automatic handling of variable bounds, constraint operators, and objective function setup. Special support is provided for goal programming with positive and negative deviation variables.

`_model`

The underlying Gurobi model instance

Type

gp.Model

`_var_mapping`

Maps OptiX variable IDs to Gurobi variable objects

Type

dict[317]

`_constraint_mapping`

Maps OptiX constraint IDs to Gurobi constraint objects

Type

dict[318]

`_constraint_expr_mapping`

Maps constraint IDs to their Gurobi expressions

Type
dict[319]

Parameters

- `use_continuous` (`bool`[320]) – Whether to use continuous variables instead of integers
- `equalizeDenominators` (`bool`[321]) – Whether to normalize fractional coefficients

Example

Direct usage of the Gurobi interface:

```
solver = OXGurobiSolverInterface(use_continuous=True)

# The solver is typically used through the factory pattern
# but can be used directly for advanced Gurobi-specific features

solver.create_variables(problem.variables)
solver.create_constraints(problem.constraints)
solver.create_objective(problem)

status = solver.solve(problem)
if status == OXSolutionStatus.OPTIMAL:
    solution = solver.get_solutions()[0]
```

`__init__`(***kwargs*)

Initialize the Gurobi solver interface with configuration parameters.

Creates a new Gurobi model instance and initializes internal mappings for variables, constraints, and constraint expressions. Configuration parameters are passed to the parent OXSolverInterface class.

Parameters
`**kwargs` – Configuration parameters including: use_continuous (bool): Use continuous variables instead of integers equalizeDenominators (bool): Normalize fractional coefficients

Note

The Gurobi model is created with the name "OptiX Model" and uses default Gurobi settings unless modified through solver parameters.

`create_objective`(*prb:* OXLPProblem *(page 88)*)

Create and configure the objective function in the Gurobi model.

Translates the OptiX objective function to Gurobi format, handling both minimization and maximization objectives. Supports continuous and integer coefficient modes with automatic goal programming objective creation.

Parameters
prb (OXLPProblem (page 88)) – Problem instance with objective function definition

Raises

- OXception – If no objective function is specified
- OXException – If float weights are used in integer mode without proper configuration

Note

- For goal programming problems, the objective is automatically created
- Fractional coefficients require equalizeDenominators parameter in integer mode
- Objective type (minimize/maximize) is preserved from the problem definition

`create_special_constraints`(*prb: OXCSPProblem (page 81)*)

Create special non-linear constraints for constraint satisfaction problems.

This method is intended for handling special constraints that cannot be expressed as standard linear constraints (e.g., multiplication, division, modulo, conditional constraints). Currently not implemented for Gurobi.

Parameters
prb (OXCSPProblem (page 81)) – Constraint satisfaction problem with special constraints

Note

Implementation is pending for advanced constraint types that require special handling in the Gurobi solver.

`get_solver_logs`() → List[322][str[323] | List[324][str[325]]] | None[326]

Retrieve solver execution logs and diagnostic information.

Returns detailed logs from the Gurobi solver execution including performance metrics, iteration details, and diagnostic messages. Currently not implemented.

Returns
Solver logs if available, None otherwise

Return type
Optional[LogsType]

Note

Implementation is pending for comprehensive log extraction from the Gurobi solver instance.

`solve`(*prb: OXCSPProblem (page 81)*) → OXSolutionStatus

Solve the optimization problem using Gurobi solver.

Executes the Gurobi optimization process and extracts solution information including variable values, constraint evaluations, and objective function value. Creates a comprehensive solution object for optimal solutions.

Parameters

`prb` (`OXCSPProblem` (page 81)) – Problem instance to solve

Returns

Status of the optimization process:

- OPTIMAL: Solution found successfully
- INFEASIBLE: No feasible solution exists
- UNBOUNDED: Problem is unbounded
- ERROR: Solver encountered an error or indeterminate status

Return type

OXSolutionStatus

Note

- Solution details are stored in the _solutions list for optimal solutions
- Constraint values include left-hand side, operator, and right-hand side
- Objective function value is included for linear programming problems

`class solvers.gurobi.OXGurobiSolverInterface`(***kwargs*)

Bases: `OXSolverInterface` (page 167)

Gurobi-specific implementation of the OptiX solver interface.

This class provides a concrete implementation of the OXSolverInterface for the Gurobi optimization solver. It handles the translation between OptiX's abstract problem representation and Gurobi's specific API calls, variable types, and constraint formats.

The interface supports both continuous and integer optimization modes, with automatic handling of variable bounds, constraint operators, and objective function setup. Special support is provided for goal programming with positive and negative deviation variables.

[317] https://docs.python.org/3/library/stdtypes.html#dict
[318] https://docs.python.org/3/library/stdtypes.html#dict
[319] https://docs.python.org/3/library/stdtypes.html#dict
[320] https://docs.python.org/3/library/functions.html#bool
[321] https://docs.python.org/3/library/functions.html#bool
[322] https://docs.python.org/3/library/typing.html#typing.List
[323] https://docs.python.org/3/library/stdtypes.html#str
[324] https://docs.python.org/3/library/typing.html#typing.List
[325] https://docs.python.org/3/library/stdtypes.html#str
[326] https://docs.python.org/3/library/constants.html#None

`_model`
The underlying Gurobi model instance

Type
gp.Model

`_var_mapping`
Maps OptiX variable IDs to Gurobi variable objects

Type
dict[327]

`_constraint_mapping`
Maps OptiX constraint IDs to Gurobi constraint objects

Type
dict[328]

`_constraint_expr_mapping`
Maps constraint IDs to their Gurobi expressions

Type
dict[329]

Parameters

- `use_continuous` (`bool`[330]) – Whether to use continuous variables instead of integers
- `equalizeDenominators` (`bool`[331]) – Whether to normalize fractional coefficients

Example

Direct usage of the Gurobi interface:

```
solver = OXGurobiSolverInterface(use_continuous=True)

# The solver is typically used through the factory pattern
# but can be used directly for advanced Gurobi-specific features

solver.create_variables(problem.variables)
solver.create_constraints(problem.constraints)
solver.create_objective(problem)

status = solver.solve(problem)
if status == OXSolutionStatus.OPTIMAL:
    solution = solver.get_solutions()[0]
```

`__init__`(***kwargs*)

Initialize the Gurobi solver interface with configuration parameters.

Creates a new Gurobi model instance and initializes internal mappings for variables, constraints, and constraint expressions. Configuration parameters are passed to the parent OXSolverInterface class.

Parameters
`**kwargs` – Configuration parameters including: use_continuous (bool): Use continuous variables instead of integers equalizeDenominators (bool): Normalize fractional coefficients

Note

The Gurobi model is created with the name "OptiX Model" and uses default Gurobi settings unless modified through solver parameters.

`create_special_constraints`(*prb:* OXCSPProblem *(page 81)*)

Create special non-linear constraints for constraint satisfaction problems.

This method is intended for handling special constraints that cannot be expressed as standard linear constraints (e.g., multiplication, division, modulo, conditional constraints). Currently not implemented for Gurobi.

Parameters
`prb` (`OXCSPProblem` (page 81)) – Constraint satisfaction problem with special constraints

Note

Implementation is pending for advanced constraint types that require special handling in the Gurobi solver.

`create_objective`(*prb:* OXLPProblem *(page 88)*)

Create and configure the objective function in the Gurobi model.

Translates the OptiX objective function to Gurobi format, handling both minimization and maximization objectives. Supports continuous and integer coefficient modes with automatic goal programming objective creation.

Parameters
`prb` (`OXLPProblem` (page 88)) – Problem instance with objective function definition

Raises

- `OXception` – If no objective function is specified
- `OXException` – If float weights are used in integer mode without proper configuration

Note

- For goal programming problems, the objective is automatically created
- Fractional coefficients require equalizeDenominators parameter in integer mode
- Objective type (minimize/maximize) is preserved from the problem definition

`solve`(*prb: OXCSPProblem (page 81)*) → OXSolutionStatus

Solve the optimization problem using Gurobi solver.

Executes the Gurobi optimization process and extracts solution information including variable values, constraint evaluations, and objective function value. Creates a comprehensive solution object for optimal solutions.

Parameters

`prb` (`OXCSPProblem` (page 81)) – Problem instance to solve

Returns

Status of the optimization process:

- OPTIMAL: Solution found successfully
- INFEASIBLE: No feasible solution exists
- UNBOUNDED: Problem is unbounded
- ERROR: Solver encountered an error or indeterminate status

Return type

OXSolutionStatus

Note

- Solution details are stored in the _solutions list for optimal solutions
- Constraint values include left-hand side, operator, and right-hand side
- Objective function value is included for linear programming problems

`get_solver_logs`() → List[332][str[333] | List[334][str[335]]] | None[336]

Retrieve solver execution logs and diagnostic information.

Returns detailed logs from the Gurobi solver execution including performance metrics, iteration details, and diagnostic messages. Currently not implemented.

Returns

Solver logs if available, None otherwise

Return type

Optional[LogsType]

Note

Implementation is pending for comprehensive log extraction from the Gurobi solver instance.

9.9.3 Solution Management

9.9.4 Examples

Basic Solving

```
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from solvers import solve

# Create problem
problem = OXLPProblem()
problem.create_decision_variable(”x”, ”Variable X”, 0, 10)
problem.create_decision_variable(”y”, ”Variable Y”, 0, 10)

# Add constraint
problem.create_constraint(
    variables=[var.id for var in problem.variables],
    weights=[1, 1],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=15
)

# Set objective
problem.create_objective_function(
    variables=[var.id for var in problem.variables],
    weights=[3, 2],
    objective_type=ObjectiveType.MAXIMIZE
)

# Solve with OR-Tools
status, solution = solve(problem, 'ORTools')
print(f”Status: {status}”)

if solution:
    for sol in solution:
        print(f”Objective value: {sol.objective_value}”)
        sol.print_solution_for(problem)

# Solve with Gurobi
try:
    status, solution = solve(problem, 'Gurobi')
```

(continues on next page)

[327] https://docs.python.org/3/library/stdtypes.html#dict
[328] https://docs.python.org/3/library/stdtypes.html#dict
[329] https://docs.python.org/3/library/stdtypes.html#dict
[330] https://docs.python.org/3/library/functions.html#bool
[331] https://docs.python.org/3/library/functions.html#bool
[332] https://docs.python.org/3/library/typing.html#typing.List
[333] https://docs.python.org/3/library/stdtypes.html#str
[334] https://docs.python.org/3/library/typing.html#typing.List
[335] https://docs.python.org/3/library/stdtypes.html#str
[336] https://docs.python.org/3/library/constants.html#None

(continued from previous page)

```
    print(f"Gurobi Status: {status}")
except Exception as e:
    print(f"Gurobi not available: {e}")
```

Solver Comparison

```
import time
from solvers import solve

def compare_solvers(problem, solvers=['ORTools', 'Gurobi']):
    """Compare performance of different solvers on the same problem."""
    results = {}

    for solver_name in solvers:
        try:
            start_time = time.time()
            status, solution = solve(problem, solver_name)
            solve_time = time.time() - start_time

            results[solver_name] = {
                'status': status,
                'solve_time': solve_time,
                'objective_value': solution[0].objective_value if solution else
 ↪None
            }

            print(f"{solver_name}:")
            print(f"  Status: {status}")
            print(f"  Time: {solve_time:.4f} seconds")
            if solution:
                print(f"  Objective: {solution[0].objective_value}")
            print()

        except Exception as e:
            print(f"{solver_name} failed: {e}")
            results[solver_name] = {'error': str(e)}

    return results

# Usage
results = compare_solvers(problem)
```

Custom Solver Implementation

```
from solvers import OXSolverInterface, OXSolverSolution, OXSolutionStatus
import random

class RandomSolverInterface(OXSolverInterface):
```

(continues on next page)

(continued from previous page)

```
    """A simple random solver for demonstration purposes."""

    def __init__(self):
        super().__init__()
        self.solutions = []

    def solve(self, problem):
        """Solve the problem using random sampling."""
        self.solutions = []

        # Simple random search (not optimal, just for demo)
        best_objective = float('-inf') if problem.objective_function.objective_
↪type == ObjectiveType.MAXIMIZE else float('inf')
        best_values = {}

        for _ in range(1000):  # 1000 random samples
            values = {}
            objective_value = 0

            # Generate random values for each variable
            for variable in problem.variables:
                random_value = random.uniform(variable.lower_bound, variable.
↪upper_bound)
                values[variable.id] = random_value

            # Check constraints (simplified)
            feasible = True
            for constraint in problem.constraints:
                constraint_value = sum(
                    constraint.weights[i] * values[constraint.variables[i]]
                    for i in range(len(constraint.variables))
                )

                if constraint.operator == RelationalOperators.LESS_THAN_EQUAL:
                    if constraint_value > constraint.value:
                        feasible = False
                        break
                elif constraint.operator == RelationalOperators.GREATER_THAN_
↪EQUAL:
                    if constraint_value < constraint.value:
                        feasible = False
                        break
                elif constraint.operator == RelationalOperators.EQUAL:
                    if abs(constraint_value - constraint.value) > 1e-6:
                        feasible = False
                        break

            if feasible:
                # Calculate objective value
```

(continues on next page)

(continued from previous page)

```
                objective_value = sum(
                    problem.objective_function.weights[i] * values[problem.
↪objective_function.variables[i]]
                    for i in range(len(problem.objective_function.variables))
                )

                # Check if this is the best solution so far
                is_better = False
                if problem.objective_function.objective_type == ObjectiveType.
↪MAXIMIZE:
                    is_better = objective_value > best_objective
                else:
                    is_better = objective_value < best_objective

                if is_better:
                    best_objective = objective_value
                    best_values = values.copy()

        # Create solution
        if best_values:
            solution = OXSolverSolution(
                objective_value=best_objective,
                variable_values=best_values,
                status=OXSolutionStatus.OPTIMAL
            )
            self.solutions = [solution]
            return OXSolutionStatus.OPTIMAL
        else:
            return OXSolutionStatus.INFEASIBLE

    def get_solution(self):
        """Return the best solution found."""
        return self.solutions

# Custom solvers would need to be registered through the solver factory
# This is an example of implementing a custom solver interface
```

Advanced Solver Configuration

```
from solvers.ortools import OXORToolsSolverInterface
from solvers.gurobi import OXGurobiSolverInterface

# Configure OR-Tools solver
ortools_solver = OXORToolsSolverInterface()
ortools_solver.set_time_limit(300)  # 5 minutes
ortools_solver.set_num_threads(4)

# Configure Gurobi solver (if available)
try:
```

(continues on next page)

(continued from previous page)

```
    gurobi_solver = OXGurobiSolverInterface()
    gurobi_solver.set_parameter('TimeLimit', 300)
    gurobi_solver.set_parameter('Threads', 4)
    gurobi_solver.set_parameter('MIPGap', 0.01)  # 1% optimality gap
except ImportError:
    print("Gurobi not available")

# Solve with configured solvers
status = ortools_solver.solve(problem)
ortools_solutions = ortools_solver.get_solution()
```

Parallel Solving

```
import concurrent.futures
import time

def solve_parallel(problem, solvers=['ORTools', 'Gurobi'], timeout=300):
    """Solve the same problem with multiple solvers in parallel."""

    def solve_with_solver(solver_name):
        try:
            start_time = time.time()
            status, solution = solve(problem, solver_name)
            solve_time = time.time() - start_time

            return {
                'solver': solver_name,
                'status': status,
                'solution': solution,
                'time': solve_time
            }
        except Exception as e:
            return {
                'solver': solver_name,
                'error': str(e),
                'time': None
            }

    # Use ThreadPoolExecutor for parallel execution
    with concurrent.futures.ThreadPoolExecutor(max_workers=len(solvers)) as
↪executor:
        # Submit all solver tasks
        future_to_solver = {
            executor.submit(solve_with_solver, solver): solver
            for solver in solvers
        }

        results = []
```

(continues on next page)

(continued from previous page)

```
        # Collect results as they complete
        for future in concurrent.futures.as_completed(future_to_solver,␣
↪timeout=timeout):
            try:
                result = future.result()
                results.append(result)
                print(f"Completed: {result['solver']} in {result.get('time', 'N/A
↪')} seconds")
            except Exception as e:
                solver = future_to_solver[future]
                results.append({
                    'solver': solver,
                    'error': str(e),
                    'time': None
                })

    return results

# Usage
parallel_results = solve_parallel(problem)

# Find the best result
best_result = None
for result in parallel_results:
    if 'error' not in result and result['solution']:
        if best_result is None or result['time'] < best_result['time']:
            best_result = result

if best_result:
    print(f"Best solver: {best_result['solver']} ({best_result['time']:.4f}s)")
```

Solution Analysis

```
def analyze_solution(solution, problem):
    """Analyze and validate a solution."""

    if not solution:
        print("No solution available")
        return

    sol = solution[0]  # Get first solution

    print("=== Solution Analysis ===")
    print(f"Objective Value: {sol.objective_value}")
    print(f"Status: {sol.status}")
    print()

    print("Variable Values:")
    for var_id, value in sol.variable_values.items():
```

(continues on next page)

(continued from previous page)

```
        variable = next((v for v in problem.variables if v.id == var_id), None)
        if variable:
            print(f”  {variable.name}: {value:.6f}”)
    print()

    # Validate constraints
    print(”Constraint Validation:”)
    all_satisfied = True

    for i, constraint in enumerate(problem.constraints):
        constraint_value = sum(
            constraint.weights[j] * sol.variable_values[constraint.variables[j]]
            for j in range(len(constraint.variables))
        )

        satisfied = False
        if constraint.operator == RelationalOperators.LESS_THAN_EQUAL:
            satisfied = constraint_value <= constraint.value + 1e-6
            op_str = ”<=”
        elif constraint.operator == RelationalOperators.GREATER_THAN_EQUAL:
            satisfied = constraint_value >= constraint.value - 1e-6
            op_str = ”>=”
        elif constraint.operator == RelationalOperators.EQUAL:
            satisfied = abs(constraint_value - constraint.value) <= 1e-6
            op_str = ”==”

        status_icon = ”▯ ” if satisfied else ”▯ ”
        print(f”  Constraint {i+1}: {constraint_value:.6f} {op_str} {constraint.
↪value} {status_icon}”)

        if not satisfied:
            all_satisfied = False

    print(f”\nAll constraints satisfied: {'▯ ' if all_satisfied else '▯ '}”)

    # Calculate objective value manually to verify
    if hasattr(problem, 'objective_function') and problem.objective_function:
        manual_objective = sum(
            problem.objective_function.weights[i] * sol.variable_values[problem.
↪objective_function.variables[i]]
            for i in range(len(problem.objective_function.variables))
        )
        print(f”Manual objective calculation: {manual_objective:.6f}”)
        print(f”Solver objective value: {sol.objective_value:.6f}”)
        print(f”Difference: {abs(manual_objective - sol.objective_value):.8f}”)

# Usage
status, solution = solve(problem, 'ORTools')
analyze_solution(solution, problem)
```

Multi-Scenario Solving

The `solve_all_scenarios` function enables comprehensive scenario-based optimization analysis:

```python
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from data import OXData
from solvers import solve_all_scenarios

# Create problem with scenario-based data
problem = OXLPProblem()

# Create decision variables
x = problem.create_decision_variable("production_x", "Production of X", 0, 100)
y = problem.create_decision_variable("production_y", "Production of Y", 0, 100)

# Create data object with scenarios
demand_data = OXData()
demand_data.demand = 100        # Default scenario
demand_data.price = 5.0

# Create scenarios for different market conditions
demand_data.create_scenario("High_Demand", demand=150, price=6.0)
demand_data.create_scenario("Low_Demand", demand=75, price=4.5)
demand_data.create_scenario("Peak_Season", demand=200, price=7.0)

# Add data to problem database
problem.db.add_object(demand_data)

# Create constraints using scenario data
problem.create_constraint(
    variables=[x.id, y.id],
    weights=[1, 1],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=demand_data.demand,
    description="Total production must not exceed demand"
)

# Create objective function using scenario data
problem.create_objective_function(
    variables=[x.id, y.id],
    weights=[demand_data.price, 3.0],
    objective_type=ObjectiveType.MAXIMIZE,
    description="Maximize revenue"
)

# Solve across all scenarios
scenario_results = solve_all_scenarios(problem, 'ORTools', maxTime=300)

print(f"Solved {len(scenario_results)} scenarios")
```

(continues on next page)

(continued from previous page)

```
print(f"Scenarios: {list(scenario_results.keys())}")

# Analyze results across scenarios
best_scenario = None
best_value = float('-inf')

for scenario_name, result in scenario_results.items():
    print(f"\n=== Scenario: {scenario_name} ===")

    if result['status'] == OXSolutionStatus.OPTIMAL:
        solution = result['solution']
        print(f"Status: Optimal")
        print(f"Objective Value: {solution.objective_value:.2f}")
        print(f"Production X: {solution.variable_values[x.id]:.2f}")
        print(f"Production Y: {solution.variable_values[y.id]:.2f}")

        # Track best scenario
        if solution.objective_value > best_value:
            best_value = solution.objective_value
            best_scenario = scenario_name
    else:
        print(f"Status: {result['status']}")

if best_scenario:
    print(f"\nBest performing scenario: {best_scenario} (${best_value:.2f})")
```

Advanced Multi-Scenario Analysis

```
from problem import OXLPProblem
from data import OXData
from solvers import solve_all_scenarios
import statistics

def comprehensive_scenario_analysis(problem, solver='ORTools'):
    """Perform comprehensive multi-scenario optimization analysis."""

    # Solve all scenarios
    results = solve_all_scenarios(problem, solver, maxTime=600)

    # Collect statistics
    optimal_scenarios = []
    objective_values = []

    for scenario_name, result in results.items():
        if result['status'] == OXSolutionStatus.OPTIMAL:
            optimal_scenarios.append(scenario_name)
            objective_values.append(result['solution'].objective_value)

    if not objective_values:
```

(continues on next page)

(continued from previous page)

```
        print(”No optimal solutions found across scenarios”)
        return

    # Statistical analysis
    print(”=== Multi-Scenario Analysis ===”)
    print(f”Total scenarios: {len(results)}”)
    print(f”Optimal scenarios: {len(optimal_scenarios)}”)
    print(f”Success rate: {len(optimal_scenarios)/len(results)*100:.1f}%”)
    print()

    print(”=== Objective Value Statistics ===”)
    print(f”Best value: {max(objective_values):.2f}”)
    print(f”Worst value: {min(objective_values):.2f}”)
    print(f”Average value: {statistics.mean(objective_values):.2f}”)
    print(f”Median value: {statistics.median(objective_values):.2f}”)
    print(f”Standard deviation: {statistics.stdev(objective_values):.2f}”)
    print()

    # Scenario ranking
    scenario_ranking = []
    for scenario_name, result in results.items():
        if result['status'] == OXSolutionStatus.OPTIMAL:
            scenario_ranking.append((scenario_name, result['solution'].objective_
↪value))

    scenario_ranking.sort(key=lambda x: x[1], reverse=True)

    print(”=== Scenario Ranking ===”)
    for i, (scenario, value) in enumerate(scenario_ranking, 1):
        print(f”{i:2d}. {scenario:<20}: ${value:8.2f}”)

    # Sensitivity analysis
    if len(objective_values) > 1:
        value_range = max(objective_values) - min(objective_values)
        cv = statistics.stdev(objective_values) / statistics.mean(objective_
↪values)

        print(f”\n=== Sensitivity Analysis ===”)
        print(f”Value range: ${value_range:.2f}”)
        print(f”Coefficient of variation: {cv:.3f}”)

        if cv > 0.2:
            print(”□   High sensitivity to scenario parameters”)
        elif cv > 0.1:
            print(”□  Moderate sensitivity to scenario parameters”)
        else:
            print(”□  Low sensitivity to scenario parameters”)

    return results
```

(continues on next page)

(continued from previous page)

```
# Usage with complex multi-object scenarios
problem = OXLPProblem()

# Create variables
x = problem.create_decision_variable("x", "Variable X", 0, 50)
y = problem.create_decision_variable("y", "Variable Y", 0, 50)

# Create multiple data objects with coordinated scenarios
capacity_data = OXData()
capacity_data.max_capacity = 100
capacity_data.create_scenario("Expansion", max_capacity=150)
capacity_data.create_scenario("Recession", max_capacity=80)
capacity_data.create_scenario("Growth", max_capacity=120)

cost_data = OXData()
cost_data.unit_cost = 2.0
cost_data.create_scenario("Expansion", unit_cost=1.8)    # Lower costs during
↪expansion
cost_data.create_scenario("Recession", unit_cost=2.5)   # Higher costs during
↪recession
cost_data.create_scenario("Growth", unit_cost=2.2)      # Moderate cost increase

# Add to database
problem.db.add_object(capacity_data)
problem.db.add_object(cost_data)

# Create constraints and objectives using scenario data
problem.create_constraint([x.id, y.id], [1, 1], "<=", capacity_data.max_capacity)
problem.create_objective_function([x.id, y.id], [cost_data.unit_cost, 3.0],
↪"maximize")

# Perform comprehensive analysis
analysis_results = comprehensive_scenario_analysis(problem, 'Gurobi')
```

Constraint-Based Scenarios

```
from constraints import RelationalOperators
from solvers import solve_all_scenarios

# Create problem with constraint scenarios
problem = OXLPProblem()

x = problem.create_decision_variable("x", "Production X", 0, 100)
y = problem.create_decision_variable("y", "Production Y", 0, 100)

# Create base constraint
resource_constraint = problem.create_constraint(
    variables=[x.id, y.id],
```

(continues on next page)

(continued from previous page)

```
    weights=[2, 1],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=200,
    description="Resource availability constraint"
)

# Add constraint scenarios for different resource conditions
resource_constraint.create_scenario(
    "Limited_Resources",
    rhs=150,
    description="Resource shortage scenario"
)

resource_constraint.create_scenario(
    "Abundant_Resources",
    rhs=300,
    description="Resource abundance scenario"
)

resource_constraint.create_scenario(
    "Emergency_Resources",
    rhs=100,
    description="Emergency resource rationing"
)

# Create objective
problem.create_objective_function(
    variables=[x.id, y.id],
    weights=[5, 4],
    objective_type=ObjectiveType.MAXIMIZE
)

# Solve across constraint scenarios
constraint_results = solve_all_scenarios(problem, 'ORTools')

# Analyze impact of resource availability
print("=== Resource Scenario Analysis ===")
for scenario_name, result in constraint_results.items():
    if result['status'] == OXSolutionStatus.OPTIMAL:
        solution = result['solution']
        total_production = solution.variable_values[x.id] + solution.variable_
↪values[y.id]

        print(f"{scenario_name}:")
        print(f"  Objective: ${solution.objective_value:.2f}")
        print(f"  Total Production: {total_production:.2f} units")
        print(f"  X Production: {solution.variable_values[x.id]:.2f}")
        print(f"  Y Production: {solution.variable_values[y.id]:.2f}")
        print()
```

Mixed Data and Constraint Scenarios

```python
from data import OXData
from solvers import solve_all_scenarios

# Complex scenario setup with both data and constraint scenarios
problem = OXLPProblem()

# Variables
x = problem.create_decision_variable("x", "Product X", 0, 100)
y = problem.create_decision_variable("y", "Product Y", 0, 100)

# Data object scenarios for market conditions
market_data = OXData()
market_data.price_x = 10.0
market_data.price_y = 8.0
market_data.create_scenario("Bull_Market", price_x=12.0, price_y=10.0)
market_data.create_scenario("Bear_Market", price_x=8.0, price_y=6.0)

problem.db.add_object(market_data)

# Constraint scenarios for operational conditions
capacity_constraint = problem.create_constraint(
    variables=[x.id, y.id],
    weights=[1, 1],
    operator=RelationalOperators.LESS_THAN_EQUAL,
    value=150,
    description="Production capacity"
)
capacity_constraint.create_scenario("Maintenance", rhs=100)
capacity_constraint.create_scenario("Overtime", rhs=200)

# Objective using data scenarios
problem.create_objective_function(
    variables=[x.id, y.id],
    weights=[market_data.price_x, market_data.price_y],
    objective_type=ObjectiveType.MAXIMIZE
)

# This will solve all combinations:
# - Default + Default, Bull_Market + Default, Bear_Market + Default
# - Default + Maintenance, Bull_Market + Maintenance, Bear_Market + Maintenance
# - Default + Overtime, Bull_Market + Overtime, Bear_Market + Overtime
mixed_results = solve_all_scenarios(problem, 'Gurobi', use_continuous=True)

print(f"Total scenario combinations solved: {len(mixed_results)}")

# Group results by data vs constraint scenarios
market_scenarios = {}
capacity_scenarios = {}
```

(continues on next page)

(continued from previous page)

```python
for scenario_name, result in mixed_results.items():
    if result['status'] == OXSolutionStatus.OPTIMAL:
        solution = result['solution']

        # Categorize scenarios
        if 'Market' in scenario_name:
            market_scenarios[scenario_name] = solution.objective_value
        elif scenario_name in ['Maintenance', 'Overtime']:
            capacity_scenarios[scenario_name] = solution.objective_value
        else:
            print(f"Default scenario value: ${solution.objective_value:.2f}")

print("\n=== Market Impact Analysis ===")
for scenario, value in market_scenarios.items():
    print(f"{scenario}: ${value:.2f}")

print("\n=== Capacity Impact Analysis ===")
for scenario, value in capacity_scenarios.items():
    print(f"{scenario}: ${value:.2f}")
```

9.9.5 See Also

- *Problem Module* (page 81) - Problem type definitions
- ../tutorials/custom_solvers - Creating custom solver implementations
- ../user_guide/solvers - Detailed solver configuration guide

9.10 Data Module

The data module provides scenario-based data management capabilities for optimization problems. It includes data objects with multi-scenario support and type-safe database collections for organizing data.

9.10.1 Data Classes

Data Objects

`class data.OXData`(*id: ~uuid.UUID = <factory>*, *class_name: str = ''*, *active_scenario: str = 'Default'*, *scenarios: dict[str, dict[str, ~typing.Any]] = <factory>*)

Bases: `OXObject`

A base class for data objects with scenario support.

This class provides a mechanism for storing different attribute values for different scenarios. When an attribute is accessed, the system first checks if it exists in the active scenario, and if not, falls back to the object's own attribute.

`active_scenario`

The name of the currently active scenario. Defaults to "Default".

Type

str[337]

`scenarios`

A dictionary mapping scenario names to dictionaries of attribute values.

Type

dict[338][str[339], dict[340][str[341], Any]]

Examples

```
>>> data = OXData()
>>> data.value = 10
>>> data.create_scenario("Optimistic", value=20)
>>> data.create_scenario("Pessimistic", value=5)
>>> print(data.value)  # Default scenario
10
>>> data.active_scenario = "Optimistic"
>>> print(data.value)  # Optimistic scenario
20
>>> data.active_scenario = "Pessimistic"
>>> print(data.value)  # Pessimistic scenario
5
```

`active_scenario: str[342] = 'Default'`

`scenarios: dict[343][str[344], dict[345][str[346], Any[347]]]`

`__getattribute__`(*item*)

Custom attribute access that checks the active scenario first.

When an attribute is accessed, this method first checks if it exists in the active scenario, and if not, falls back to the object's own attribute.

Parameters

`item` (str[348]) – The name of the attribute to access.

Returns

The value of the attribute in the active scenario, or the object's own attribute if not found in the active scenario.

Return type

Any

`create_scenario`(*scenario_name: str[349]*, ***kwargs*)

Create a new scenario with the specified attribute values.

If the "Default" scenario doesn't exist yet, it is created first, capturing the object's current attribute values.

Parameters

- `scenario_name` (str[350]) – The name of the new scenario.

- `**kwargs` – Attribute-value pairs for the new scenario.

Raises

`OXception` – If an attribute in kwargs doesn't exist in the object.

Examples

```
>>> data = OXData()
>>> data.value = 10
>>> data.create_scenario("Optimistic", value=20)
>>> data.active_scenario = "Optimistic"
>>> print(data.value)
20
```

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', active_scenario: str = 'Default', scenarios: dict[str, dict[str, ~typing.Any]] = <factory>*) → None[351]

Database Collections

`class data.OXDatabase`(*id: ~uuid.UUID = <factory>, class_name: str = '', objects: list[~base.OXObject.OXObject] = <factory>*)

Bases: `OXObjectPot`

A container for OXData objects.

This class extends OXObjectPot to provide a container specifically for OXData objects. It enforces type safety by ensuring that only OXData objects can be added to or removed from the database.

Examples

```
>>> db = OXDatabase()
>>> data1 = OXData()
>>> data2 = OXData()
>>> db.add_object(data1)
>>> db.add_object(data2)
>>> len(db)
2
>>> for data in db:
```

(continues on next page)

337 https://docs.python.org/3/library/stdtypes.html#str
338 https://docs.python.org/3/library/stdtypes.html#dict
339 https://docs.python.org/3/library/stdtypes.html#str
340 https://docs.python.org/3/library/stdtypes.html#dict
341 https://docs.python.org/3/library/stdtypes.html#str
342 https://docs.python.org/3/library/stdtypes.html#str
343 https://docs.python.org/3/library/stdtypes.html#dict
344 https://docs.python.org/3/library/stdtypes.html#str
345 https://docs.python.org/3/library/stdtypes.html#dict
346 https://docs.python.org/3/library/stdtypes.html#str
347 https://docs.python.org/3/library/typing.html#typing.Any
348 https://docs.python.org/3/library/stdtypes.html#str
349 https://docs.python.org/3/library/stdtypes.html#str
350 https://docs.python.org/3/library/stdtypes.html#str
351 https://docs.python.org/3/library/constants.html#None

(continued from previous page)

```
...     print(data.id)
12345678-1234-5678-1234-567812345678
87654321-4321-8765-4321-876543210987
```

See also

`base.OXObjectPot.OXObjectPot` `data.OXData.OXData` (page 205)

`add_object`(*obj: OXObject*)

Add an OXData object to the database.

Parameters

`obj` (`OXObject`) – The object to add. Must be an instance of OXData.

Raises

`OXception` – If the object is not an instance of OXData.

`remove_object`(*obj: OXObject*)

Remove an OXData object from the database.

Parameters

`obj` (`OXObject`) – The object to remove. Must be an instance of OXData.

Raises

- `OXception` – If the object is not an instance of OXData.
- `ValueError`[352] – If the object is not in the database.

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', objects: list[~base.OXObject.OXObject] = <factory>*) → None[353]

9.10.2 Constants

`data.NON_SCENARIO_FIELDS = ['active_scenario', 'scenarios', 'id', 'class_name']`

Built-in mutable sequence.

If no argument is given, the constructor creates a new empty list. The argument must be an iterable if specified.

List of field names that are excluded from scenario management to prevent infinite loops and maintain object integrity. These fields are always accessed from the base object.

9.10.3 Examples

Basic Data Objects with Scenarios

```
from data import OXData

# Create a data object with base values
```

(continues on next page)

[352] https://docs.python.org/3/library/exceptions.html#ValueError
[353] https://docs.python.org/3/library/constants.html#None

(continued from previous page)

```
demand_data = OXData()
demand_data.quantity = 100
demand_data.cost = 50.0

# Create scenarios for sensitivity analysis
demand_data.create_scenario("High_Demand", quantity=150, cost=55.0)
demand_data.create_scenario("Low_Demand", quantity=75, cost=45.0)

# Access values in different scenarios
print(demand_data.quantity)  # 100 (Default scenario)

demand_data.active_scenario = "High_Demand"
print(demand_data.quantity)  # 150

demand_data.active_scenario = "Low_Demand"
print(demand_data.quantity)  # 75
```

Database Collections

```
from data import OXDatabase, OXData

# Create a database for organizing data objects
db = OXDatabase()

# Create multiple data objects
factory_a = OXData()
factory_a.location = "Factory_A"
factory_a.capacity = 500
factory_a.create_scenario("Expansion", capacity=750)

factory_b = OXData()
factory_b.location = "Factory_B"
factory_b.capacity = 300
factory_b.create_scenario("Expansion", capacity=450)

# Add objects to database
db.add_object(factory_a)
db.add_object(factory_b)

print(f"Total factories: {len(db)}")

# Iterate through all objects
for factory in db:
    print(f"Factory at {factory.location}: capacity {factory.capacity}")

# Switch to expansion scenario
factory_a.active_scenario = "Expansion"
factory_b.active_scenario = "Expansion"
```

(continues on next page)

(continued from previous page)

```
# Check expanded capacities
for factory in db:
    print(f"Expanded {factory.location}: capacity {factory.capacity}")
```

Scenario Management for Optimization

```
from data import OXData

# Create demand data with multiple scenarios
demand = OXData()
demand.product = "Widget_A"
demand.base_demand = 1000
demand.seasonal_factor = 1.0

# Create scenarios for different market conditions
demand.create_scenario(
    "Optimistic",
    base_demand=1200,
    seasonal_factor=1.2
)

demand.create_scenario(
    "Pessimistic",
    base_demand=800,
    seasonal_factor=0.8
)

demand.create_scenario(
    "Realistic",
    base_demand=1000,
    seasonal_factor=1.0
)

# Function to calculate total demand
def calculate_total_demand(data, season_multiplier=1.0):
    return data.base_demand * data.seasonal_factor * season_multiplier

# Compare scenarios
scenarios = ["Default", "Optimistic", "Pessimistic", "Realistic"]

for scenario in scenarios:
    demand.active_scenario = scenario
    total = calculate_total_demand(demand)
    print(f"{scenario} scenario: {total} units")
```

Scenario-Based Sensitivity Analysis

```
def run_sensitivity_analysis(data_objects, scenarios, optimization_function):
    """Run optimization across multiple scenarios."""

    results = {}

    for scenario_name in scenarios:
        print(f"Running scenario: {scenario_name}")

        # Switch all data objects to the same scenario
        for data_obj in data_objects:
            if scenario_name in data_obj.scenarios:
                data_obj.active_scenario = scenario_name
            else:
                data_obj.active_scenario = "Default"

        # Run optimization with current scenario data
        result = optimization_function(data_objects)
        results[scenario_name] = result

    return results

# Usage example
def simple_profit_calculation(data_objects):
    total_profit = 0
    for obj in data_objects:
        if hasattr(obj, 'revenue') and hasattr(obj, 'cost'):
            total_profit += obj.revenue - obj.cost
    return total_profit

# Create test data
product1 = OXData()
product1.revenue = 100
product1.cost = 60
product1.create_scenario("HighPrice", revenue=120, cost=60)
product1.create_scenario("LowCost", revenue=100, cost=45)

product2 = OXData()
product2.revenue = 80
product2.cost = 50
product2.create_scenario("HighPrice", revenue=95, cost=50)
product2.create_scenario("LowCost", revenue=80, cost=40)

products = [product1, product2]
scenarios = ["Default", "HighPrice", "LowCost"]

results = run_sensitivity_analysis(products, scenarios, simple_profit_
↪calculation)

for scenario, profit in results.items():
```

(continues on next page)

(continued from previous page)

```
    print(f"{scenario}: ${profit} profit")
```

Working with Multiple Data Types

```
from data import OXData, OXDatabase

def create_supply_chain_data():
    """Create a multi-type supply chain dataset."""

    # Create suppliers with different scenarios
    supplier_db = OXDatabase()

    supplier_a = OXData()
    supplier_a.name = "Supplier_A"
    supplier_a.lead_time = 14
    supplier_a.reliability = 0.95
    supplier_a.cost_factor = 1.0
    supplier_a.create_scenario("Crisis", lead_time=21, reliability=0.85, cost_
↪factor=1.3)

    supplier_b = OXData()
    supplier_b.name = "Supplier_B"
    supplier_b.lead_time = 7
    supplier_b.reliability = 0.98
    supplier_b.cost_factor = 1.1
    supplier_b.create_scenario("Crisis", lead_time=10, reliability=0.90, cost_
↪factor=1.4)

    supplier_db.add_object(supplier_a)
    supplier_db.add_object(supplier_b)

    # Create demand points with scenarios
    demand_db = OXDatabase()

    region_1 = OXData()
    region_1.name = "Region_1"
    region_1.demand = 1000
    region_1.max_price = 50
    region_1.create_scenario("Growth", demand=1300, max_price=55)
    region_1.create_scenario("Recession", demand=700, max_price=45)

    region_2 = OXData()
    region_2.name = "Region_2"
    region_2.demand = 800
    region_2.max_price = 48
    region_2.create_scenario("Growth", demand=1100, max_price=52)
    region_2.create_scenario("Recession", demand=600, max_price=42)

    demand_db.add_object(region_1)
```

(continues on next page)

(continued from previous page)

```
    demand_db.add_object(region_2)

    return supplier_db, demand_db

# Create the data
suppliers, demand_points = create_supply_chain_data()

# Analyze different scenarios
scenarios = ["Default", "Growth", "Crisis", "Recession"]

for scenario in scenarios:
    print(f"\n=== {scenario} Scenario ===")

    # Switch all objects to the scenario
    for supplier in suppliers:
        if scenario in supplier.scenarios:
            supplier.active_scenario = scenario
        else:
            supplier.active_scenario = "Default"

    for demand in demand_points:
        if scenario in demand.scenarios:
            demand.active_scenario = scenario
        else:
            demand.active_scenario = "Default"

    # Calculate scenario metrics
    total_demand = sum(d.demand for d in demand_points)
    avg_lead_time = sum(s.lead_time for s in suppliers) / len(suppliers)
    avg_reliability = sum(s.reliability for s in suppliers) / len(suppliers)

    print(f"Total demand: {total_demand}")
    print(f"Avg lead time: {avg_lead_time:.1f} days")
    print(f"Avg reliability: {avg_reliability:.2%}")
```

UUID-Based Object Access

```
from data import OXDatabase, OXData

# Create database with data objects
db = OXDatabase()

# Add several objects
for i in range(5):
    obj = OXData()
    obj.value = i * 10
    obj.category = f"Type_{i % 3}"
    db.add_object(obj)
```

(continues on next page)

(continued from previous page)

```
print(f"Database contains {len(db)} objects")

# Access objects by UUID (inherited from OXObjectPot)
first_obj = list(db)[0]
found_obj = db.get_object_by_id(first_obj.id)
print(f"Found object with value: {found_obj.value}")

# Manual filtering using iteration
type_0_objects = [obj for obj in db if obj.category == "Type_0"]
print(f"Found {len(type_0_objects)} objects of Type_0")

# Remove objects
db.remove_object(first_obj)
print(f"After removal: {len(db)} objects")
```

Advanced Scenario Patterns

```
def create_monte_carlo_scenarios(base_data, num_scenarios=100):
    """Create multiple scenarios for Monte Carlo analysis."""
    import random

    for i in range(num_scenarios):
        # Create variations around base values
        demand_variation = random.uniform(0.8, 1.2)
        cost_variation = random.uniform(0.9, 1.1)

        scenario_name = f"MonteCarlo_{i+1:03d}"
        base_data.create_scenario(
            scenario_name,
            demand=int(base_data.demand * demand_variation),
            cost=base_data.cost * cost_variation
        )

# Create base data
product = OXData()
product.demand = 1000
product.cost = 50.0
product.revenue = 75.0

# Generate Monte Carlo scenarios
create_monte_carlo_scenarios(product, 10)  # Create 10 scenarios

# Run analysis across all scenarios
profits = []
for scenario_name in product.scenarios:
    product.active_scenario = scenario_name
    profit = product.revenue - product.cost
    profits.append((scenario_name, profit))
```

(continues on next page)

(continued from previous page)

```python
# Show results
for scenario, profit in sorted(profits, key=lambda x: x[1], reverse=True)[:5]:
    print(f"{scenario}: ${profit:.2f} profit")
```

Type Safety and Error Handling

```python
from data import OXDatabase, OXData
from base import OXObject, OXception

# Demonstrate type safety
db = OXDatabase()
data_obj = OXData()

# This works - OXData is allowed
db.add_object(data_obj)
print("OXData object added successfully")

# This will fail - only OXData objects allowed
try:
    non_data_obj = OXObject()  # Base object, not OXData
    db.add_object(non_data_obj)
except OXception as e:
    print(f"Type safety error: {e}")

# Scenario error handling
try:
    data_obj.create_scenario("TestScenario", nonexistent_attr="value")
except OXception as e:
    print(f"Scenario creation error: {e}")
```

9.10.4 See Also

- base - Base classes that data objects inherit from
- *Problem Module* (page 81) - Problem classes that use data objects
- *Variables Module* (page 134) - Variable creation that can be linked to data objects
- ../user_guide/scenarios - Advanced scenario modeling guide

9.11 Analysis Module

The analysis module provides comprehensive analysis tools for OptiX optimization problems, including sensitivity analysis, scenario comparison, and performance evaluation capabilities that leverage the built-in scenario management system.

9.11.1 Analysis Classes

Objective Function Analysis

`class analysis.OXObjectiveFunctionAnalysis(`*problem: OXLPProblem (page 88) | OXGPProblem (page 92), solver: str*[354]*, **kwargs*`)`

Bases: `object`[355]

Comprehensive objective function analysis tool for multi-scenario optimization problems.

This class provides systematic analysis of objective function behavior across different scenarios in OptiX optimization problems. It leverages the built-in scenario management system to automatically solve problems under various parameter configurations and provides detailed statistical analysis and comparative insights.

The analyzer is designed to work seamlessly with linear programming (LP) and goal programming (GP) problems that have objective functions, automatically handling scenario discovery, problem solving, and result aggregation to deliver comprehensive objective function sensitivity analysis.

Key Capabilities:

- **Automatic Scenario Discovery**: Scans problem database to identify all available scenarios across data objects for comprehensive analysis coverage
- **Multi-Scenario Solving**: Systematically solves the optimization problem under each scenario configuration using the specified solver
- **Statistical Analysis**: Computes comprehensive statistics including central tendency, variability, and distribution metrics for objective function values
- **Performance Ranking**: Identifies best and worst performing scenarios based on optimization direction (maximization or minimization)
- **Success Rate Analysis**: Tracks solver success rates across scenarios to identify problematic parameter configurations
- **Comparative Insights**: Provides structured comparison framework for evaluating parameter sensitivity and scenario impact on optimization outcomes

`problem`

The optimization problem instance to analyze. Must have an objective function and scenario-enabled data objects.

Type

Union[*OXLPProblem* (page 88), *OXGPProblem* (page 92)]

`solver`

Identifier of the solver to use for all scenario solving operations. Must be available in the OptiX solver registry.

Type

str[356]

`solver_kwargs`

Additional parameters passed to the solver for each scenario solving operation. Enables custom solver configuration and performance tuning.

Type
Dict[str[357], Any]

Examples

Basic objective function analysis:

```
from analysis.OXObjectiveFunctionAnalysis import OXObjectiveFunctionAnalysis

# Create analyzer
analyzer = OXObjectiveFunctionAnalysis(problem, 'ORTools')

# Perform analysis
results = analyzer.analyze()

# Access comprehensive results
print(f"Analyzed {results.total_scenario_count} scenarios")
print(f"Success rate: {results.success_rate:.1%}")
print(f"Best scenario: {results.best_scenario}")
print(f"Objective value range: {results.statistics['min']:.2f} - {results.
↪statistics['max']:.2f}")
```

Advanced analysis with custom solver parameters:

```
# Create analyzer with custom solver settings
analyzer = OXObjectiveFunctionAnalysis(
    problem,
    'Gurobi',
    maxTime=300,
    use_continuous=True
)

# Perform analysis
results = analyzer.analyze()

# Detailed scenario ranking
ranking = results.get_scenario_ranking()
print("Scenario Performance Ranking:")
for rank, (scenario, value) in enumerate(ranking, 1):
    print(f"{rank:2d}. {scenario:20s}: {value:10.2f}")

# Statistical insights
stats = results.statistics
print(f"\nStatistical Summary:")
print(f"Mean: {stats['mean']:.2f} ± {stats['std_dev']:.2f}")
print(f"Range: [{stats['min']:.2f}, {stats['max']:.2f}]")
print(f"Coefficient of Variation: {stats['std_dev']/stats['mean']:.3f}")
```

`__init__`(*problem: OXLPProblem (page 88) | OXGPProblem (page 92), solver: str*[358], ***kwargs*)

Initialize the objective function analyzer.

Parameters

- `problem` (`Union[OXLPProblem` (page 88), `OXGPProblem` (page 92)`]`) – The optimization problem to analyze. Must have an objective function and scenario-enabled data in the database.
- `solver` (`str`[359]) – The solver identifier to use for scenario solving. Must be available in the OptiX solver registry.
- `**kwargs` – Additional keyword arguments passed to the solver for each scenario solving operation. Enables custom solver configuration.

Raises

`OXception` – If the problem doesn't have an objective function or if the problem database is empty.

Examples

```
>>> analyzer = OXObjectiveFunctionAnalysis(lp_problem, 'ORTools')
>>> analyzer = OXObjectiveFunctionAnalysis(gp_problem, 'Gurobi',
↪maxTime=600)
```

`analyze()` → *OXObjectiveFunctionAnalysisResult* (page 219)

Perform comprehensive objective function analysis across all scenarios.

This method orchestrates the complete analysis workflow including scenario discovery, multi-scenario solving, statistical computation, and result aggregation to provide comprehensive objective function insights.

Analysis Workflow:

1. **Scenario Solving**: Uses solve_all_scenarios to solve the problem under each scenario configuration with the specified solver
2. **Data Extraction**: Extracts objective function values from optimal solutions and tracks solution status for each scenario
3. **Statistical Analysis**: Computes comprehensive statistics including central tendency, variability, and distribution metrics
4. **Performance Ranking**: Identifies best and worst scenarios based on optimization direction (maximization or minimization)
5. **Result Aggregation**: Organizes all analysis results into a structured OXObjectiveFunctionAnalysisResult for easy access

Returns

Comprehensive analysis results containing
scenario values, statistical metrics, performance rankings, and success rates.

Return type

OXObjectiveFunctionAnalysisResult (page 219)

Raises

`OXception` – If no scenarios are found or if all scenarios fail to solve.

Examples

```
>>> analyzer = OXObjectiveFunctionAnalysis(problem, 'ORTools')
>>> results = analyzer.analyze()
>>> print(f"Best scenario: {results.best_scenario} = {results.scenario_
↪values[results.best_scenario]:.2f}")
```

`compare_scenarios(`*scenario_names: List*[360]*[str*[361]*]*`)` → Dict[362][str[363], Dict[364][str[365], Any[366]]]

Compare specific scenarios in detail.

This method provides detailed comparison of specified scenarios including objective function values, solution status, and relative performance metrics for focused analysis of particular parameter configurations.

Parameters

`scenario_names` (`List[str`[367]`]`) – List of scenario names to compare. Must be valid scenario names from the problem database.

Returns

Detailed comparison results for each scenario
including objective values, status, and rankings.

Return type

Dict[str[368], Dict[str[369], Any]]

Raises

`OXception` – If any specified scenario name is not found in the analysis results.

Examples

```
>>> analyzer = OXObjectiveFunctionAnalysis(problem, 'ORTools')
>>> results = analyzer.analyze()
>>> comparison = analyzer.compare_scenarios(['High_Demand', 'Low_Demand
↪'])
>>> for scenario, details in comparison.items():
...     print(f"{scenario}: {details['objective_value']:.2f} ({details[
↪'status']})")
```

354 https://docs.python.org/3/library/stdtypes.html#str
355 https://docs.python.org/3/library/functions.html#object
356 https://docs.python.org/3/library/stdtypes.html#str
357 https://docs.python.org/3/library/stdtypes.html#str
358 https://docs.python.org/3/library/stdtypes.html#str
359 https://docs.python.org/3/library/stdtypes.html#str
360 https://docs.python.org/3/library/typing.html#typing.List
361 https://docs.python.org/3/library/stdtypes.html#str
362 https://docs.python.org/3/library/typing.html#typing.Dict
363 https://docs.python.org/3/library/stdtypes.html#str
364 https://docs.python.org/3/library/typing.html#typing.Dict
365 https://docs.python.org/3/library/stdtypes.html#str
366 https://docs.python.org/3/library/typing.html#typing.Any
367 https://docs.python.org/3/library/stdtypes.html#str
368 https://docs.python.org/3/library/stdtypes.html#str
369 https://docs.python.org/3/library/stdtypes.html#str

`class analysis.OXObjectiveFunctionAnalysisResult(`*id: ~uuid.UUID = <factory>, class_name: str = '', scenario_values: ~typing.Dict[str, float] = <factory>, scenario_statuses: ~typing.Dict[str, ~solvers.OXSolverInterface.OXSolutionStatus] = <factory>, statistics: ~typing.Dict[str, float] = <factory>, best_scenario: str | None = None, worst_scenario: str | None = None, optimal_scenario_count: int = 0, total_scenario_count: int = 0, success_rate: float = 0.0, objective_direction: str = 'maximize'*`)`

Bases: `OXObject`

Comprehensive data structure containing objective function analysis results.

This class encapsulates all analysis results from multi-scenario objective function evaluation, providing structured access to statistical metrics, scenario comparisons, and performance insights for systematic analysis and reporting.

The result structure is designed to support both programmatic analysis and human-readable reporting, with detailed metadata and comprehensive statistical information for thorough objective function sensitivity analysis.

`scenario_values`

Dictionary mapping scenario names to their corresponding optimal objective function values. Only includes scenarios that achieved optimal solutions for accurate statistical analysis.

Type

Dict[str[370], float[371]]

`scenario_statuses`

Dictionary mapping scenario names to their solution termination status. Enables identification of scenarios that failed to solve optimally.

Type

Dict[str[372], OXSolutionStatus]

`statistics`

Comprehensive statistical analysis of objective function values across all optimal scenarios including: - mean: Average objective function value - median: Middle value when scenarios are sorted - std_dev: Standard deviation measuring variability - variance: Statistical variance of objective values - min: Minimum objective function value observed - max: Maximum objective function value observed - range: Difference between maximum and minimum values

Type

Dict[str[373], float[374]]

`best_scenario`

Name of the scenario that achieved the best (highest for maximization, lowest for minimization) objective function value. None if no optimal solutions.

Type
Optional[str[375]]

`worst_scenario`
Name of the scenario that achieved the worst (lowest for maximization, highest for minimization) objective function value. None if no optimal solutions.

Type
Optional[str[376]]

`optimal_scenario_count`
Number of scenarios that achieved optimal solutions. Important metric for understanding solution reliability across different parameter configurations.

Type
int[377]

`total_scenario_count`
Total number of scenarios analyzed, including those that failed to solve optimally. Used for calculating success rates and identifying problematic scenarios.

Type
int[378]

`success_rate`
Percentage of scenarios that achieved optimal solutions. Calculated as (optimal_scenario_count / total_scenario_count). High success rates indicate robust problem formulation.

Type
float[379]

`objective_direction`
Direction of optimization ("maximize" or "minimize") used to correctly identify best and worst scenarios. Automatically determined from problem configuration.

Type
str[380]

Examples

```
>>> result = OXObjectiveFunctionAnalysisResult()
>>> print(f"Success rate: {result.success_rate:.1%}")
>>> print(f"Best scenario: {result.best_scenario} = {result.scenario_
↪values[result.best_scenario]}")
>>> print(f"Statistical summary: mean={result.statistics['mean']:.2f}, std=
↪{result.statistics['std_dev']:.2f}")
```

`scenario_values: Dict[381][str[382], float[383]]`

`scenario_statuses: Dict[384][str[385], OXSolutionStatus]`

`statistics: Dict[386][str[387], float[388]]`

`best_scenario: str[389] | None[390] = None` `worst_scenario: str`[391] `| None`[392] `= None`

`optimal_scenario_count: int`[393] `= 0`

`total_scenario_count: int`[394] `= 0`

`success_rate: float`[395] `= 0.0`

`objective_direction: str`[396] `= 'maximize'`

`get_scenario_ranking()` → List[397][tuple[398][str[399], float[400]]]

Get scenarios ranked by objective function value.

Returns scenarios sorted by objective function value according to the optimization direction. For maximization problems, scenarios are sorted in descending order (best to worst). For minimization problems, scenarios are sorted in ascending order (best to worst).

Returns

List of (scenario_name, objective_value) tuples
sorted by performance. Only includes scenarios that achieved optimal solutions.

Return type
List[tuple[401][str[402], float[403]]]

Examples

```
>>> result = analyzer.analyze()
>>> ranking = result.get_scenario_ranking()
>>> for rank, (scenario, value) in enumerate(ranking, 1):
...     print(f"{rank}. {scenario}: {value:.2f}")
```

`get_percentile(`*percentile: float*[404]`)` → float[405] | None[406]

Calculate percentile value for objective function distribution.

Parameters
`percentile` (float[407]) – Percentile value between 0 and 100.

Returns
Percentile value, or None if no optimal scenarios exist.

Return type
Optional[float[408]]

Examples

```
>>> result = analyzer.analyze()
>>> median = result.get_percentile(50)  # Same as statistics['median']
>>> q75 = result.get_percentile(75)     # 75th percentile
```

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = ", scenario_values: ~typing.Dict[str, float] = <factory>, scenario_statuses: ~typing.Dict[str, ~solvers.OXSolverInterface.OXSolutionStatus] = <factory>, statistics: ~typing.Dict[str, float] = <factory>, best_scenario: str | None = None, worst_scenario: str | None = None, optimal_scenario_count: int = 0, total_scenario_count: int = 0, success_rate: float = 0.0, objective_direction: str = 'maximize'*) → None[409]

Right-Hand Side Analysis

class `analysis.OXRightHandSideAnalysis`(*problem:* OXLPProblem *(page 88)* | OXGPProblem *(page 92)* | OXCSPProblem *(page 81), solver:* str[410], *target_constraints:* Set[411][UUID[412]] | None[413] = None, ***kwargs*)

Bases: `object`[414]

Comprehensive Right Hand Side analysis tool for multi-scenario optimization problems.

This class provides systematic analysis of constraint RHS values across different sce-

370 https://docs.python.org/3/library/stdtypes.html#str
371 https://docs.python.org/3/library/functions.html#float
372 https://docs.python.org/3/library/stdtypes.html#str
373 https://docs.python.org/3/library/stdtypes.html#str
374 https://docs.python.org/3/library/functions.html#float
375 https://docs.python.org/3/library/stdtypes.html#str
376 https://docs.python.org/3/library/stdtypes.html#str
377 https://docs.python.org/3/library/functions.html#int
378 https://docs.python.org/3/library/functions.html#int
379 https://docs.python.org/3/library/functions.html#float
380 https://docs.python.org/3/library/stdtypes.html#str
381 https://docs.python.org/3/library/typing.html#typing.Dict
382 https://docs.python.org/3/library/stdtypes.html#str
383 https://docs.python.org/3/library/functions.html#float
384 https://docs.python.org/3/library/typing.html#typing.Dict
385 https://docs.python.org/3/library/stdtypes.html#str
386 https://docs.python.org/3/library/typing.html#typing.Dict
387 https://docs.python.org/3/library/stdtypes.html#str
388 https://docs.python.org/3/library/functions.html#float
389 https://docs.python.org/3/library/stdtypes.html#str
390 https://docs.python.org/3/library/constants.html#None
391 https://docs.python.org/3/library/stdtypes.html#str
392 https://docs.python.org/3/library/constants.html#None
393 https://docs.python.org/3/library/functions.html#int
394 https://docs.python.org/3/library/functions.html#int
395 https://docs.python.org/3/library/functions.html#float
396 https://docs.python.org/3/library/stdtypes.html#str
397 https://docs.python.org/3/library/typing.html#typing.List
398 https://docs.python.org/3/library/stdtypes.html#tuple
399 https://docs.python.org/3/library/stdtypes.html#str
400 https://docs.python.org/3/library/functions.html#float
401 https://docs.python.org/3/library/stdtypes.html#tuple
402 https://docs.python.org/3/library/stdtypes.html#str
403 https://docs.python.org/3/library/functions.html#float
404 https://docs.python.org/3/library/functions.html#float
405 https://docs.python.org/3/library/functions.html#float
406 https://docs.python.org/3/library/constants.html#None
407 https://docs.python.org/3/library/functions.html#float
408 https://docs.python.org/3/library/functions.html#float
409 https://docs.python.org/3/library/constants.html#None

narios in OptiX optimization problems. It uses UUID-based constraint access to track individual constraints across scenarios and provides detailed insights into RHS sensitivity, binding status, and optimization impact analysis.

The analyzer supports both data object scenarios and constraint-specific scenarios, enabling more precise RHS analysis. It automatically discovers all scenarios from both sources, tracks RHS values that may come from constraint scenarios or data scenarios, solves the optimization problem for each unique scenario configuration, and provides comprehensive analysis of constraint behavior and sensitivity to RHS changes.

Key Capabilities:

- **UUID-Based Constraint Tracking**: Uses OptiX's UUID system for precise constraint identification and analysis across scenario variations
- **RHS Value Extraction**: Automatically extracts RHS values from constraints for each scenario, handling scenario data integration seamlessly
- **Binding Status Analysis**: Identifies which constraints are binding (active) in each scenario's optimal solution for bottleneck analysis
- **Shadow Price Analysis**: Extracts and analyzes shadow prices (dual values) to understand marginal value of constraint relaxation
- **Sensitivity Scoring**: Computes numerical sensitivity scores to quantify impact of RHS changes on objective function values
- **Critical Constraint Identification**: Identifies constraints that are consistently binding across scenarios as potential system bottlenecks

`problem`

The optimization problem to analyze with constraints and scenario data.

Type

Union[*OXLPProblem* (page 88), *OXGPProblem* (page 92), *OXCSPProblem* (page 81)]

`solver`

Identifier of the solver to use for all scenario solving operations.

Type

str[415]

`solver_kwargs`

Additional parameters for solver configuration.

Type

Dict[str[416], Any]

`target_constraints`

Specific constraint UUIDs to analyze. If None, analyzes all constraints.

Type

Optional[Set[UUID]]

Examples

Basic RHS analysis across all constraints:

```python
from analysis.OXRightHandSideAnalysis import OXRightHandSideAnalysis

# Create analyzer
analyzer = OXRightHandSideAnalysis(problem, 'ORTools')

# Perform comprehensive RHS analysis
results = analyzer.analyze()

# Access results
print(f"Analyzed {len(results.constraint_analyses)} constraints")
print(f"Critical constraints: {len(results.critical_constraints)}")

# Examine most sensitive constraint
top_sensitive = results.get_top_sensitive_constraints(1)[0]
print(f"Most sensitive: {top_sensitive.constraint_name}")
print(f"Sensitivity score: {top_sensitive.sensitivity_score:.3f}")
```

Analysis with constraint-specific scenarios:

```python
# Create constraints with their own scenarios
capacity_constraint = problem.create_constraint([x, y], [1, 1], "<=", 100)
capacity_constraint.create_scenario("Peak_Hours", rhs=150, name="Peak␣
↪capacity")
capacity_constraint.create_scenario("Off_Peak", rhs=80, name="Off-peak␣
↪capacity")
capacity_constraint.create_scenario("Maintenance", rhs=50, name=
↪"Maintenance mode")

budget_constraint = problem.create_constraint([x, y], [5, 10], "<=", 1000)
budget_constraint.create_scenario("High_Budget", rhs=1500)
budget_constraint.create_scenario("Low_Budget", rhs=800)

# Analyze all constraint scenarios
analyzer = OXRightHandSideAnalysis(problem, 'ORTools')
results = analyzer.analyze()

# Results will include all unique scenarios from constraints
print(f"Total scenarios analyzed: {results.scenario_count}")
print(f"Scenarios: {list(results.scenario_feasibility.keys())}")

# Constraint-specific analysis
cap_analysis = results.get_constraint_analysis(capacity_constraint.id)
print(f"\nCapacity constraint RHS values:")
for scenario, rhs in cap_analysis.rhs_values.items():
    print(f"  {scenario}: {rhs}")
```

Targeted analysis of specific constraints:

```
# Analyze only capacity constraints
capacity_constraint_ids = {constraint.id for constraint in problem.
↪constraints
                          if 'capacity' in constraint.name.lower()}

analyzer = OXRightHandSideAnalysis(
    problem,
    'Gurobi',
    target_constraints=capacity_constraint_ids,
    maxTime=300
)

results = analyzer.analyze()

# Detailed constraint-level analysis
for constraint_id in capacity_constraint_ids:
    analysis = results.get_constraint_analysis(constraint_id)
    stats = analysis.get_rhs_statistics()
    print(f"\nConstraint: {analysis.constraint_name}")
    print(f"RHS Range: [{stats['min']:.1f}, {stats['max']:.1f}]")
    print(f"Binding Rate: {len(analysis.binding_scenarios)/len(analysis.rhs_
↪values):.1%}")
    print(f"Sensitivity: {analysis.sensitivity_score:.3f}")
```

`__init__`(*problem: OXLPProblem (page 88) | OXGPProblem (page 92) | OXCSPProblem (page 81), solver: str*[417]*, target_constraints: Set*[418]*[UUID*[419]*] | None*[420] *= None, **kwargs*)

Initialize the Right Hand Side analyzer.

Parameters

- `problem` (`Union[OXLPProblem` (page 88), `OXGPProblem` (page 92), `OXCSPProblem` (page 81)`]`) – The optimization problem to analyze with constraints and scenario data.
- `solver` (`str`[421]) – The solver identifier to use for scenario solving.
- `target_constraints` (`Optional[Set[UUID]]`) – Specific constraint UUIDs to analyze. If None, analyzes all constraints in the problem.
- `**kwargs` – Additional keyword arguments passed to the solver for scenario solving.

Raises

`OXception` – If the problem has no constraints or if the problem database is empty.

Examples

```
>>> analyzer = OXRightHandSideAnalysis(problem, 'ORTools')
>>> analyzer = OXRightHandSideAnalysis(problem, 'Gurobi', target_
↪constraints={constraint.id}, maxTime=600)
```

`analyze()` → *OXRightHandSideAnalysisResult* (page 228)

Perform comprehensive Right Hand Side analysis across all scenarios.

This method orchestrates the complete RHS analysis workflow including scenario discovery, multi-scenario solving, constraint RHS extraction, binding status analysis, and sensitivity calculation to provide comprehensive RHS insights.

Analysis Workflow:

1. **Scenario Solving**: Uses solve_all_scenarios to solve the problem under each scenario configuration with the specified solver
2. **Constraint Discovery**: Identifies target constraints for analysis based on constructor parameters and problem structure
3. **RHS Extraction**: Extracts RHS values for each constraint across all scenarios, handling scenario data dependencies
4. **Binding Analysis**: Analyzes constraint solutions to identify binding status and slack values for each scenario
5. **Sensitivity Calculation**: Computes sensitivity scores based on correlation between RHS changes and objective function changes
6. **Result Aggregation**: Organizes all analysis results into structured format for easy access and reporting

Returns

Comprehensive RHS analysis results containing
constraint-level analysis, sensitivity metrics, and system-wide RHS insights.

Return type
OXRightHandSideAnalysisResult (page 228)

Raises
`OXception` – If no scenarios are found or if all scenarios fail to solve.

Examples

```
>>> analyzer = OXRightHandSideAnalysis(problem, 'ORTools')
>>> results = analyzer.analyze()
>>> print(f"Most sensitive constraint: {results.get_top_sensitive_
↪constraints(1)[0].constraint_name}")
```

`analyze_constraint_subset(`*constraint_ids: Set[422][UUID[423]]*`)` → Dict[424][UUID[425], *OXConstraintRHSAnalysis* (page 231)]

Analyze a specific subset of constraints for focused RHS analysis.

This method provides targeted analysis of specific constraints, useful for analyzing particular constraint categories or investigating specific bottlenecks.

Parameters
`constraint_ids` (`Set[UUID]`) – Set of constraint UUIDs to analyze.

Returns

Dictionary mapping constraint UUIDs
to their detailed RHS analysis results.

Return type
Dict[UUID, *OXConstraintRHSAnalysis* (page 231)]

Raises
`OXception` – If any specified constraint ID is not found in the problem.

Examples

```
>>> # Analyze only capacity constraints
>>> capacity_ids = {c.id for c in problem.constraints if 'capacity' in␣
↪c.name}
>>> analyzer = OXRightHandSideAnalysis(problem, 'ORTools')
>>> capacity_analysis = analyzer.analyze_constraint_subset(capacity_
↪ids)
>>> for constraint_id, analysis in capacity_analysis.items():
...     print(f"{analysis.constraint_name}: sensitivity = {analysis.
↪sensitivity_score:.3f}")
```

`class analysis.OXRightHandSideAnalysisResult`(*id: ~uuid.UUID = <factory>, class_name: str = ", constraint_analyses: ~typing.Dict[~uuid.UUID, ~analysis.OXRightHandSideAnalysis.OXConstraintRHSAnalysis = <factory>, scenario_feasibility: ~typing.Dict[str, bool] = <factory>, scenario_objective_values: ~typing.Dict[str, float] = <factory>, critical_constraints: ~typing.List[~uuid.UUID] = <factory>, most_sensitive_constraints: ~typing.List[~uuid.UUID] = <factory>, rhs_sensitivity_summary: ~typing.Dict[str, float] = <factory>, scenario_count: int = 0, feasible_scenario_count: int = 0, success_rate: float = 0.0*)

Bases: `OXObject`

[410] https://docs.python.org/3/library/stdtypes.html#str
[411] https://docs.python.org/3/library/typing.html#typing.Set
[412] https://docs.python.org/3/library/uuid.html#uuid.UUID
[413] https://docs.python.org/3/library/constants.html#None
[414] https://docs.python.org/3/library/functions.html#object
[415] https://docs.python.org/3/library/stdtypes.html#str
[416] https://docs.python.org/3/library/stdtypes.html#str
[417] https://docs.python.org/3/library/stdtypes.html#str
[418] https://docs.python.org/3/library/typing.html#typing.Set
[419] https://docs.python.org/3/library/uuid.html#uuid.UUID
[420] https://docs.python.org/3/library/constants.html#None
[421] https://docs.python.org/3/library/stdtypes.html#str
[422] https://docs.python.org/3/library/typing.html#typing.Set
[423] https://docs.python.org/3/library/uuid.html#uuid.UUID
[424] https://docs.python.org/3/library/typing.html#typing.Dict
[425] https://docs.python.org/3/library/uuid.html#uuid.UUID

Comprehensive data structure containing Right Hand Side analysis results.

This class encapsulates all analysis results from multi-scenario RHS evaluation, providing structured access to constraint-level analysis, scenario comparisons, and system-wide RHS sensitivity insights for optimization model analysis.

`constraint_analyses`

Dictionary mapping constraint UUIDs to their detailed RHS analysis results.

Type

Dict[UUID, *OXConstraintRHSAnalysis* (page 231)]

`scenario_feasibility`

Dictionary mapping scenario names to their feasibility status across all constraints.

Type

Dict[str[426], bool[427]]

`scenario_objective_values`

Dictionary mapping scenario names to optimal objective function values.

Type

Dict[str[428], float[429]]

`critical_constraints`

List of constraint UUIDs identified as critical based on binding frequency analysis.

Type

List[UUID]

`most_sensitive_constraints`

List of constraint UUIDs with highest sensitivity scores for RHS changes.

Type

List[UUID]

`rhs_sensitivity_summary`

System-wide RHS sensitivity metrics including average sensitivity and variability.

Type

Dict[str[430], float[431]]

`scenario_count`

Total number of scenarios analyzed in the study.

Type

int[432]

`feasible_scenario_count`

Number of scenarios that yielded feasible solutions.

Type

int[433]

`success_rate`

Percentage of scenarios with optimal solutions.

Type

float[434]

`constraint_analyses: Dict`[435]`[UUID`[436]`, OXConstraintRHSAnalysis (page 231)]`

`scenario_feasibility: Dict`[437]`[str`[438]`, bool`[439]`]`

`scenario_objective_values: Dict`[440]`[str`[441]`, float`[442]`]`

`critical_constraints: List`[443]`[UUID`[444]`]`

`most_sensitive_constraints: List`[445]`[UUID`[446]`]`

`rhs_sensitivity_summary: Dict`[447]`[str`[448]`, float`[449]`]`

`scenario_count: int`[450] `= 0`

`feasible_scenario_count: int`[451] `= 0`

`success_rate: float`[452] `= 0.0`

`get_constraint_analysis(`*constraint_id: UUID*[453]`)` → *OXConstraintRHSAnalysis* (page 231) | None[454]

Retrieve detailed analysis for a specific constraint.

Parameters
`constraint_id` (`UUID`) – Unique identifier of the constraint.

Returns

Detailed constraint analysis or None
if constraint ID not found.

Return type
Optional[*OXConstraintRHSAnalysis* (page 231)]

`get_top_sensitive_constraints(`*top_n: int*[455] *= 5*`)` → List[456][*OXConstraintRHSAnalysis* (page 231)]

Get the most sensitive constraints ranked by sensitivity score.

Parameters
`top_n` (`int`[457]) – Number of top constraints to return.

Returns

List of constraint analyses sorted by
sensitivity score in descending order.

Return type
List[*OXConstraintRHSAnalysis* (page 231)]

`get_constraints_by_binding_frequency(`*min_frequency: float*[458] *= 0.3*`)` → List[459][*OXConstraintRHSAnalysis* (page 231)]

Get constraints that are binding in at least min_frequency of scenarios.

Parameters
`min_frequency` (`float`[460]) – Minimum binding frequency (0.0 to 1.0).

Returns
List of constraints meeting binding criteria.

Return type
List[*OXConstraintRHSAnalysis* (page 231)]

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = ”, constraint_analyses: ~typing.Dict[~uuid.UUID, ~analysis.OXRightHandSideAnalysis.OXConstraintRHSAnalysis] = <factory>, scenario_feasibility: ~typing.Dict[str, bool] = <factory>, scenario_objective_values: ~typing.Dict[str, float] = <factory>, critical_constraints: ~typing.List[~uuid.UUID] = <factory>, most_sensitive_constraints: ~typing.List[~uuid.UUID] = <factory>, rhs_sensitivity_summary: ~typing.Dict[str, float] = <factory>, scenario_count: int = 0, feasible_scenario_count: int = 0, success_rate: float = 0.0*) → None[461]

`class analysis.OXConstraintRHSAnalysis`(*id: ~uuid.UUID = <factory>, class_name: str = ”, constraint_id: ~uuid.UUID = <factory>, constraint_name: str = ”, rhs_values: ~typing.Dict[str, float] = <factory>, binding_scenarios: ~typing.List[str] = <factory>, shadow_prices: ~typing.Dict[str, float] = <factory>, slack_values: ~typing.Dict[str, float] = <factory>, rhs_range: ~typing.Dict[str, float] = <factory>, sensitivity_score: float = 0.0, constraint_type: str = ”*)

Bases: `OXObject`

[426] https://docs.python.org/3/library/stdtypes.html#str
[427] https://docs.python.org/3/library/functions.html#bool
[428] https://docs.python.org/3/library/stdtypes.html#str
[429] https://docs.python.org/3/library/functions.html#float
[430] https://docs.python.org/3/library/stdtypes.html#str
[431] https://docs.python.org/3/library/functions.html#float
[432] https://docs.python.org/3/library/functions.html#int
[433] https://docs.python.org/3/library/functions.html#int
[434] https://docs.python.org/3/library/functions.html#float
[435] https://docs.python.org/3/library/typing.html#typing.Dict
[436] https://docs.python.org/3/library/uuid.html#uuid.UUID
[437] https://docs.python.org/3/library/typing.html#typing.Dict
[438] https://docs.python.org/3/library/stdtypes.html#str
[439] https://docs.python.org/3/library/functions.html#bool
[440] https://docs.python.org/3/library/typing.html#typing.Dict
[441] https://docs.python.org/3/library/stdtypes.html#str
[442] https://docs.python.org/3/library/functions.html#float
[443] https://docs.python.org/3/library/typing.html#typing.List
[444] https://docs.python.org/3/library/uuid.html#uuid.UUID
[445] https://docs.python.org/3/library/typing.html#typing.List
[446] https://docs.python.org/3/library/uuid.html#uuid.UUID
[447] https://docs.python.org/3/library/typing.html#typing.Dict
[448] https://docs.python.org/3/library/stdtypes.html#str
[449] https://docs.python.org/3/library/functions.html#float
[450] https://docs.python.org/3/library/functions.html#int
[451] https://docs.python.org/3/library/functions.html#int
[452] https://docs.python.org/3/library/functions.html#float
[453] https://docs.python.org/3/library/uuid.html#uuid.UUID
[454] https://docs.python.org/3/library/constants.html#None
[455] https://docs.python.org/3/library/functions.html#int
[456] https://docs.python.org/3/library/typing.html#typing.List
[457] https://docs.python.org/3/library/functions.html#int
[458] https://docs.python.org/3/library/functions.html#float
[459] https://docs.python.org/3/library/typing.html#typing.List
[460] https://docs.python.org/3/library/functions.html#float
[461] https://docs.python.org/3/library/constants.html#None

Analysis results for a specific constraint's RHS behavior across scenarios.

This class encapsulates detailed analysis of how a single constraint's right-hand side values change across scenarios and the resulting impact on optimization outcomes, binding status, and shadow prices.

`constraint_id`

Unique identifier of the analyzed constraint.

Type
UUID

`constraint_name`

Human-readable name of the constraint for reporting.

Type
str[462]

`rhs_values`

Dictionary mapping scenario names to their corresponding RHS values for this constraint.

Type
Dict[str[463], float[464]]

`binding_scenarios`

List of scenario names where this constraint is binding (active) at the optimal solution.

Type
List[str[465]]

`shadow_prices`

Dictionary mapping scenario names to the shadow price (dual value) of this constraint.

Type
Dict[str[466], float[467]]

`slack_values`

Dictionary mapping scenario names to the slack value of this constraint at optimum.

Type
Dict[str[468], float[469]]

`rhs_range`

Statistical summary of RHS values including min, max, mean, and standard deviation.

Type
Dict[str[470], float[471]]

`sensitivity_score`

Numerical measure of how sensitive the objective function is to changes in this constraint's RHS.

Type
float[472]

`constraint_type`

The relational operator type (<=, >=, =) for context.

Type

str[473]

`constraint_id: UUID[474]`

`constraint_name: str[475] = ''`

`rhs_values: Dict[476][str[477], float[478]]`

`binding_scenarios: List[479][str[480]]`

`shadow_prices: Dict[481][str[482], float[483]]`

`slack_values: Dict[484][str[485], float[486]]`

`rhs_range: Dict[487][str[488], float[489]]`

`sensitivity_score: float[490] = 0.0`

`constraint_type: str[491] = ''`

`get_rhs_statistics()` → Dict[492][str[493], float[494]]

Calculate comprehensive statistics for RHS values across scenarios.

Returns

Statistical metrics including mean, median, std_dev,
min, max, range, and coefficient of variation.

Return type

Dict[str[495], float[496]]

`is_critical_constraint`(*binding_threshold: float[497] = 0.5*) → bool[498]

Determine if this constraint is critical based on binding frequency.

Parameters

`binding_threshold` (float[499]) – Minimum fraction of scenarios where constraint must be binding to be considered critical.

Returns

True if constraint is binding in more than binding_threshold
fraction of scenarios.

Return type

bool[500]

`__init__`(*id: ~uuid.UUID = <factory>, class_name: str = '', constraint_id: ~uuid.UUID = <factory>, constraint_name: str = '', rhs_values: ~typing.Dict[str, float] = <factory>, binding_scenarios: ~typing.List[str] = <factory>, shadow_prices: ~typing.Dict[str, float] = <factory>, slack_values: ~typing.Dict[str, float] = <factory>, rhs_range: ~typing.Dict[str, float] = <factory>, sensitivity_score: float = 0.0, constraint_type: str = ''*) → None[501]

9.11.2 Examples

Basic Analysis Workflow

```python
from analysis import OXObjectiveFunctionAnalysis, OXRightHandSideAnalysis
from problem import OXLPProblem

# Create and configure your optimization problem
problem = OXLPProblem(name="Production Planning")

# Set up variables, constraints, objective function with scenario data
# ... problem configuration code ...

# Perform objective function analysis
obj_analyzer = OXObjectiveFunctionAnalysis(problem, 'ORTools')
obj_results = obj_analyzer.analyze()

# Perform RHS constraint analysis
```

(continues on next page)

[462] https://docs.python.org/3/library/stdtypes.html#str
[463] https://docs.python.org/3/library/stdtypes.html#str
[464] https://docs.python.org/3/library/functions.html#float
[465] https://docs.python.org/3/library/stdtypes.html#str
[466] https://docs.python.org/3/library/stdtypes.html#str
[467] https://docs.python.org/3/library/functions.html#float
[468] https://docs.python.org/3/library/stdtypes.html#str
[469] https://docs.python.org/3/library/functions.html#float
[470] https://docs.python.org/3/library/stdtypes.html#str
[471] https://docs.python.org/3/library/functions.html#float
[472] https://docs.python.org/3/library/functions.html#float
[473] https://docs.python.org/3/library/stdtypes.html#str
[474] https://docs.python.org/3/library/uuid.html#uuid.UUID
[475] https://docs.python.org/3/library/stdtypes.html#str
[476] https://docs.python.org/3/library/typing.html#typing.Dict
[477] https://docs.python.org/3/library/stdtypes.html#str
[478] https://docs.python.org/3/library/functions.html#float
[479] https://docs.python.org/3/library/typing.html#typing.List
[480] https://docs.python.org/3/library/stdtypes.html#str
[481] https://docs.python.org/3/library/typing.html#typing.Dict
[482] https://docs.python.org/3/library/stdtypes.html#str
[483] https://docs.python.org/3/library/functions.html#float
[484] https://docs.python.org/3/library/typing.html#typing.Dict
[485] https://docs.python.org/3/library/stdtypes.html#str
[486] https://docs.python.org/3/library/functions.html#float
[487] https://docs.python.org/3/library/typing.html#typing.Dict
[488] https://docs.python.org/3/library/stdtypes.html#str
[489] https://docs.python.org/3/library/functions.html#float
[490] https://docs.python.org/3/library/functions.html#float
[491] https://docs.python.org/3/library/stdtypes.html#str
[492] https://docs.python.org/3/library/typing.html#typing.Dict
[493] https://docs.python.org/3/library/stdtypes.html#str
[494] https://docs.python.org/3/library/functions.html#float
[495] https://docs.python.org/3/library/stdtypes.html#str
[496] https://docs.python.org/3/library/functions.html#float
[497] https://docs.python.org/3/library/functions.html#float
[498] https://docs.python.org/3/library/functions.html#bool
[499] https://docs.python.org/3/library/functions.html#float
[500] https://docs.python.org/3/library/functions.html#bool
[501] https://docs.python.org/3/library/constants.html#None

(continued from previous page)

```
rhs_analyzer = OXRightHandSideAnalysis(problem, 'ORTools')
rhs_results = rhs_analyzer.analyze()

# Access analysis results
print(f"Best scenario: {obj_results.best_scenario}")
print(f"Worst scenario: {obj_results.worst_scenario}")
print(f"Success rate: {obj_results.success_rate:.1%}")
print(f"Critical constraints: {len(rhs_results.critical_constraints)}")
```

Objective Function Analysis

```
from analysis import OXObjectiveFunctionAnalysis
from problem import OXLPProblem

# Assume problem is already configured with multiple scenarios
analyzer = OXObjectiveFunctionAnalysis(problem, solver_name='ORTools')

# Run comprehensive analysis across all scenarios
results = analyzer.analyze()

# Access detailed results
print(f"Total scenarios analyzed: {results.total_scenarios}")
print(f"Successful solutions: {results.successful_scenarios}")
print(f"Failed scenarios: {results.failed_scenarios}")

# Best and worst case scenarios
if results.best_scenario:
    print(f"Best objective value: {results.best_value}")
    print(f"Best scenario ID: {results.best_scenario}")

if results.worst_scenario:
    print(f"Worst objective value: {results.worst_value}")
    print(f"Worst scenario ID: {results.worst_scenario}")

# Statistical summary
print(f"Average objective value: {results.average_value:.2f}")
print(f"Standard deviation: {results.std_deviation:.2f}")
print(f"Value range: [{results.min_value}, {results.max_value}]")
```

Right-Hand Side Analysis

```
from analysis import OXRightHandSideAnalysis
from problem import OXGPProblem

# Create analyzer for goal programming problem
analyzer = OXRightHandSideAnalysis(problem, solver_name='Gurobi')

# Analyze constraint behavior across scenarios
```

(continues on next page)

(continued from previous page)

```
results = analyzer.analyze()

# Check critical constraints
for constraint_analysis in results.critical_constraints:
    constraint_id = constraint_analysis.constraint_id
    print(f"Critical constraint: {constraint_id}")
    print(f"  Binding frequency: {constraint_analysis.binding_frequency:.1%}")
    print(f"  Average slack: {constraint_analysis.average_slack:.2f}")
    print(f"  Never feasible in: {len(constraint_analysis.infeasible_scenarios)}
↪ scenarios")

# Analyze specific constraint
constraint_id = "resource_capacity_constraint_uuid"
if constraint_id in results.constraint_analyses:
    analysis = results.constraint_analyses[constraint_id]
    print(f"Constraint {constraint_id} analysis:")
    print(f"  Min slack: {analysis.min_slack}")
    print(f"  Max slack: {analysis.max_slack}")
    print(f"  Binding scenarios: {analysis.binding_scenarios}")
```

Scenario Comparison

```
from analysis import OXObjectiveFunctionAnalysis, OXRightHandSideAnalysis

# Compare objective function performance
obj_analyzer = OXObjectiveFunctionAnalysis(problem, 'ORTools')
obj_results = obj_analyzer.analyze()

# Identify performance outliers
if obj_results.std_deviation > 0:
    z_scores = {}
    for scenario_id, value in obj_results.scenario_values.items():
        z_score = (value - obj_results.average_value) / obj_results.std_deviation
        if abs(z_score) > 2:  # Outlier threshold
            z_scores[scenario_id] = z_score

    print(f"Found {len(z_scores)} outlier scenarios")
    for scenario_id, z_score in sorted(z_scores.items(), key=lambda x: abs(x[1]),
↪ reverse=True):
        print(f"  Scenario {scenario_id}: Z-score = {z_score:.2f}")
```

Multi-Solver Analysis

```
from analysis import OXObjectiveFunctionAnalysis

# Compare solver performance
solvers = ['ORTools', 'Gurobi']
solver_results = {}
```

(continues on next page)

(continued from previous page)

```python
for solver in solvers:
    try:
        analyzer = OXObjectiveFunctionAnalysis(problem, solver)
        results = analyzer.analyze()
        solver_results[solver] = results
        print(f"{solver}: Success rate = {results.success_rate:.1%}, "
              f"Avg value = {results.average_value:.2f}")
    except Exception as e:
        print(f"{solver} failed: {e}")

# Compare results
if len(solver_results) > 1:
    values = [r.average_value for r in solver_results.values()]
    if max(values) - min(values) > 0.01:
        print("Warning: Significant difference in solver results detected")
```

Performance Analysis

```python
import time
from analysis import OXObjectiveFunctionAnalysis, OXRightHandSideAnalysis

# Time analysis operations
start_time = time.time()

# Run both analyses
obj_analyzer = OXObjectiveFunctionAnalysis(problem, 'ORTools')
obj_results = obj_analyzer.analyze()

rhs_analyzer = OXRightHandSideAnalysis(problem, 'ORTools')
rhs_results = rhs_analyzer.analyze()

analysis_time = time.time() - start_time

# Performance metrics
scenarios_per_second = obj_results.total_scenarios / analysis_time
print(f"Analysis completed in {analysis_time:.2f} seconds")
print(f"Processing rate: {scenarios_per_second:.1f} scenarios/second")

# Memory efficiency check
total_constraints = len(problem.constraints)
total_scenarios = obj_results.total_scenarios
total_analyses = total_constraints * total_scenarios
print(f"Analyzed {total_analyses:,} constraint-scenario combinations")
```

Integration with Problem Types

```python
from analysis import OXObjectiveFunctionAnalysis
from problem import OXLPProblem, OXGPProblem, OXCSPProblem

def analyze_problem(problem, solver='ORTools'):
    """Analyze any OptiX problem type."""

    # Objective function analysis (not applicable to CSP)
    if not isinstance(problem, OXCSPProblem):
        obj_analyzer = OXObjectiveFunctionAnalysis(problem, solver)
        obj_results = obj_analyzer.analyze()

        print(f"Problem: {problem.name}")
        print(f"Type: {type(problem).__name__}")
        print(f"Objective analysis:")
        print(f"  Success rate: {obj_results.success_rate:.1%}")
        print(f"  Value range: [{obj_results.min_value}, {obj_results.max_value}]
↪")

        # Goal programming specific
        if isinstance(problem, OXGPProblem):
            print(f"  Note: Values represent weighted deviation sums")

    # RHS analysis (applicable to all problem types)
    rhs_analyzer = OXRightHandSideAnalysis(problem, solver)
    rhs_results = rhs_analyzer.analyze()

    print(f"Constraint analysis:")
    print(f"  Total constraints: {len(problem.constraints)}")
    print(f"  Critical constraints: {len(rhs_results.critical_constraints)}")
    print(f"  Always binding: {rhs_results.always_binding_count}")
    print(f"  Never binding: {rhs_results.never_binding_count}")

# Usage with different problem types
lp_problem = OXLPProblem(name="Linear Program")
gp_problem = OXGPProblem(name="Goal Program")
csp_problem = OXCSPProblem(name="Constraint Satisfaction")

for problem in [lp_problem, gp_problem, csp_problem]:
    analyze_problem(problem)
    print("-" * 50)
```

9.11.3 See Also

- *Problem Module* (page 81) - Problem classes that generate analysis data
- *Constraints Module* (page 105) - Constraint definitions analyzed by RHS analysis
- *Solvers Module* (page 163) - Solver implementations used in analysis
- *Data Module* (page 205) - Data management for scenario-based analysis

- ../user_guide/analysis - Advanced analysis techniques guide

9.12 Utilities Module

The utilities module provides simple utility functions for dynamic class loading within the OptiX framework. It contains basic functions that support object serialization and deserialization by enabling runtime class resolution.

9.12.1 Module Functions

Class Loading Functions

`utilities.get_fully_qualified_name`(*cls: type*[502]) → str[503]

Generate a fully qualified name string for a Python class.

This function creates a string representation of a class by concatenating the module name and class name with a dot separator. The result can be used with the `load_class()` (page 239) function to dynamically load the class.

Parameters
`cls` (type[504]) – The class object to generate a name for.

Returns
A string in the format `module_name.ClassName`.

Return type
str[505]

Examples

Generate fully qualified names for classes:

```
from base.OXObject import OXObject

# Get the class name string
name = get_fully_qualified_name(OXObject)
print(name)  # Output: 'base.OXObject.OXObject'

# Works with built-in types too
list_name = get_fully_qualified_name(list)
print(list_name)  # Output: 'builtins.list'
```

See also

`load_class()` (page 239): Load a class from its fully qualified name.

[502] https://docs.python.org/3/library/functions.html#type
[503] https://docs.python.org/3/library/stdtypes.html#str
[504] https://docs.python.org/3/library/functions.html#type
[505] https://docs.python.org/3/library/stdtypes.html#str

`utilities.load_class`(*fully_qualified_name: str*[506]) → type[507]

Dynamically load a Python class from its fully qualified name.

This function loads a class by parsing the fully qualified name string, importing the module, and retrieving the class object from the module.

Parameters

`fully_qualified_name` (str[508]) – The fully qualified name of the class to load in the format `module.ClassName`.

Returns

The loaded class object.

Return type

type[509]

Raises

`OXception` – If the class cannot be loaded due to import errors or missing class names.

Examples

Load classes dynamically:

```
# Load a framework class
obj_class = load_class("base.OXObject.OXObject")
instance = obj_class()

# Roundtrip demonstration
from base.OXObject import OXObject
name = get_fully_qualified_name(OXObject)
loaded_class = load_class(name)
assert loaded_class is OXObject
```

See also

`get_fully_qualified_name()` (page 239): Generate class names for use with this function.

9.12.2 Examples

Basic Class Loading

```
from utilities import get_fully_qualified_name, load_class
from base.OXObject import OXObject

# Generate module.ClassName string
class_id = get_fully_qualified_name(OXObject)
```

(continues on next page)

506 https://docs.python.org/3/library/stdtypes.html#str
507 https://docs.python.org/3/library/functions.html#type
508 https://docs.python.org/3/library/stdtypes.html#str
509 https://docs.python.org/3/library/functions.html#type

(continued from previous page)

```
print(class_id)  # Output: 'base.OXObject.OXObject'

# Dynamically load the class
loaded_class = load_class(class_id)
instance = loaded_class()

# Verify roundtrip integrity
assert loaded_class is OXObject
```

Integration with Serialization

```
from utilities import get_fully_qualified_name, load_class
from serialization import serialize_to_python_dict, deserialize_from_python_dict
from problem.OXProblem import OXLPProblem

# Create a problem
problem = OXLPProblem(name="Test Problem")

# Serialize to dictionary
problem_dict = serialize_to_python_dict(problem)
print(f"Class name in serialized data: {problem_dict['class_name']}")

# Deserialize from dictionary
restored_problem = deserialize_from_python_dict(problem_dict)
assert restored_problem.name == "Test Problem"
```

Error Handling

```
from utilities import load_class
from base.OXception import OXception

try:
    # Attempt to load a non-existent class
    bad_class = load_class("nonexistent.module.BadClass")
except OXception as e:
    print(f"Failed to load class: {e}")
```

9.12.3 See Also

- base - Base classes that use the utilities module
- serialization - Serialization system that relies on dynamic class loading
- ../development/architecture - Framework architecture overview

9.13 Changelog

All notable changes to the OptiX project are documented in this file.

The format is based on Keep a Changelog[510], and this project adheres to Semantic Versioning[511].

9.13.1 [Unreleased]

Added

- OR-Tools solver integration with `OXORToolsSolverInterface`
- Comprehensive solver interface framework (`OXSolverInterface`)
- Solution management system with `OXSolverSolution` and `OXSolutionStatus`
- Special constraints support (`OXSpecialConstraints`)
- Solver factory pattern for easy solver selection
- Bus assignment problem example demonstrating real-world usage
- Diet problem optimization example showcasing classic linear programming
- Enhanced constraint value tracking and evaluation
- Comprehensive package structure with proper `__init__.py` files
- Extended test coverage for all major components
- Comprehensive API documentation across all modules
- Complete Sphinx documentation with modern themes

Enhanced

- Problem classes now support constraint satisfaction problems (CSP)
- Improved variable creation from database objects
- Enhanced expression handling in `OXpression`
- Better serialization support for complex data structures
- Extended utility functions for class loading and management
- Documentation coverage for base, data, constraints, OXpression, serialization, utilities, variables, solvers, and test modules
- Sample problem documentation with detailed API references

[510] https://keepachangelog.com/en/1.0.0/
[511] https://semver.org/spec/v2.0.0.html

Fixed

- Core framework bugs and improved test functionality
- Variable and constraint management in solver interfaces
- Solution retrieval and value tracking
- Database integration and object relationships
- Fraction calculation and import paths in constraints module

9.13.2 [1.0.0] - 2024-12-15

Added

- Initial stable release of OptiX Mathematical Optimization Framework
- Complete documentation system with Sphinx
- Multi-solver architecture supporting OR-Tools and Gurobi
- Three problem types: CSP, LP, and GP with progressive complexity
- Special constraints for non-linear operations
- Database integration with OXData and OXDatabase
- Comprehensive examples and tutorials
- Full API documentation
- Custom HTML and LaTeX themes
- Interactive documentation features

Changed

- Reorganized project structure for better maintainability
- Improved import paths and module organization
- Enhanced error handling and validation
- Standardized naming conventions across all modules

9.13.3 [0.1.0] - 2024-06-01

Added

- Initial release of OptiX
- Framework for defining and solving optimization problems
- Support for linear programming (LP) and goal programming (GP)
- Decision variables with bounds
- Constraint definition with relational operators
- Objective function creation (minimize/maximize)
- Custom exception handling with OXception

- Data management with OXData and OXDatabase
- Variable management with OXVariable and OXVariableSet
- Serialization utilities
- Class loading utilities

Requirements

- Python 3.12 or higher
- Poetry for dependency management

9.13.4 Migration Guide

Upgrading from 0.1.0 to 1.0.0

Import Changes:

```
# Old (0.1.0)
from problem.OXProblem import OXLPProblem
from constraints.OXConstraint import RelationalOperators
from solvers.OXSolverFactory import solve

# New (1.0.0)
from problem import OXLPProblem, ObjectiveType
from constraints import RelationalOperators
from solvers import solve
```

API Changes:

- Simplified import structure
- Enhanced solver interface
- Improved error handling
- Better documentation integration

New Features:

- Gurobi solver support
- Special constraints
- Comprehensive documentation
- Enhanced examples

9.13.5 Deprecation Notices

Version 1.0.0: - None

Future Deprecations: - Legacy import paths will be deprecated in version 2.0.0 - Direct solver instantiation will be replaced by factory pattern

9.13.6 Breaking Changes

Version 1.0.0: - Import path restructuring (see migration guide) - Solver interface standardization - Enhanced type checking

9.13.7 Security Updates

Version 1.0.0: - Enhanced input validation - Improved error handling - Secure serialization methods

9.13.8 Performance Improvements

Version 1.0.0: - Optimized variable and constraint management - Improved solver interface performance - Enhanced memory usage for large problems - Better algorithm complexity for search operations

9.13.9 Bug Fixes

Version 1.0.0: - Fixed constraint evaluation edge cases - Resolved variable bounds validation issues - Corrected serialization of complex objects - Fixed solver status reporting

9.13.10 Known Issues

Current Issues: - None known

Workarounds: - For very large problems (>100k variables), consider problem decomposition - Use appropriate solver timeouts for complex problems

9.13.11 Contributing

Contributions to OptiX are welcome! Please see our contribution guidelines:

1. **Bug Reports**: Use GitHub Issues with detailed reproduction steps
2. **Feature Requests**: Discuss in GitHub Discussions before implementation
3. **Code Contributions**: Follow our development guidelines
4. **Documentation**: Help improve and expand documentation

Reporting Issues

When reporting issues, please include:

- OptiX version
- Python version
- Operating system
- Minimal reproduction example
- Expected vs. actual behavior
- Error messages and stack traces

9.13.12 Release Process

OptiX follows semantic versioning:

- **Major** (X.0.0): Breaking changes, major new features
- **Minor** (0.X.0): New features, enhancements, backwards compatible
- **Patch** (0.0.X): Bug fixes, documentation updates

Release Schedule

- **Major releases**: Annually
- **Minor releases**: Quarterly
- **Patch releases**: As needed for critical fixes

9.13.13 Acknowledgments

Core Contributors: - Tolga BERBER - Lead Developer & Project Architect - Beyzanur SİYAH - Core Developer & Research Assistant

Special Thanks: - OR-Tools team for the excellent optimization library - Gurobi team for solver integration support - OptiX community for feedback and contributions

Dependencies: - OR-Tools: Google's optimization tools - Gurobi: Commercial optimization solver - Python ecosystem: NumPy, SciPy, and other supporting libraries

9.13.14 License Information

OptiX is licensed under the Academic Free License (AFL) v. 3.0. See the LICENSE file for full license text.

Key Points: - Academic and research use encouraged - Commercial use permitted with attribution - Modifications and redistribution allowed - No warranty provided

9.13.15 Support and Resources

Documentation: - Complete API reference - Tutorials and examples - User guides and best practices

Community: - GitHub Discussions for questions and ideas - GitHub Issues for bug reports - Academic publications and research papers

Professional Support: - Consulting services available - Custom development and integration - Training and workshops

9.13.16 Version Comparison

9.13.17 Download Information

Current Stable Release: 1.0.0

Installation:

```
# Latest stable
git clone https://github.com/yourusername/optix.git
cd OptiX
poetry install
```

Development Version:

```
# Development branch
git clone -b develop https://github.com/yourusername/optix.git
```

Release Archives: - v1.0.0 Source[512] - v0.1.0 Source[513]

9.13.18 Statistics

Project Metrics (v1.0.0): - Lines of code: ~15,000 - Test coverage: >95% - Documentation pages: 50+ - Example problems: 10+ - Supported platforms: Windows, macOS, Linux

Community Growth: - Contributors: 2+ - GitHub stars: Growing - Academic citations: In progress - Commercial adoption: Emerging

9.14 License

OptiX is licensed under the Academic Free License (AFL) v. 3.0.

9.14.1 Academic Free License v. 3.0

This Academic Free License (the "License") applies to any original work of authorship (the "Original Work") whose owner (the "Licensor") has placed the following licensing notice adjacent to the copyright notice for the Original Work:

Licensed under the Academic Free License version 3.0

1) **Grant of Copyright License.** Licensor grants You a worldwide, royalty-free, non-exclusive, sublicensable license, for the duration of the copyright, to do the following:

    a) to reproduce the Original Work in copies, either alone or as part of a collective work;

    b) to translate, adapt, alter, transform, modify, or arrange the Original Work, thereby creating derivative works ("Derivative Works") based upon the Original Work;

    c) to distribute or communicate copies of the Original Work and Derivative Works to the public, under any license of your choice that does not contradict the terms and conditions, including Licensor's reserved rights and remedies, in this Academic Free License;

    d) to perform the Original Work publicly; and

    e) to display the Original Work publicly.

[512] https://github.com/yourusername/optix/archive/v1.0.0.tar.gz
[513] https://github.com/yourusername/optix/archive/v0.1.0.tar.gz

2) **Grant of Patent License.** Licensor grants You a worldwide, royalty-free, non-exclusive, sublicensable license, under patent claims owned or controlled by the Licensor that are embodied in the Original Work as furnished by the Licensor, for the duration of the patents, to make, use, sell, offer for sale, have made, and import the Original Work and Derivative Works.

3) **Grant of Source Code License.** The term "Source Code" means the preferred form of the Original Work for making modifications to it and all available documentation describing how to modify the Original Work. Licensor agrees to provide a machine-readable copy of the Source Code of the Original Work along with each copy of the Original Work that Licensor distributes. Licensor reserves the right to satisfy this obligation by placing a machine-readable copy of the Source Code in an information repository reasonably calculated to permit inexpensive and convenient access by You for as long as Licensor continues to distribute the Original Work.

4) **Exclusions From License Grant.** Neither the names of Licensor, nor the names of any contributors to the Original Work, nor any of their trademarks or service marks, may be used to endorse or promote products derived from this Original Work without express prior permission of the Licensor. Except as expressly stated herein, nothing in this License grants any license to Licensor's trademarks, copyrights, patents, trade secrets or any other intellectual property. No patent license is granted to make, use, sell, offer for sale, have made, or import embodiments of any patent claims other than the licensed claims defined in Section 2. No license is granted to the trademarks of Licensor even if such marks are included in the Original Work. Nothing in this License shall be interpreted to prohibit Licensor from licensing under terms different from this License any Original Work that Licensor otherwise would have a right to license.

5) **External Deployment.** The term "External Deployment" means the use, distribution, or communication of the Original Work or Derivative Works in any way such that the Original Work or Derivative Works may be used by anyone other than You, whether those works are distributed or communicated to those persons or made available as an application intended for use over a network. As an express condition for the grants of license hereunder, You must treat any External Deployment by You of the Original Work or a Derivative Work as a distribution under section 1(c).

6) **Attribution Rights.** You must retain, in the Source Code of any Derivative Works that You create, all copyright, patent, or trademark notices from the Source Code of the Original Work, as well as any notices of licensing and any descriptive text identified therein as an "Attribution Notice." You must cause the Source Code for any Derivative Works that You create to carry a prominent Attribution Notice reasonably calculated to inform recipients that You have modified the Original Work.

7) **Warranty of Provenance and Disclaimer of Warranty.** Licensor warrants that the copyright in and to the Original Work and the patent rights granted herein by Licensor are owned by the Licensor or are sublicensed to You under the terms of this License with the permission of the contributor(s) of those copyrights and patent rights. Except as expressly stated in the immediately preceding sentence, the Original Work is provided under this License on an "AS IS" BASIS and WITHOUT WARRANTY, either express or implied, including, without limitation, the warranties of non-infringement, merchantability or fitness for a particular purpose. THE ENTIRE RISK AS TO THE QUALITY OF THE ORIGINAL WORK IS WITH YOU. This DISCLAIMER OF WARRANTY constitutes an essential part of this License. No license to the Original Work is granted by this License except under this disclaimer.

8) **Limitation of Liability.** Under no circumstances and under no legal theory, whether in tort (including negligence), contract, or otherwise, shall the Licensor be liable to anyone for any indirect, special, incidental, or consequential damages of any character arising as a result of this License or the use of the Original Work including, without limitation, damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses. This limitation of liability shall not apply to the extent applicable law prohibits such limitation.

9) **Acceptance and Termination.** If, at any time, You expressly assented to this License, that assent indicates your acceptance of this License and all of its terms and conditions. If You distribute or communicate copies of the Original Work or a Derivative Work, You must make a reasonable effort under the circumstances to obtain the express assent of recipients to the terms of this License. This License conditions your rights to undertake the activities listed in Section 1, including your right to create Derivative Works based upon the Original Work, and doing so without honoring these terms and conditions is prohibited by copyright law and international treaty. Nothing in this License is intended to affect copyright exceptions and limitations (including "fair use" or "fair dealing"). This License shall terminate immediately and You may no longer exercise any of the rights granted to You by this License upon your failure to honor the conditions in Section 1(c).

10) **Termination for Patent Action.** This License shall terminate automatically and You may no longer exercise any of the rights granted to You by this License as of the date You commence an action, including a cross-claim or counterclaim, against Licensor or any licensee alleging that the Original Work infringes a patent. This termination provision shall not apply for an action alleging patent infringement by combinations of the Original Work with other software or hardware.

11) **Jurisdiction, Venue and Governing Law.** Any action or suit relating to this License may be brought only in the courts of a jurisdiction wherein the Licensor resides or in which Licensor conducts its primary business, and under the laws of that jurisdiction excluding its conflict-of-law provisions. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded. Any use of the Original Work outside the scope of this License or after its termination shall be subject to the requirements and penalties of copyright or patent law in the appropriate jurisdiction. This section shall survive the termination of this License.

12) **Attorneys' Fees.** In any action to enforce the terms of this License or seeking damages relating thereto, the prevailing party shall be entitled to recover its costs and expenses, including, without limitation, reasonable attorneys' fees and costs incurred in connection with such action, including any appeal of such action. This section shall survive the termination of this License.

13) **Miscellaneous.** If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable.

14) **Definition of "You" in This License.** "You" throughout this License, whether in upper or lower case, means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License. For legal entities, "You" includes any entity that controls, is controlled by, or is under common control with you. For purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

15) **Right to Use.** You may use the Original Work in all ways not otherwise restricted or

conditioned by this License or by law, and Licensor promises not to interfere with or be responsible for such uses by You.

16) **Modification of This License.** This License is Copyright © 2005 Lawrence Rosen. Permission is granted to copy, distribute, or communicate this License without modification. Nothing in this License permits You to modify this License as applied to the Original Work or to Derivative Works. However, You may modify the text of this License and copy, distribute or communicate your modified version (the "Modified License") and apply it to other original works of authorship subject to the following conditions: (i) You may not indicate in any way that your Modified License is the "Academic Free License" or "AFL" and you may not use those names in the name of your Modified License; (ii) You must replace the notice specified in the first paragraph above with a notice of your own that is not confusingly similar to the notice in this License; and (iii) You may not claim that your original works are open source software unless your Modified License has been approved by Open Source Initiative (OSI) and You comply with its license review and certification process.

9.14.2 License Summary

The Academic Free License (AFL) v. 3.0 is a copyleft-style license that:

Permits: - □ Commercial use - □ Distribution - □ Modification - □ Patent use - □ Private use

Requires: - □ License and copyright notice - □ Source code disclosure for external deployment - □ Attribution notices

Prohibits: - □ Trademark use - □ Liability claims - □ Warranty claims

9.14.3 Key Differences from Other Licenses

vs MIT License: - AFL requires source code disclosure for external deployment - AFL includes patent grant provisions - AFL has stronger copyleft requirements

vs GPL License: - AFL is less restrictive than GPL - AFL allows linking with proprietary software - AFL focuses on academic and research use

vs Apache License: - AFL has stronger academic focus - AFL includes external deployment clause - AFL attribution requirements differ

9.14.4 Copyright Notice

Copyright (c) 2024 Tolga BERBER, Beyzanur SİYAH

Licensed under the Academic Free License version 3.0

9.14.5 Third-Party Licenses

OptiX incorporates or depends on several third-party libraries:

OR-Tools - License: Apache License 2.0 - Copyright: Google Inc. - Website: https://developers.google.com/optimization

Gurobi (Optional) - License: Commercial (separate license required) - Copyright: Gurobi Optimization, LLC - Website: https://www.gurobi.com

Python Standard Library - License: Python Software Foundation License - Copyright: Python Software Foundation

Sphinx (Documentation) - License: BSD License - Copyright: The Sphinx developers

9.14.6 License Compatibility

The AFL v. 3.0 is compatible with:

□ **MIT License** - Can incorporate MIT-licensed code □ **BSD License** - Can incorporate BSD-licensed code □ **Apache 2.0** - Can incorporate Apache-licensed code □ **GPL** - Limited compatibility, consult legal advice □ **Proprietary** - Cannot incorporate proprietary code without permission

9.14.7 For Academic Use

OptiX is specifically designed for academic and research use:

Academic Benefits: - Free for research and educational purposes - Source code available for study and modification - Can be used in academic publications - Suitable for thesis and dissertation projects

Commercial Academic Use: - Universities can use for commercial research - Technology transfer offices can license derivatives - Spin-off companies can use with proper attribution

Citation Requirements: When using OptiX in academic work, please cite:

```
@software{optix2024,
  title={OptiX: Mathematical Optimization Framework},
  author={Berber, Tolga and Siyah, Beyzanur},
  year={2024},
  url={https://github.com/yourusername/optix},
  version={1.0.0}
}
```

9.14.8 For Commercial Use

Commercial use of OptiX is permitted under AFL v. 3.0:

Requirements: - Include license and copyright notices - Provide source code for external deployments - Maintain attribution notices - Comply with patent grant terms

Recommendations: - Consult legal counsel for complex commercial deployments - Consider commercial support options - Evaluate solver licensing (especially Gurobi) - Plan for source code disclosure requirements

Commercial Support: - Custom development services available - Integration consulting - Training and workshops - Priority support options

9.14.9 License Enforcement

Compliance Monitoring: - Regular license audits of derivative works - Community reporting of violations - Automated license checking tools

Violation Response: - Educational outreach for unintentional violations - Formal notice and cure period - Legal action as last resort

Compliance Resources: - License compliance checklist - Legal FAQ document - Community support forums

9.14.10 Getting Legal Help

When to Consult Lawyers: - Complex commercial deployments - Questions about derivative work licensing - International licensing considerations - Patent-related concerns

Resources: - Open Source Initiative (OSI) - Software Freedom Law Center - University technology transfer offices - IP attorneys with open source expertise

Common Questions: For frequently asked legal questions, see our *Legal FAQ* (page **??**) (coming soon).

9.14.11 Contact Information

For license-related questions:

Academic Inquiries: - Email: tolga.berber@fen.ktu.edu.tr - Institution: Karadeniz Technical University

Commercial Inquiries: - Email: contact@optix-framework.org - Business development and partnerships

Legal Issues: - Email: legal@optix-framework.org - License compliance and legal matters

Note

This license summary is provided for convenience only and does not constitute legal advice. The full license text above is the authoritative version.

Warning

Always consult with qualified legal counsel for specific licensing questions and commercial use planning.

Chapter 10

Indices and Tables

- genindex
- modindex
- search

10.1 Project Information

License: Academic Free License (AFL) v. 3.0

Authors:

- **Tolga BERBER** - Lead Developer & Project Architect
  - Email: tolga.berber@fen.ktu.edu.tr
  - Institution: Karadeniz Technical University, Computer Science Department
- **Beyzanur SİYAH** - Core Developer & Research Assistant
  - Email: beyzanursiyah@ktu.edu.tr
  - Institution: Karadeniz Technical University

Academic References:

1. Stigler, G. J. (1945). "The Cost of Subsistence". Journal of Farm Economics
2. Charnes, A., & Cooper, W. W. (1961). "Management Models and Industrial Applications of Linear Programming"
3. Dantzig, G. B. (1963). "Linear Programming and Extensions"

Note

OptiX is under active development. For the latest updates and releases, visit our GitHub repository[514].

[514] https://github.com/yourusername/optix

Tip

New to optimization? Start with our *Quick Start Guide* (page 24) guide and explore the *Classic Diet Problem* (page 34) for a practical introduction to linear programming.

Important

Ready to optimize? Start with our *Examples* (page 32) or dive into the *Problem Module* (page 81) documentation!

Python Module Index

Index