Pipeline-Sim Documentation Suite

Table of Contents

User Guide

Getting Started with Pipeline-Sim

Installation

System Requirements

- **Operating System**: Linux, macOS, or Windows 10+
- **Memory**: 8 GB RAM minimum (16 GB recommended for large networks)
- **Disk Space**: 2 GB for installation
- **Processor**: Multi-core CPU recommended for parallel computing

Quick Installation

```bash
# Using pip (Python package)
pip install pipeline-sim

# From source
git clone https://github.com/pipeline-sim/pipeline-sim.git
cd pipeline-sim
mkdir build && cd build
cmake ..
make -j$(nproc)
sudo make install
cd ../python
pip install -e .
```

Your First Simulation

1. Create a Simple Network

python

```python
import pipeline_sim as ps

# Create network
network = ps.Network()

# Add nodes
source = network.add_node("source", ps.NodeType.SOURCE)
sink = network.add_node("sink", ps.NodeType.SINK)

# Add pipe
pipe = network.add_pipe("pipe1", source, sink,
                       length=1000,    # meters
                       diameter=0.3)   # meters

# Set boundary conditions
network.set_pressure(source, 50e5)    # 50 bar
network.set_flow_rate(sink, 0.1)      # 0.1 m³/s
```

2. Define Fluid Properties

python

```python
# Create fluid
fluid = ps.FluidProperties()
fluid.oil_density = 850       # kg/m³
fluid.oil_viscosity = 0.01    # Pa.s
fluid.oil_fraction = 1.0      # Single phase
```

3. Run Simulation

python

```python
# Create solver
solver = ps.SteadyStateSolver(network, fluid)

# Run simulation
results = solver.solve()

# Check results
if results.converged:
    print(f"Outlet pressure: {results.node_pressures['sink']/1e5:.1f} bar")
    print(f"Pressure drop: {results.pressure_drop(pipe)/1e5:.1f} bar")
```

Working with Network Files

JSON Format

json

```json
{
  "nodes": [
    {
      "id": "wellhead",
      "type": "SOURCE",
      "pressure": 7000000,
      "elevation": -1500
    },
    {
      "id": "platform",
      "type": "SINK",
      "flow_rate": 0.2,
      "elevation": 0
    }
  ],
  "pipes": [
    {
      "id": "riser",
      "upstream": "wellhead",
      "downstream": "platform",
      "length": 1500,
      "diameter": 0.3,
      "roughness": 0.000045,
      "inclination": 1.5708
    }
  ],
  "fluid": {
    "oil_density": 820,
    "gas_density": 0.8,
    "water_density": 1025,
    "gas_oil_ratio": 100,
    "water_cut": 0.1
```

```
  }
}
```

Loading Networks

```python
# Load from file
network, fluid = ps.load_network("network.json")

# Save network
network.save_to_json("modified_network.json")
```

Visualization

Network Topology

```python
import matplotlib.pyplot as plt

# Plot network
fig = ps.plot_network(network, results)
plt.show()
```

Pressure Profiles

```python
# Extract data along path
path = ["source", "pump", "valve", "sink"]
distances = [0, 1000, 2000, 3000]
pressures = [results.node_pressures[n]/1e5 for n in path]

plt.plot(distances, pressures, 'b-o')
plt.xlabel('Distance (m)')
plt.ylabel('Pressure (bar)')
plt.title('Pressure Profile')
plt.grid(True)
plt.show()
```

Advanced Features

Multiphase Flow

```python
# Configure multiphase fluid
fluid = ps.FluidProperties()
fluid.oil_density = 850
fluid.gas_density = 0.85
fluid.water_density = 1025

# Set phase fractions
fluid.gas_oil_ratio = 150    # sm³/sm³
fluid.water_cut = 0.3        # 30% water

# Calculate phase fractions
fluid.water_fraction = 0.3
fluid.oil_fraction = 0.6
fluid.gas_fraction = 0.1

# Select correlation
solver.set_correlation("Beggs-Brill")
```

Transient Simulation

```python
# Create transient solver
transient = ps.TransientSolver(network, fluid)
transient.set_time_step(0.1)          # seconds
transient.set_simulation_time(300)     # 5 minutes

# Add events
valve_closure = ps.ValveClosureEvent(
    "valve1",
    start_time=30,    # seconds
    duration=10       # seconds
)
transient.add_event(valve_closure)

# Run simulation
results = transient.solve()

# Get time history
history = transient.get_time_history()
```

Equipment Models

```python
# Add pump
pump = network.add_equipment("pump1", ps.CentrifugalPump)
pump.set_curve_coefficients(a=150, b=0, c=1000)  # H = a - b*Q - c*Q²
pump.set_speed_ratio(0.9)  # 90% of rated speed

# Add valve
valve = network.add_equipment("valve1", ps.ControlValve)
valve.set_cv(200)         # Valve coefficient
valve.set_opening(0.7)    # 70% open

# Add separator
separator = network.add_equipment("sep1", ps.Separator)
separator.set_separation_efficiency(0.99, 0.95)  # Gas, liquid
```

Troubleshooting

Common Issues

1. **Convergence Problems**
   - Check boundary conditions (over/under-specified)
   - Verify fluid properties are physical
   - Increase relaxation factor
   - Check for disconnected nodes
2. **Slow Performance**
   - Reduce tolerance for faster (less accurate) results
   - Enable parallel computing
   - Simplify network where possible
3. **Unrealistic Results**

- Verify unit consistency
- Check pipe inclinations (radians)
- Ensure roughness values are reasonable

Debug Mode

```python
# Enable verbose output
solver.config.verbose = True

# Check network validity
ps.validate_network(network)

# Print detailed results
ps.print_results_summary(results)
```

Theory Manual

Governing Equations

Conservation of Mass

For each node in the network:

$$\sum_i Q_{in,i} - \sum_j Q_{out,j} + Q_{external} = 0$$

Where:

- $Q_{in,i}$ = Inflow from pipe i
- $Q_{out,j}$ = Outflow to pipe j

- $Q_{external}$ = External flow (source/sink)

Conservation of Momentum

For steady-state flow in pipes:

$$\frac{\partial P}{\partial x} = -\frac{f \rho v^2}{2D} - \rho g \sin \theta$$

Where:

- P = Pressure (Pa)
- f = Friction factor
- ρ = Fluid density (kg/m^3)
- v = Velocity (m/s)
- D = Diameter (m)
- g = Gravitational acceleration (9.81 m/s^2)
- θ = Inclination angle (radians)

Friction Factor

Laminar Flow (Re < 2300)

$$f = \frac{64}{Re}$$

Turbulent Flow (Re ≥ 2300)

Colebrook-White equation: $\frac{1}{\sqrt{f}} = -2 \log_{10} \left(\frac{\epsilon/D}{3.7} + \frac{2.51}{Re\sqrt{f}} \right)$

Where:

- $Re = \frac{\rho v D}{\mu}$ = Reynolds number
- ϵ = Pipe roughness (m)

- μ = Dynamic viscosity (Pa·s)

Multiphase Flow Correlations

Beggs-Brill Correlation

Flow Pattern Determination

The correlation identifies four flow patterns:

1. Segregated
2. Intermittent
3. Distributed
4. Annular

Flow pattern boundaries:

- $L_1 = 316 \lambda^{0.302}$
- $L_2 = 0.0009252 \lambda^{-2.4684}$
- $L_3 = 0.10 \lambda^{-1.4516}$
- $L_4 = 0.5 \lambda^{-6.738}$

Where $\lambda = v_{sl}/(v_{sl} + v_{sg})$ is the no-slip liquid holdup.

Liquid Holdup

Horizontal flow: $H_L(0) = a\lambda^b/Fr^c$

Where coefficients a, b, c depend on flow pattern.

Inclined flow correction: $H_L(\theta) = H_L(0) \times \psi \, \psi = 1 + C[\sin(1.8\theta) - 0.333\sin^3(1.8\theta)]$

Pressure Gradient

Total pressure gradient: $\frac{dP}{dx} = \left(\frac{dP}{dx}\right)_{friction} + \left(\frac{dP}{dx}\right)_{gravity}$

Friction component: $\left(\frac{dP}{dx}\right)_{friction} = \frac{2 f_{tp} \rho_m v_m^2}{D}$

Where two-phase friction factor: $f_{tp} = f_{ns} e^S$

Hagedorn-Brown Correlation

Specialized for vertical wells:

1. Calculate CNL number
2. Determine flow pattern from Griffith-Wallis map
3. Calculate liquid holdup from correlation charts
4. Apply Hagedorn-Brown pressure gradient equation

Mechanistic Models

Physics-based approach:

1. Determine flow pattern from closure relationships
2. Solve momentum equations for each phase
3. Apply interfacial closure laws

Numerical Methods

Steady-State Solution

The system forms a set of nonlinear equations:

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}$$

Where $\mathbf{x}$ contains pressures and flow rates.

Newton-Raphson iteration: $\mathbf{J}^{(k)}\Delta\mathbf{x}^{(k)} = -\mathbf{F}(\mathbf{x}^{(k)})\ \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha\Delta\mathbf{x}^{(k)}$

Where:

- $\mathbf{J}$ = Jacobian matrix
- α = Relaxation factor ($0 < \alpha \leq 1$)

Transient Solution

Method of Characteristics

For water hammer analysis:

Characteristic equations: $\frac{dP}{dt} \pm \rho a \frac{dV}{dt} = 0$

Along characteristics: $\frac{dx}{dt} = \pm a$

Where a = wave speed: $a = \sqrt{\frac{K/\rho}{1+(K/E)(D/e)}}$

Time Integration

Implicit Euler: $\frac{\mathbf{x}^{n+1}-\mathbf{x}^{n}}{\Delta t} = \mathbf{f}(\mathbf{x}^{n+1})$

Crank-Nicolson: $\frac{\mathbf{x}^{n+1}-\mathbf{x}^{n}}{\Delta t} = \frac{1}{2}[\mathbf{f}(\mathbf{x}^{n+1}) + \mathbf{f}(\mathbf{x}^{n})]$

Equipment Models

Centrifugal Pumps

Head-flow relationship: $H = H_0 - AQ - BQ^2$

Power consumption: $P = \frac{\rho g Q H}{\eta}$

Affinity laws:

- $Q_2/Q_1 = N_2/N_1$

- $H_2/H_1 = (N_2/N_1)^2$
- $P_2/P_1 = (N_2/N_1)^3$

Control Valves

Flow equation: $Q = C_v\sqrt{\frac{\Delta P}{SG}}$

Where:

- C_v = Valve coefficient
- SG = Specific gravity

Effective C_v:

- Linear: $C_v^* = f \cdot C_v$
- Equal percentage: $C_v^* = C_v \cdot R^{(f-1)}$

Compressors

Polytropic process: $\frac{T_2}{T_1} = \left(\frac{P_2}{P_1}\right)^{\frac{n-1}{n}}$

Where polytropic exponent: $n = \frac{k}{\eta_p(k-1)+1}$

Power requirement: $P = \frac{\dot{m}nRT_1}{n-1}\left[\left(\frac{P_2}{P_1}\right)^{\frac{n-1}{n}} - 1\right]$

API Reference

Core Classes

Network

```cpp
class Network {
public:
    // Node management
    Ptr<Node> add_node(const std::string& id, NodeType type);
    Ptr<Node> get_node(const std::string& id) const;

    // Pipe management
    Ptr<Pipe> add_pipe(const std::string& id,
                       Ptr<Node> upstream,
                       Ptr<Node> downstream,
                       Real length,
                       Real diameter);

    // Boundary conditions
    void set_pressure(const Ptr<Node>& node, Real pressure);
    void set_flow_rate(const Ptr<Node>& node, Real flow_rate);

    // Serialization
    void load_from_json(const std::string& filename);
    void save_to_json(const std::string& filename) const;
};
```

Node

```cpp
class Node {
public:
    // Properties
    const std::string& id() const;
    NodeType type() const;
    Real pressure() const;
    Real temperature() const;
    Real elevation() const;

    // Setters
    void set_pressure(Real p);
    void set_temperature(Real t);
    void set_elevation(Real e);
};

enum class NodeType {
    SOURCE,
    SINK,
    JUNCTION,
    PUMP,
    COMPRESSOR,
    VALVE,
    SEPARATOR
};
```

Pipe

```cpp
class Pipe {
public:
    // Geometry
    Real length() const;
    Real diameter() const;
    Real area() const;
    Real volume() const;

    // Properties
    Real roughness() const;
    Real inclination() const;

    // Flow state
    Real flow_rate() const;
    Real velocity() const;
    Real reynolds_number(Real viscosity, Real density) const;
};
```

FluidProperties

cpp

```cpp
struct FluidProperties {
    // Densities (kg/m³)
    Real oil_density;
    Real gas_density;     // Relative to air
    Real water_density;

    // Viscosities (Pa·s)
    Real oil_viscosity;
    Real gas_viscosity;
    Real water_viscosity;

    // Phase fractions
    Real oil_fraction;
    Real gas_fraction;
    Real water_fraction;

    // PVT properties
    Real gas_oil_ratio;   // sm³/sm³
    Real water_cut;       // fraction

    // Methods
    Real mixture_density() const;
    Real mixture_viscosity() const;
};
```

Solver

```cpp
class SteadyStateSolver : public Solver {
public:
    SteadyStateSolver(Ptr<Network> network,
                      const FluidProperties& fluid);

    SolutionResults solve() override;

    // Configuration
    SolverConfig& config();
};

struct SolverConfig {
    Real tolerance = 1e-6;
    int max_iterations = 1000;
    Real relaxation_factor = 0.7;
    bool verbose = false;
    bool use_parallel = true;
};

struct SolutionResults {
    bool converged;
    int iterations;
    Real residual;
    Real computation_time;

    std::map<std::string, Real> node_pressures;
    std::map<std::string, Real> pipe_flow_rates;
    std::map<std::string, Real> pipe_pressure_drops;
};
```

Python API

Basic Usage

python

```python
import pipeline_sim as ps

# Create network
network = ps.Network()

# Add components
node = network.add_node(id, type)
pipe = network.add_pipe(id, upstream, downstream, length, diameter)

# Set properties
node.pressure = value
pipe.roughness = value

# Solve
solver = ps.SteadyStateSolver(network, fluid)
results = solver.solve()
```

Utility Functions

```python
# I/O
network, fluid = ps.load_network(filename)
ps.save_results(results, filename)

# Visualization
fig = ps.plot_network(network, results)
ps.generate_report(network, results, fluid, filename)

# Validation
errors = ps.validate_network(network)
```

Advanced Features

```python
# Correlations
from pipeline_sim.correlations import BeggsBrill

result = BeggsBrill.calculate(fluid, pipe, flow_rate)
print(f"Pressure gradient: {result.pressure_gradient} Pa/m")
print(f"Flow pattern: {result.flow_pattern_name}")

# ML Integration
from pipeline_sim.ml_integration import DigitalTwin

twin = DigitalTwin()
twin.initialize(network, fluid)
twin.update_with_measurements(pressures, flows, timestamp)
state = twin.estimate_state()
anomalies = twin.detect_discrepancies()
```

Plugin Development Guide

Creating a Custom Correlation

C++ Implementation

```cpp
#include "pipeline_sim/correlations.h"

class MyCorrelation : public FlowCorrelation {
public:
    Results calculate(
        const FluidProperties& fluid,
        const Pipe& pipe,
        Real flow_rate,
        Real inlet_pressure,
        Real inlet_temperature
    ) const override {
        Results results;

        // Your correlation logic here
        results.pressure_gradient = /* calculated value */;
        results.liquid_holdup = /* calculated value */;

        return results;
    }

    std::string name() const override {
        return "My Custom Correlation";
    }
};

// Register correlation
extern "C" FlowCorrelation* create_correlation() {
    return new MyCorrelation();
}
```

Python Plugin

python

```python
from pipeline_sim import FlowCorrelation, FlowPattern

class MyPythonCorrelation(FlowCorrelation):
    def calculate(self, fluid, pipe, flow_rate,
                 inlet_pressure, inlet_temperature):
        # Calculate pressure gradient
        velocity = flow_rate / pipe.area()
        reynolds = pipe.reynolds_number(
            fluid.mixture_viscosity(),
            fluid.mixture_density()
        )

        # Your correlation logic
        pressure_gradient = self._my_calculation(...)

        return {
            'pressure_gradient': pressure_gradient,
            'liquid_holdup': 0.8,
            'flow_pattern': FlowPattern.INTERMITTENT
        }

    def name(self):
        return "My Python Correlation"

# Register
ps.register_correlation(MyPythonCorrelation())
```

Creating Equipment Models

cpp

```cpp
class CustomValve : public Equipment {
public:
    CustomValve(const std::string& id)
        : Equipment(id, NodeType::VALVE) {}

    void calculate(
        Real inlet_pressure,
        Real inlet_temperature,
        Real flow_rate,
        Real& outlet_pressure,
        Real& outlet_temperature
    ) override {
        // Custom valve model
        Real dp = calculate_pressure_drop(flow_rate);
        outlet_pressure = inlet_pressure - dp;
        outlet_temperature = inlet_temperature;
    }

private:
    Real calculate_pressure_drop(Real flow) {
        // Your model here
        return k_ * flow * flow;
    }

    Real k_{1000.0};  // Resistance coefficient
};
```

Best Practices

Network Design

1. Topology Guidelines

- **Avoid Redundant Nodes**: Merge nodes that are very close
- **Consistent Naming**: Use descriptive, systematic names
- **Elevation Data**: Always specify node elevations for gravity calculations

2. Pipe Specifications

- **Roughness Values**:
  - New steel: 0.045 mm
  - Commercial steel: 0.045-0.09 mm
  - Rusty steel: 0.15-4 mm
  - Concrete: 0.3-3 mm
- **Diameter Selection**: Consider velocity limits (typically 3-10 m/s)

3. Boundary Conditions

- **Well-Posed Problems**:
  - One pressure BC per disconnected region
  - Flow BCs for remaining degrees of freedom
  - Don't over-constrain

Performance Optimization

1. Large Networks

python

```python
# Enable parallel computing
solver.config.use_parallel = True
solver.config.num_threads = 8

# Adjust tolerance for speed
solver.config.tolerance = 1e-4  # Less strict

# Use sparse matrix optimizations
solver.config.use_sparse = True
```

2. Transient Simulations

python

```python
# Adaptive time stepping
transient.set_adaptive_timestep(True)
transient.set_cfl_limit(0.9)

# Output only what's needed
transient.set_output_variables(['pressure', 'flow'])
transient.set_output_nodes(['critical_junction'])
```

3. Memory Management

```python
# Process results in chunks
for chunk in ps.iterate_results_chunks(results, chunk_size=1000):
    process_chunk(chunk)

# Clear unnecessary data
network.clear_results_cache()
```

Validation and Testing

1. Unit Testing

```python
import pytest

def test_pressure_drop():
    # Create simple test case
    network = create_test_network()
    results = solve_network(network)

    # Verify against analytical solution
    analytical_dp = calculate_analytical_dp()
    assert abs(results.pressure_drop - analytical_dp) < 0.01
```

2. Benchmarking

```python
import time

def benchmark_solver(network_size):
    network = create_network(size=network_size)

    start = time.time()
    results = solver.solve()
    elapsed = time.time() - start

    return {
        'size': network_size,
        'time': elapsed,
        'iterations': results.iterations
    }
```

3. Validation Checklist

- [ ] Mass balance at all junctions
- [ ] Pressure monotonically decreasing along flow
- [ ] Velocities within reasonable range
- [ ] No negative pressures
- [ ] Reynolds numbers physical

Common Pitfalls

1. Unit Inconsistency

Always use SI units internally:

- Pressure: Pa (not bar, psi)
- Length: m (not km, ft)

- Flow: m^3/s (not m^3/day, bbl/day)

2. Numerical Issues

```python
# Avoid division by zero
velocity = flow / max(pipe.area(), 1e-10)

# Handle small Reynolds numbers
if reynolds < 1:
    friction = 64.0  # Laminar limit
else:
    friction = calculate_friction(reynolds)
```

3. Convergence Problems

```python
# Start with good initial guess
solver.set_initial_guess(previous_results)

# Use continuation method for difficult problems
for pressure in np.linspace(low_pressure, high_pressure, 10):
    network.set_pressure(source, pressure)
    results = solver.solve()
    solver.set_initial_guess(results)
```

Reporting

1. Essential Outputs

- Executive summary with key metrics

- Pressure and flow tables at critical points
- Visualization of bottlenecks
- Flow assurance warnings

2. Visualization Standards

python

```
# Consistent color schemes
pressure_cmap = 'viridis'
flow_cmap = 'coolwarm'
anomaly_cmap = 'Reds'

# Clear labeling
plt.xlabel('Distance (km)', fontsize=12)
plt.ylabel('Pressure (bar)', fontsize=12)
plt.title('Pipeline Pressure Profile', fontsize=14, fontweight='bold')

# Include units and timestamps
plt.text(0.02, 0.98, f'Generated: {datetime.now()}',
         transform=ax.transAxes, verticalalignment='top')
```

3. Documentation

- Document all assumptions
- Include fluid property sources
- Note correlation selections
- Specify boundary condition rationale

Conclusion

Pipeline-Sim provides a comprehensive framework for petroleum pipeline simulation with:

- **Accuracy**: Industry-standard correlations and rigorous physics
- **Performance**: Optimized algorithms and parallel computing
- **Extensibility**: Plugin architecture for custom models
- **Usability**: Intuitive Python API and comprehensive documentation

For additional support:

- GitHub: https://github.com/pipeline-sim/pipeline-sim
- Documentation: https://pipeline-sim.readthedocs.io
- Community Forum: https://forum.pipeline-sim.org

Generated with AI assistance - Pipeline-Sim v0.1.0