

pyfuzzy-toolbox

Professional Fuzzy Systems for Python





Moiseis Cecconello

Copyright © 2025 Moiseis Cecconello

Table of Contents

1. pyfuzzy-toolbox Documentation	5
1.1 Features	5
1.2 Quick Links	5
1.3 Installation	5
1.4 Quick Example	5
1.5 Community & Support	6
1.6 Citation	6
1.7 License	6
2. Installation	7
2.1 Requirements	7
2.2 Install from PyPI	7
2.3 Install from Source	7
2.4 Verify Installation	7
2.5 Import Convention	7
2.6 Next Steps	7
3. Quickstart Guide	8
3.1 Installation	8
3.2 Your First Fuzzy System	8
3.3 Complete Example	8
3.4 Visualize Your System	8
3.5 Test Multiple Values	9
3.6 Next Steps	9
3.7 Common Patterns	9
3.8 Help & Support	9
4. User Guide: Fundamentals	10
4.1 What is Fuzzy Logic?	10
4.2 Membership Functions	10
4.3 Fuzzy Sets	11
4.4 Linguistic Variables	11
4.5 Fuzzy Operators	11
4.6 Practical Example: Thermal Comfort	12
4.7 Common Patterns	12
4.8 Tips and Best Practices	13
4.9 Troubleshooting	13
4.10 Next Steps	14





4.11 Further Reading	14
5. User Guide: Inference Systems	15
5.1 What is a Fuzzy Inference System?	15
5.2 Mamdani vs Sugeno	15
5.3 Mamdani Systems	15
5.4 Sugeno Systems	17
5.5 Advanced Topics	17
5.6 Design Guidelines	18
5.7 Troubleshooting	18
5.8 Next Steps	19
5.9 Further Reading	19
6. User Guide: Learning Systems	20
6.1 Why Learn Fuzzy Systems from Data?	20
6.2 Overview of Learning Methods	20
6.3 Wang-Mendel Algorithm	20
6.4 ANFIS (Adaptive Neuro-Fuzzy Inference System)	21
6.5 Mamdani Learning with Metaheuristics	22
6.6 Choosing a Learning Method	24
6.7 Advanced Topics	24
6.8 Design Guidelines	25
6.9 Troubleshooting	26
6.10 Next Steps	26
6.11 Further Reading	27
7. User Guide: Fuzzy Dynamical Systems	28
7.1 What are Fuzzy Dynamical Systems?	28
7.2 Overview of Methods	28
7.3 Fuzzy Numbers	28
7.4 Fuzzy ODE Solver	29
7.5 p-Fuzzy Systems (Discrete-Time)	30
7.6 p-Fuzzy Systems (Continuous-Time)	31
7.7 Comparing Methods	32
7.8 Design Guidelines	33
7.9 Advanced Topics	34
7.10 Troubleshooting	35
7.11 Next Steps	35
7.12 Further Reading	35
8. Core API Reference	36
8.1 Membership Functions	36

8.2	Classes	37
8.3	Fuzzy Operators	38
8.4	Defuzzification	38
8.5	Complete Example	39
8.6	See Also	39
9.	Inference API Reference	40
9.1	MamdaniSystem	40
9.2	SugenoSystem	42
9.3	Complete Examples	43
9.4	See Also	44
10.	Learning API Reference	45
10.1	WangMendelLearning	45
10.2	ANFIS	47
10.3	MamdaniLearning	48
10.4	Metaheuristics	49
10.5	Comparison Table	50
10.6	See Also	50
11.	Dynamics API Reference	51
11.1	Fuzzy ODEs	51
11.2	p-Fuzzy Systems	53
11.3	Comparison: Fuzzy ODE vs p-Fuzzy	55
11.4	See Also	55
12.	Examples Gallery	56
12.1	 Fundamentals (Beginner)	56
12.2	 Inference Systems (Intermediate)	56
12.3	 Learning & Optimization (Advanced)	56
12.4	 Dynamic Systems (Advanced)	57
12.5	By Difficulty Level	58
12.6	Running the Examples	58
12.7	Need Help?	58

1. pyfuzzy-toolbox Documentation

Welcome to **pyfuzzy-toolbox**, a comprehensive Python library for Fuzzy Systems with focus on education and professional applications.

1.1 Features

-  **Core:** Membership functions, fuzzy sets, linguistic variables, operators
-  **Inference:** Mamdani and Sugeno/TSK systems
-  **Learning:** ANFIS, Wang-Mendel, metaheuristic optimization (PSO, DE, GA)
-  **Dynamics:** Fuzzy ODEs and p-fuzzy systems

1.2 Quick Links

Getting Started

Install pyfuzzy-toolbox and create your first fuzzy system in 5 minutes

→ [Installation](#) → [Quickstart](#)

User Guide

Learn how to use fuzzy systems to solve real-world problems

→ [Fundamentals](#) → [Inference Systems](#)

API Reference

Complete reference for all classes and methods

→ [Core API](#) → [Inference API](#)

Examples

Gallery of Colab notebooks with practical examples

→ [Examples Gallery](#)

1.3 Installation

```
pip install pyfuzzy-toolbox
```

1.4 Quick Example

```
import fuzzy_systems as fs

# Create Mamdani system
system = fs.MamdaniSystem()
system.add_input('temperature', (0, 40))
system.add_output('fan_speed', (0, 100))
```

```
# Add terms
system.add_term('temperature', 'cold', 'triangular', (0, 0, 20))
system.add_term('temperature', 'hot', 'triangular', (20, 40, 40))
system.add_term('fan_speed', 'slow', 'triangular', (0, 0, 50))
system.add_term('fan_speed', 'fast', 'triangular', (50, 100, 100))

# Add rules
system.add_rules([('cold', 'slow'), ('hot', 'fast')])

# Evaluate
result = system.evaluate(temperature=25)
print(f"Fan speed: {result['fan_speed']:.1f}%")
```

1.5 Community & Support

- **PyPI:** pypi.org/project/pyfuzzy-toolbox
- **GitHub:** github.com/lmoi6/pyfuzzy-toolbox
- **Issues:** [Report bugs or request features](#)

1.6 Citation

```
@software{pyfuzzy_toolbox,
  title = {pyfuzzy-toolbox: A Comprehensive Python Library for Fuzzy Systems},
  author = {Cecconello, Moiseis},
  year = {2025},
  url = {https://github.com/lmoi6/pyfuzzy-toolbox}
}
```

1.7 License

MIT License - see [LICENSE](#) for details.

2. Installation

2.1 Requirements

- Python 3.8 or higher
- pip (Python package manager)

2.2 Install from PyPI

The simplest way to install pyfuzzy-toolbox is via pip:

```
pip install pyfuzzy-toolbox
```

Optional Dependencies

Install with machine learning support (ANFIS, Wang-Mendel, optimization):

```
pip install pyfuzzy-toolbox[ml]
```

Install with development tools (testing, linting):

```
pip install pyfuzzy-toolbox[dev]
```

Install everything:

```
pip install pyfuzzy-toolbox[all]
```

2.3 Install from Source

For development or to get the latest features:

```
git clone https://github.com/lmoi6/pyfuzzy-toolbox.git
cd pyfuzzy-toolbox
pip install -e .
```

For editable install with development dependencies:

```
pip install -e .[dev]
```

2.4 Verify Installation

```
import fuzzy_systems as fs
print(f"pyfuzzy-toolbox version: {fs.__version__}")
```

2.5 Import Convention

The recommended import convention is:

```
import fuzzy_systems as fs
```

Note: The package name on PyPI is `pyfuzzy-toolbox`, but you import it as `fuzzy_systems`.

2.6 Next Steps

- [Quickstart](#): Create your first fuzzy system in 5 minutes
- [Key Concepts](#): Learn fundamental fuzzy logic concepts

3. Quickstart Guide

Get started with pyfuzzy-toolbox in 5 minutes!

3.1 Installation

```
pip install pyfuzzy-toolbox
```

3.2 Your First Fuzzy System

Let's build a simple temperature-controlled fan system: - **Input:** Temperature (0-40°C) - **Output:** Fan speed (0-100%) - **Rules:** If cold → slow, If hot → fast

Step 1: Import

```
import fuzzy_systems as fs
```

```
# Add output: fan speed
system.add_output('fan_speed', (0, 100))

# Add linguistic terms
system.add_term('fan_speed', 'slow', 'triangular', (0, 0, 50))
system.add_term('fan_speed', 'fast', 'triangular', (50, 100, 100))
```

Step 2: Create System

```
# Create Mamdani system
system = fs.MamdaniSystem()
```

Step 5: Add Rules

```
# Define fuzzy rules
system.add_rules([
    ('cold', 'slow'), # IF temperature is cold THEN fan_speed is slow
    ('hot', 'fast')   # IF temperature is hot THEN fan_speed is fast
])
```

Step 3: Define Input Variable

```
# Add input: temperature
system.add_input('temperature', (0, 40))

# Add linguistic terms
system.add_term('temperature', 'cold', 'triangular', (0, 0, 20))
system.add_term('temperature', 'hot', 'triangular', (20, 40, 40))
```

Step 6: Evaluate

```
# Test the system
result = system.evaluate(temperature=25)
print(f"Fan speed: {result['fan_speed']:.1f}%")
# Output: Fan speed: 50.0%
```

Step 4: Define Output Variable

3.3 Complete Example

```
import fuzzy_systems as fs

# Create and configure system
system = fs.MamdaniSystem()
system.add_input('temperature', (0, 40))
system.add_output('fan_speed', (0, 100))

# Add terms
system.add_term('temperature', 'cold', 'triangular', (0, 0, 20))
system.add_term('temperature', 'hot', 'triangular', (20, 40, 40))
system.add_term('fan_speed', 'slow', 'triangular', (0, 0, 50))
system.add_term('fan_speed', 'fast', 'triangular', (50, 100, 100))

# Add rules
system.add_rules([
    ('cold', 'slow'),
    ('hot', 'fast')
])

# Evaluate
result = system.evaluate(temperature=25)
print(f"Fan speed: {result['fan_speed']:.1f}%")
```

3.4 Visualize Your System

```
# Plot input variable
system.plot_variables(['temperature'])

# Plot output variable
system.plot_variables(['fan_speed'])

# Plot rule matrix
system.plot_rule_matrix()
```

3.5 Test Multiple Values

```
test_temps = [5, 15, 25, 35]

for temp in test_temps:
    result = system.evaluate(temperature=temp)
    print(f"Temperature: {temp}°C → Fan speed: {result['fan_speed']:.1f}%")
```

Output:

```
Temperature: 5°C → Fan speed: 12.5%
Temperature: 15°C → Fan speed: 37.5%
Temperature: 25°C → Fan speed: 62.5%
Temperature: 35°C → Fan speed: 87.5%
```

3.6 Next Steps

Now that you have a working fuzzy system, explore more:

- **User Guide: Fundamentals** - Learn about membership functions, fuzzification, and operators
- **User Guide: Inference** - Build complex Mamdani and Sugeno systems
- **Examples Gallery** - See practical applications in Colab notebooks
- **API Reference** - Detailed documentation of all classes and methods

3.7 Common Patterns

Adding More Terms

```
system.add_term('temperature', 'cold', 'triangular', (0, 0, 15))
system.add_term('temperature', 'warm', 'triangular', (10, 20, 30))
system.add_term('temperature', 'hot', 'triangular', (25, 40, 40))

system.add_term('fan_speed', 'slow', 'triangular', (0, 0, 40))
system.add_term('fan_speed', 'medium', 'triangular', (30, 50, 70))
system.add_term('fan_speed', 'fast', 'triangular', (60, 100, 100))
```

Adding More Rules

```
system.add_rules([
    ('cold', 'slow'),
```

```
( 'warm', 'medium'),
( 'hot', 'fast')
])
```

Multiple Inputs

```
system.add_input('humidity', (0, 100))
system.add_term('humidity', 'dry', 'triangular', (0, 0, 50))
system.add_term('humidity', 'humid', 'triangular', (50, 100, 100))

# Rules with multiple conditions
system.add_rules([
    {'temperature': 'hot', 'humidity': 'humid', 'fan_speed': 'fast'},
    {'temperature': 'cold', 'humidity': 'dry', 'fan_speed': 'slow'}
])
```

3.8 Help & Support

- **Documentation:** [Full docs](#)
- **Issues:** [Report bugs](#)
- **PyPI:** [Package page](#)

4. User Guide: Fundamentals

This guide introduces the fundamental concepts of fuzzy logic and how to use the `fuzzy_systems.core` module.

4.1 What is Fuzzy Logic?

Classical logic uses binary values: **True** or **False** (1 or 0). Fuzzy logic extends this to handle **partial truth**: values between 0 and 1.

Example: - Classical: "Is 18°C cold?" → **Yes** (1) or **No** (0) - Fuzzy: "Is 18°C cold?" → **0.4** (somewhat cold)

This allows systems to handle **uncertainty** and **gradual transitions**, making them more human-like.

4.2 Membership Functions

Membership functions (MFs) define **how much** an input belongs to a fuzzy set.

Types of Membership Functions

1. Triangular

Most common for its simplicity.

Parameters: `(a, b, c)` where `b` is the peak

```
from fuzzy_systems.core import triangular
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
mu = triangular(x, (2, 5, 8))

plt.plot(x, mu)
plt.title('Triangular MF')
plt.xlabel('x')
plt.ylabel('μ(x)')
plt.show()
```

When to use: - Simple concepts with clear peaks - Fast computation needed
- Educational purposes

2. Trapezoidal

Has a **plateau** where $\mu = 1$.

Parameters: `(a, b, c, d)` where `[b, c]` is the plateau

```
from fuzzy_systems.core import trapezoidal

mu = trapezoidal(x, (1, 3, 7, 9))
```

When to use: - Ranges that are "fully true" (e.g., "room temperature" = 20-24°C) - Modeling endpoints (e.g., "very low" includes everything below 5)

3. Gaussian

Smooth, bell-shaped curve.

Parameters: `(mean, sigma)` where `sigma` controls width

```
from fuzzy_systems.core import gaussian

mu = gaussian(x, (5, 1.5))
```

When to use: - Natural phenomena (measurements, sensors) - Smooth transitions needed - Mathematical modeling

4. Sigmoid

S-shaped curve, asymmetric.

Parameters: `(slope, inflection_point)`

```
from fuzzy_systems.core import sigmoid

mu = sigmoid(x, (1, 5))
```

When to use: - Asymmetric concepts (e.g., "increasing", "above threshold")
- Modeling saturation effects

Choosing the Right MF

Type	Speed	Smoothness	Best For
Triangular	⚡⚡⚡	★	Simple logic, fast computation
Trapezoidal	⚡⚡⚡	★	Ranges with "fully true" states
Gaussian	⚡⚡	★★★★	Natural phenomena, smooth transitions
Sigmoid	⚡⚡	★★★	Asymmetric concepts, saturation effects

Rule of thumb: Start with **triangular**, switch to **gaussian** if you need smoothness.

4.3 Fuzzy Sets

A **FuzzySet** combines a name with a membership function.

Creating Fuzzy Sets

```
from fuzzy_systems.core import FuzzySet

# Create a fuzzy set for "comfortable temperature"
comfortable = FuzzySet(
    name="comfortable",
    mf_type="triangular",
    params=(18, 22, 26)
)

# Calculate membership
temp = 20
mu = comfortable.membership(temp)
print(f"20°C is {mu:.2f} comfortable") # 0.50 comfortable
```

Custom Membership Functions

```
def custom_mf(x):
    """Custom bell-shaped function."""
    return np.exp(-(x - 5)**2 / 8)

custom_set = FuzzySet(
    name="custom",
    mf_type="custom",
    params=(),
    mf_func=custom_mf
)
```

4.4 Linguistic Variables

A **LinguisticVariable** groups multiple fuzzy sets under one variable.

Basic Example

```
from fuzzy_systems.core import LinguisticVariable

# Create variable
temperature = LinguisticVariable(
    name="temperature",
    universe=(0, 40)
)

# Add fuzzy terms
temperature.add_term("cold", "trapezoidal", (0, 0, 10, 18))
temperature.add_term("warm", "triangular", (15, 22, 29))
temperature.add_term("hot", "trapezoidal", (26, 32, 40, 40))
```

Key points: - Variable has a **universe** (valid range) - Each **term** is a fuzzy set - Terms can **overlap** (this is normal!)

Fuzzification

Convert a crisp value to membership degrees in all terms.

```
# Fuzzify a value
current_temp = 24
degrees = temperature.fuzzify(current_temp)

print(degrees)
# {'cold': 0.0, 'warm': 0.357, 'hot': 0.0}
```

Interpretation: 24°C is **35.7% warm** and **0% cold/hot**.

Visualizing Variables

```
temperature.plot()
```

This creates a plot showing all terms overlapping on the same axis.

4.5 Fuzzy Operators

Combine fuzzy values using AND, OR, NOT.

AND (T-norm)

Minimum is the standard:

```
from fuzzy_systems.core import fuzzy_and_min
```

```
mu_warm = 0.7
mu_humid = 0.5

comfort = fuzzy_and_min(mu_warm, mu_humid)
print(comfort) # 0.5 (takes minimum)
```

Alternative (Product):

```
from fuzzy_systems.core import fuzzy_and_product

comfort = fuzzy_and_product(0.7, 0.5)
print(comfort) # 0.35 (7 * 0.5)
```

When to use: - Use **min** for standard Mamdani systems - Use **product** for stricter combinations

OR (S-norm)

Maximum is the standard:

```
from fuzzy_systems.core import fuzzy_or_max

discomfort = fuzzy_or_max(0.3, 0.6)
print(discomfort) # 0.6 (takes maximum)
```

Alternative (Probabilistic):

```
from fuzzy_systems.core import fuzzy_or_probabilistic

result = fuzzy_or_probabilistic(0.3, 0.6)
print(result) # 0.72 (= 0.3 + 0.6 - 0.3*0.6)
```

NOT (Negation)

Complement:

```
from fuzzy_systems.core import fuzzy_not

mu_cold = 0.2
mu_not_cold = fuzzy_not(mu_cold)
print(mu_not_cold) # 0.8 (= 1 - 0.2)
```

4.6 Practical Example: Thermal Comfort

Let's build a complete example combining everything.

Step 1: Define Variables

```
from fuzzy_systems.core import LinguisticVariable

# Temperature
temp_var = LinguisticVariable("temperature", (0, 40))
temp_var.add_term("cold", "trapezoidal", (0, 0, 12, 20))
temp_var.add_term("comfortable", "triangular", (18, 24, 30))
temp_var.add_term("hot", "trapezoidal", (28, 35, 40, 40))

# Humidity
humid_var = LinguisticVariable("humidity", (0, 100))
humid_var.add_term("dry", "trapezoidal", (0, 0, 30, 50))
humid_var.add_term("normal", "triangular", (40, 60, 80))
humid_var.add_term("humid", "trapezoidal", (70, 85, 100, 100))
```

Output:

```
Temperature:
cold: 0.000
comfortable: 0.667
hot: 0.000

Humidity:
dry: 0.000
normal: 0.750
humid: 0.000
```

Step 2: Fuzzify Inputs

```
current_temp = 26
current_humidity = 65

temp_degrees = temp_var.fuzzify(current_temp)
humid_degrees = humid_var.fuzzify(current_humidity)

print("Temperature:")
for term, degree in temp_degrees.items():
    print(f" {term}: {degree:.3f}")

print("\nHumidity:")
for term, degree in humid_degrees.items():
    print(f" {term}: {degree:.3f}")
```

Step 3: Apply Rules

```
from fuzzy_systems.core import fuzzy_and_min, fuzzy_or_max

# Rule 1: IF temp is comfortable AND humidity is normal THEN very comfortable
rule1 = fuzzy_and_min(temp_degrees['comfortable'], humid_degrees['normal'])
print(f"Rule 1 (very comfortable): {rule1:.3f}") # 0.667

# Rule 2: IF temp is hot OR humidity is humid THEN uncomfortable
rule2 = fuzzy_or_max(temp_degrees['hot'], humid_degrees['humid'])
print(f"Rule 2 (uncomfortable): {rule2:.3f}") # 0.000

# Rule 3: IF temp is cold THEN uncomfortable
rule3 = temp_degrees['cold']
print(f"Rule 3 (cold uncomfortable): {rule3:.3f}") # 0.000
```

Interpretation: - 26°C with 65% humidity is **66.7% very comfortable** - Not uncomfortable (0%)

4.7 Common Patterns

Pattern 1: Three-Term Variable

Standard partition for most variables:

```
var = LinguisticVariable("variable", (0, 100))
var.add_term("low", "trapezoidal", (0, 0, 20, 40))
var.add_term("medium", "triangular", (30, 50, 70))
var.add_term("high", "trapezoidal", (60, 80, 100, 100))
```

Pattern 2: Five-Term Variable

More granular control:

```
var = LinguisticVariable("variable", (0, 100))
var.add_term("very_low", "trapezoidal", (0, 0, 10, 25))
var.add_term("low", "triangular", (15, 25, 40))
var.add_term("medium", "triangular", (30, 50, 70))
var.add_term("high", "triangular", (60, 75, 85))
var.add_term("very_high", "trapezoidal", (75, 90, 100, 100))
```

Pattern 3: Asymmetric Endpoints

Use trapezoidal at boundaries:

```
var = LinguisticVariable("variable", (0, 100))
# Left endpoint: trapezoidal with flat left side
var.add_term("very_low", "trapezoidal", (0, 0, 15, 30))

# Middle: triangular
var.add_term("medium", "triangular", (25, 50, 75))

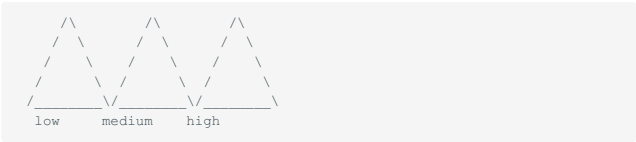
# Right endpoint: trapezoidal with flat right side
var.add_term("very_high", "trapezoidal", (70, 85, 100, 100))
```

4.8 Tips and Best Practices

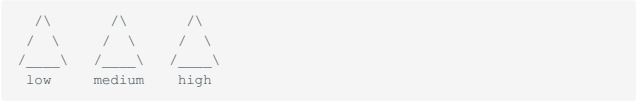
1. Overlapping is Good

Terms should **overlap** by 25-50% for smooth transitions.

Good:



Bad (no overlap):



```
# Good: covers [0, 100]
var.add_term("low", "trapezoidal", (0, 0, 30, 50))
var.add_term("high", "trapezoidal", (50, 70, 100, 100))

# Bad: gap between 50-60
var.add_term("low", "triangular", (0, 25, 50))
var.add_term("high", "triangular", (60, 80, 100))
```

3. Symmetric vs Asymmetric

- **Symmetric** (triangular/gaussian): Neutral concepts (medium, normal)
- **Asymmetric** (sigmoid/trapezoidal): Directional concepts (increasing, above)

4. Number of Terms

- **3 terms:** Simple, fast, interpretable
- **5 terms:** Good balance
- **7-9 terms:** Complex, precise (use with learning algorithms)

Rule: Start with 3, add more only if needed.

2. Universe Coverage

Make sure terms **cover the entire universe**:

4.9 Troubleshooting

Problem: "Value outside universe"

```
temperature = LinguisticVariable("temp", (0, 40))
temp_degrees = temperature.fuzzify(50) # ⚠️ Warning!
```

Solution: Extend universe or clip input:

```
value = min(max(value, 0), 40) # Clip to [0, 40]
```

Solution: Check term coverage with plots:

```
temperature.plot()
plt.axvline(x=value, color='r', linestyle='--') # Check if covered
plt.show()
```

Problem: "Membership degree is always 1"

Cause: Terms are too wide or value is exactly at a peak.

Solution: Adjust term parameters to reduce overlap.

Problem: "All membership degrees are zero"

Cause: No term covers the input value.

4.10 Next Steps

Now that you understand fuzzy logic fundamentals:

1. [Inference Systems](#) - Build complete Mamdani and Sugeno systems
 2. [API Reference: Core](#) - Detailed API documentation
 3. [Examples: Fundamentals](#) - Interactive notebooks
-

4.11 Further Reading

- **Zadeh, L.A. (1965):** "Fuzzy Sets". *Information and Control*, 8(3), 338-353.
- **Ross, T.J. (2010):** *Fuzzy Logic with Engineering Applications*. Wiley.
- **Membership Functions:** Complete API reference

5. User Guide: Inference Systems

This guide covers how to build complete fuzzy inference systems using Mamdani and Sugeno methods.

5.1 What is a Fuzzy Inference System?

A **Fuzzy Inference System (FIS)** transforms fuzzy inputs into fuzzy (or crisp) outputs through a rule base.

Components: 1. **Fuzzification:** Convert crisp inputs → fuzzy degrees 2. **Rule Base:** IF-THEN rules 3. **Inference Engine:** Apply rules 4. **Aggregation:** Combine rule outputs 5. **Defuzzification:** Convert fuzzy output → crisp value

5.2 Mamdani vs Sugeno

Feature	Mamdani	Sugeno (TSK)
Output	Linguistic fuzzy sets	Mathematical functions
Defuzzification	Centroid, MOM, etc.	Weighted average
Interpretability	★★★★ Very high	★★★ Moderate
Computation	Slower (integration)	⚡ Faster (direct)
Best for	Expert systems, control	Function approximation, modeling
Example output	"Fan speed is FAST"	"Fan speed = 0.8*temp + 10"

When to use: - **Mamdani:** You need interpretable rules with linguistic outputs - **Sugeno:** You need precise numerical modeling or faster computation

5.3 Mamdani Systems

The 5 Steps of Mamdani Inference

Let's build a temperature-controlled fan system step by step.

Step 1: Fuzzification

```
from fuzzy_systems import MamdaniSystem

# Create system
system = MamdaniSystem(name="Fan Controller")

# Add input
system.add_input('temperature', (0, 40))
system.add_term('temperature', 'cold', 'triangular', (0, 0, 20))
system.add_term('temperature', 'warm', 'triangular', (10, 20, 30))
system.add_term('temperature', 'hot', 'triangular', (20, 40, 40))

# Fuzzify (internal step when evaluating)
# For 25°C: cold=0.25, warm=0.5, hot=0.25
```

Step 2: Rule Application

```
# Add output
system.add_output('fan_speed', (0, 100))
system.add_term('fan_speed', 'slow', 'triangular', (0, 0, 50))
system.add_term('fan_speed', 'medium', 'triangular', (25, 50, 75))
system.add_term('fan_speed', 'fast', 'triangular', (50, 100, 100))
```

```
# Add rules
system.add_rules([
    ('cold', 'slow'), # IF temp is cold THEN speed is slow
    ('warm', 'medium'), # IF temp is warm THEN speed is medium
    ('hot', 'fast') # IF temp is hot THEN speed is fast
])
```

Step 3-5: Implication, Aggregation, Defuzzification

These happen automatically in `.evaluate()` :

```
result = system.evaluate(temperature=25)
print(f"Fan speed: {result['fan_speed']:.1f}%") # 50.0%
```

What happened internally: 1. **Implication:** Each rule "cuts" its output MF at the activation level 2. **Aggregation:** All cut MFs are combined (usually MAX) 3. **Defuzzification:** Center of gravity (COG) → crisp value

Building Your First Mamdani System

Complete Example: Tipping System

```
from fuzzy_systems import MamdaniSystem

# Step 1: Create system
system = MamdaniSystem(name="Tipping System")
```

```
# Step 2: Add inputs
system.add_input('service', (0, 10))
system.add_input('food', (0, 10))

# Step 3: Add input terms
for var in ['service', 'food']:
    system.add_term(var, 'poor', 'triangular', (0, 0, 5))
    system.add_term(var, 'good', 'triangular', (0, 5, 10))
    system.add_term(var, 'excellent', 'triangular', (5, 10, 10))

# Step 4: Add output
system.add_output('tip', (0, 25))

# Step 5: Add output terms
system.add_term('tip', 'low', 'triangular', (0, 0, 13))
system.add_term('tip', 'medium', 'triangular', (0, 13, 25))
system.add_term('tip', 'high', 'triangular', (13, 25, 25))

# Step 6: Add rules
system.add_rules([
    {'service': 'poor', 'food': 'poor', 'tip': 'low'},
    {'service': 'good', 'food': 'good', 'tip': 'medium'},
    {'service': 'excellent', 'food': 'excellent', 'tip': 'high'},
])

# Step 7: Evaluate
result = system.evaluate(service=7, food=8)
print(f"Tip: {result['tip']:.1f}%")
```

Rule Formats

Format 1: Dictionary (Explicit)

Most readable for complex rules:

```
system.add_rules([
    {
        'service': 'poor',
        'food': 'poor',
        'tip': 'low',
        'operator': 'AND', # Optional
        'weight': 1.0      # Optional
    }
])
```

Format 2: Tuple (Compact)

Best for simple systems:

```
# Order: (input1, input2, ..., output1, output2, ...)
system.add_rules([
    ('poor', 'poor', 'low'),
    ('good', 'good', 'medium'),
    ('excellent', 'excellent', 'high')
])
```

Format 3: Indices

Use term index instead of name:

```
# 0 = first term, 1 = second term, etc.
system.add_rules([
    (0, 0, 0), # poor, poor → low
    (1, 1, 1), # good, good → medium
    (2, 2, 2) # excellent, excellent → high
])
```

Operators in Rules

AND (default)

Both conditions must be satisfied:

```
system.add_rule({
    'temperature': 'hot',
    'humidity': 'high',
    'fan_speed': 'fast',
    'operator': 'AND' # Takes MIN of activations
})
```

OR

At least one condition must be satisfied:

```
system.add_rule({
    'temperature': 'hot',
    'humidity': 'high',
    'fan_speed': 'fast',
    'operator': 'OR' # Takes MAX of activations
})
```

Rule Weights

Reduce a rule's influence:

```
system.add_rule({
    'temperature': 'cold',
    'fan_speed': 'slow',
    'weight': 0.5 # Only 50% influence
})
```

Defuzzification Methods

Choose how to convert the fuzzy output to a crisp value:

```
system = MamdaniSystem(defuzz_method='centroid') # Default
```

Available methods:

Method	Description	When to use
'centroid'	Center of gravity	Default, balanced
'bisector'	Divides area in half	Alternative to COG
'mom'	Mean of maximum	Emphasize peak values
'som'	Smallest of maximum	Conservative choice
'lom'	Largest of maximum	Aggressive choice

Example comparison:

```
methods = ['centroid', 'bisector', 'mom', 'som', 'lom']

for method in methods:
    system = MamdaniSystem(defuzz_method=method)
    # ... configure system ...
    result = system.evaluate(temperature=25)
    print(f"{method}: {result['fan_speed']:.2f}%")
```


Visualization

Plot Variables

```
# Plot all variables
system.plot_variables()

# Plot specific variables
system.plot_variables(['temperature', 'fan_speed'])
```

Plot Rule Matrix

For 2-input systems, shows rules as a heatmap:

```
system.plot_rule_matrix()
```

Saving and Loading

```
# Save system
system.save('my_system.pkl')

# Load system
from fuzzy_systems import MamdaniSystem
system = MamdaniSystem.load('my_system.pkl')

# Export rules only
system.export_rules('rules.json', format='json')
system.export_rules('rules.txt', format='txt')

# Import rules
system.import_rules('rules.json', format='json')
```

5.4 Sugeno Systems

Zero-Order Sugeno

Outputs are **constants**.

```
from fuzzy_systems import SugenoSystem

# Create system
system = SugenoSystem()

# Add input
system.add_input('x', (0, 10))
system.add_term('x', 'low', 'triangular', (0, 0, 5))
system.add_term('x', 'medium', 'triangular', (0, 5, 10))
system.add_term('x', 'high', 'triangular', (5, 10, 10))

# Add output (order 0 = constant)
system.add_output('y', order=0)

# Add rules with constant outputs
system.add_rules([
    ('low', 2.0),      # IF x is low THEN y = 2.0
    ('medium', 5.0),   # IF x is medium THEN y = 5.0
    ('high', 8.0)      # IF x is high THEN y = 8.0
])

# Evaluate
result = system.evaluate(x=6)
print(f"y = {result['y']:.2f}")
```

How it works: 1. Fuzzify input: $x=6 \rightarrow \text{low}=0, \text{medium}=0.8, \text{high}=0.2$ 2. Apply rules: $y_1=2.0$ ($w_1=0$), $y_2=5.0$ ($w_2=0.8$), $y_3=8.0$ ($w_3=0.2$) 3. Weighted average: $y = (0 \times 2 + 0.8 \times 5 + 0.2 \times 8) / (0 + 0.8 + 0.2) = 5.6$

```
system = SugenoSystem()

# Add inputs
system.add_input('x1', (0, 10))
system.add_input('x2', (0, 10))

# Add terms
for var in ['x1', 'x2']:
    system.add_term(var, 'low', 'triangular', (0, 0, 5))
    system.add_term(var, 'high', 'triangular', (5, 10, 10))

# Add output (order 1 = linear function)
system.add_output('y', order=1)

# Rules: y = a*x1 + b*x2 + c
system.add_rules([
    # (input1_term, input2_term, a, b, c)
    ('low', 'low', 1.0, 0.5, 2.0),    # y = 1.0*x1 + 0.5*x2 + 2.0
    ('low', 'high', 2.0, 1.0, 0.0),   # y = 2.0*x1 + 1.0*x2 + 0.0
    ('high', 'low', 0.5, 2.0, 1.0),   # y = 0.5*x1 + 2.0*x2 + 1.0
    ('high', 'high', 1.0, 1.0, 3.0)   # y = 1.0*x1 + 1.0*x2 + 3.0
])

# Evaluate
result = system.evaluate(x1=7, x2=3)
print(f"y = {result['y']:.2f}")
```

How it works: 1. Fuzzify: $x_1=7 \rightarrow \text{low}=0.6, \text{high}=0.4$; $x_2=3 \rightarrow \text{low}=0.4, \text{high}=0.6$ 2. Calculate rule activations (AND = MIN): - Rule 1: $\min(0.6, 0.4) = 0.4 \rightarrow y_1 = 1.0 \times 7 + 0.5 \times 3 + 2.0 = 10.5$ - Rule 2: $\min(0.6, 0.6) = 0.6 \rightarrow y_2 = 2.0 \times 7 + 1.0 \times 3 + 0.0 = 17.0$ - Rule 3: $\min(0.4, 0.4) = 0.4 \rightarrow y_3 = 0.5 \times 7 + 2.0 \times 3 + 1.0 = 10.5$ - Rule 4: $\min(0.4, 0.6) = 0.4 \rightarrow y_4 = 1.0 \times 7 + 1.0 \times 3 + 3.0 = 13.0$ 3. Weighted average: $y = (0.4 \times 10.5 + 0.6 \times 17 + 0.4 \times 10.5 + 0.4 \times 13) / (0.4 + 0.6 + 0.4 + 0.4)$

First-Order Sugeno

Outputs are **linear functions** of inputs.

5.5 Advanced Topics

Custom T-norms and S-norms

```
system = MamdaniSystem(
    t_norm='product',      # AND: a * b instead of min(a, b)
    s_norm='probabilistic', # OR: a + b - a*b instead of max(a, b)
```

```
implication='product'     # Larsen instead of Mamdani
)
```

Options:

T-norms (AND): - 'min' (default): $\min(a, b)$ - 'product': $a \times b$ - 'lukasiewicz': $\max(0, a + b - 1)$

S-norms (OR): - 'max' (default): $\max(a, b)$ - 'probabilistic': $a + b - a \times b$ - 'bounded': $\min(1, a + b)$

Detailed Evaluation

Get intermediate results:

```
details = system.evaluate_detailed(temperature=25)

print("Fuzzified inputs:")
print(details['inputs'])
# {'temperature': {'cold': 0.25, 'warm': 0.5, 'hot': 0.25}}

print("\nRule activations:")
for i, activation in enumerate(details['rule_activations']):
    print(f" Rule {i+1}: {activation:.3f}")

print("\nAggregated output MF:")
print(details['aggregated'])

print("\nFinal outputs:")
print(details['outputs'])
```

5.6 Design Guidelines

1. Number of Rules

For a system with n inputs and k terms per input: - **Maximum rules:** k^n (combinatorial explosion!) - **Typical rules:** $0.3 \times k^n$ to $0.7 \times k^n$

Example: 2 inputs, 5 terms each: - Max: $5^2 = 25$ rules - Typical: 8-18 rules (skip irrelevant combinations)

2. Term Overlap

Adjacent terms should overlap by **25-50%**:

```
# Good overlap
system.add_term('temp', 'cold', 'triangular', (0, 0, 20))
system.add_term('temp', 'warm', 'triangular', (15, 25, 35))
# Overlaps at 15-20
system.add_term('temp', 'hot', 'triangular', (30, 40, 40))
# Overlaps at 30-35
```

3. Rule Completeness

Every possible input combination should activate **at least one rule**.

Check coverage:

```
# Test grid
import numpy as np
```

5.7 Troubleshooting

Problem: Output is always the same

Cause: Rules are not being activated.

Debug:

```
details = system.evaluate_detailed(temperature=25)
print(details['rule_activations']) # All zeros?
```

Fix: Check term coverage with `system.plot_variables()`.

Problem: Output is stuck at universe boundary

Cause: All active rules point to extreme values.

Fix: Add intermediate terms or adjust MF parameters.

Problem: System is too slow

Solutions: 1. Use Sugeno instead of Mamdani 2. Reduce number of points in universe (default: 1000) 3. Use simpler MFs (triangular instead of gaussian) 4. Cache results for repeated inputs

```
temps = np.linspace(0, 40, 20)
humids = np.linspace(0, 100, 20)

for t in temps:
    for h in humids:
        try:
            result = system.evaluate(temperature=t, humidity=h)
        except:
            print(f"No coverage at temp={t}, humidity={h}")
```

4. Rule Consistency

Avoid contradictory rules:

Bad:

```
system.add_rules([
    {'temp': 'hot', 'humidity': 'high', 'comfort': 'good'}, # ✗
    {'temp': 'hot', 'humidity': 'high', 'comfort': 'bad'}  # ✗
])
Conflict!
```

Good:

```
system.add_rules([
    {'temp': 'hot', 'humidity': 'high', 'comfort': 'bad'}, # ✓
    {'temp': 'hot', 'humidity': 'low', 'comfort': 'moderate'} # ✓
])
No conflict
```

5.8 Next Steps

- **Learning:** Automatically generate rules from data
 - **API Reference: Inference:** Complete method documentation
 - **Examples: Inference:** Interactive notebooks
-

5.9 Further Reading

- **Mamdani, E. H. (1974):** "Application of fuzzy algorithms for control of simple dynamic plant". *Proceedings of the IEE*, 121(12), 1585-1588.
- **Takagi, T., & Sugeno, M. (1985):** "Fuzzy identification of systems and its applications to modeling and control". *IEEE Transactions on Systems, Man, and Cybernetics*, (1), 116-132.

6. User Guide: Learning Systems

This guide covers how to automatically generate fuzzy systems from data using various learning algorithms.

6.1 Why Learn Fuzzy Systems from Data?

- Manual approach:** - Define membership functions by hand - Write rules based on expert knowledge - Time-consuming and subjective
- Learning approach:** - Automatically extract rules from data - Optimize membership function parameters - Data-driven and objective
- When to use learning:** - You have training data (input-output pairs) - Expert knowledge is incomplete or unavailable - You need to tune an existing system - The system needs to adapt over time

6.2 Overview of Learning Methods

Method	Type	Speed	Interpretability	Best For
Wang-Mendel	Rule extraction	⚡⚡⚡ Fast	★★★★ High	Quick prototyping, simple datasets
ANFIS	Neuro-fuzzy	⚡⚡ Moderate	★★★ Moderate	Function approximation, regression
Mamdani Learning	Metaheuristics	⚡ Slow	★★★★ High	Complex optimization, interpretable rules

Quick decision guide: - Need interpretable rules fast? → Wang-Mendel - Need precise predictions? → ANFIS - Need custom optimization? → Mamdani Learning + PSO/DE/GA

6.3 Wang-Mendel Algorithm

The Wang-Mendel algorithm generates fuzzy rules directly from data in **one pass**.

How It Works

1. **Partition input/output spaces** into fuzzy sets
2. **Generate candidate rules** from each data point
3. **Resolve conflicts** by keeping rules with highest degree
4. **Create rule base** from non-conflicting rules

Basic Example: Regression

```
from fuzzy_systems.learning import WangMendelLearning
import numpy as np

# Generate training data
X = np.linspace(0, 10, 50).reshape(-1, 1)
y = np.sin(X).ravel() + np.random.normal(0, 0.1, 50)

# Create learner
learner = WangMendelLearning(
    n_inputs=1,
    n_outputs=1,
    n_terms=5, # 5 fuzzy sets per variable
    input_ranges=[(0, 10)],
    output_ranges=[(-1.5, 1.5)]
)

# Learn from data
system = learner.fit(X, y)
```

```
# Predict
X_test = np.linspace(0, 10, 100).reshape(-1, 1)
y_pred = learner.predict(X_test)

# Evaluate
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y, learner.predict(X))
print(f"MSE: {mse:.4f}")
```

What happened: 1. Algorithm partitioned [0, 10] into 5 fuzzy sets: very_low, low, medium, high, very_high 2. For each data point, it created a rule like: "IF x is medium THEN y is medium" 3. Conflicting rules were resolved by keeping the one with highest membership degree 4. Result: A Mamdani system with ~15-25 rules (fewer than 5¹=5 maximum)

Classification Example

Wang-Mendel also works for classification:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# Load data
iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(
    iris.data, iris.target, test_size=0.3, random_state=42
)
```

```
# Learn classifier
learner = WangMendelLearning(
    n_inputs=4,
    n_outputs=1,
    n_terms=3,
    input_ranges=[(X_train[:, i].min(), X_train[:, i].max()) for i in range(4)],
    output_ranges=[(0, 2)] # 3 classes: 0, 1, 2
)

system = learner.fit(X_train, y_train)

# Predict
y_pred = learner.predict(X_test).round().astype(int)
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2%}")
```

Wang-Mendel Parameters

Number of Terms (n_terms)

Controls granularity of fuzzy partitions:

```
# Coarse (faster, fewer rules)
learner = WangMendelLearning(n_terms=3, ...)

# Fine (slower, more rules, more precise)
learner = WangMendelLearning(n_terms=7, ...)
```

Guidelines: - **3 terms:** Simple problems, fast prototyping - **5 terms:** Good default for most problems - **7-9 terms:** Complex, nonlinear relationships

Conflict Resolution

When multiple rules have the same antecedent:

```
learner = WangMendelLearning(
    conflict_resolution='degree', # Default: keep rule with highest degree
    # conflict_resolution='first' # Keep first rule encountered
)
```

Strengths and Limitations

Strengths: - ⚡ Very fast (single pass over data) - 🧠 Generates interpretable rules - 🇺🇸 Works with small datasets - 🎯 Good for quick prototyping

Limitations: - 🛠 No parameter optimization (MF shapes are fixed) - 📉 May not achieve best accuracy - 🌀 Sensitive to initial partitioning - 🗑 May generate redundant rules

When to use: - You need a baseline quickly - Interpretability is more important than accuracy - Data is limited or expensive - You want to understand the problem structure

6.4 ANFIS (Adaptive Neuro-Fuzzy Inference System)

ANFIS combines neural networks with fuzzy logic to learn both **rule structure** and **parameters**.

Architecture

ANFIS is a **Sugeno system** trained like a neural network:

```
Input → Fuzzification → Rules → Normalization → Defuzzification → Output
      (Layer 1)      (Layer 2)  (Layer 3)      (Layer 4)
```

Learnable parameters: - **Premise parameters:** Membership function shapes (c, σ for gaussian) - **Consequent parameters:** Linear function coefficients (a, b, c in $y = ax_1 + bx_2 + c$)

```
validation_split=0.2,
verbose=True
)

# Predict
X_test = np.random.rand(50, 2) * 10
y_pred = anfis.predict(X_test)

# Check convergence
import matplotlib.pyplot as plt
plt.plot(history['loss'], label='Train')
plt.plot(history['val_loss'], label='Validation')
plt.xlabel('Epoch')
plt.ylabel('MSE')
plt.legend()
plt.show()
```

Basic Example

```
from fuzzy_systems.learning import ANFIS
import numpy as np

# Generate training data
X = np.random.rand(200, 2) * 10
y = X[:, 0]*2 + 2*X[:, 1] + np.random.normal(0, 0.5, 200)

# Create ANFIS
anfis = ANFIS(
    n_inputs=2,
    n_terms=3, # 3 MFs per input → 3² = 9 rules
    mf_type='gaussian'
)

# Train
history = anfis.fit(
    X, y,
    epochs=50,
    learning_rate=0.01,
    batch_size=32,
```

Training Parameters

Learning Rate

Controls step size of gradient descent:

```
# Too high: unstable, oscillates
anfis = ANFIS(n_inputs=2, n_terms=3)
anfis.fit(X, y, learning_rate=0.1) # ⚠ May diverge

# Too low: slow convergence
anfis.fit(X, y, learning_rate=0.0001) # 🐢 Takes forever

# Good range: 0.001 - 0.01
anfis.fit(X, y, learning_rate=0.005) # ✅ Usually works well
```

Number of Epochs

```
# Monitor validation loss to avoid overfitting
history = anfis.fit(
    X, y,
    epochs=100,
    validation_split=0.2,
    early_stopping=True, # Stop if val_loss doesn't improve
    patience=10
)
```

Batch Size

```
# Small batches: more updates, noisier gradients
anfis.fit(X, y, batch_size=16)

# Large batches: fewer updates, smoother gradients
anfis.fit(X, y, batch_size=128)

# Rule of thumb: 32 or 64 for most problems
```

Hybrid Learning (Advanced)

ANFIS supports **hybrid learning**: gradient descent for premise parameters + least squares for consequent parameters.

```
anfis = ANFIS(
    n_inputs=2,
    n_terms=3,
    learning_method='hybrid' # Faster convergence
)

anfis.fit(X, y, epochs=30) # Needs fewer epochs
```

Comparison:

Method	Speed	Stability	When to use
'gradient'	⚡⚡ Moderate	☆☆ Can be unstable	Small datasets, simple problems
'hybrid'	⚡⚡⚡ Fast	☆☆☆ Stable	Large datasets, complex problems

6.5 Mamdani Learning with Metaheuristics

For **highly interpretable** systems, learn Mamdani rules using metaheuristic optimization.

Why Metaheuristics?

Mamdani systems are hard to optimize with gradients because: - Defuzzification (centroid) is non-differentiable - Rule structure is discrete - MF parameters interact in complex ways

Metaheuristics (PSO, DE, GA) are gradient-free and handle this naturally.

Particle Swarm Optimization (PSO)

PSO simulates a swarm of particles searching for optimal parameters.

```
from fuzzy_systems.learning import MamdaniLearning
import numpy as np

# Generate data
X = np.linspace(0, 10, 100).reshape(-1, 1)
y = np.sin(X).ravel()
```

Extracting Rules

After training, inspect learned rules:

```
# Get Sugeno system
system = anfis.get_system()

# Print rules
for i, rule in enumerate(system.rules):
    print(f"Rule {i+1}:")
    print(f"  IF x1 is {rule['x1']} AND x2 is {rule['x2']}")
    print(f"  THEN y = {rule['a']}*x1 + {rule['b']}*x2 + {rule['c']}")
    print()

# Visualize membership functions
system.plot_variables(['x1', 'x2'])
```

Strengths and Limitations

Strengths: - 🎯 High accuracy on regression tasks - ⚙️ Optimizes both structure and parameters - 📊 Handles nonlinear relationships well - 🚀 Efficient gradient-based learning

Limitations: - 😞 Less interpretable than Wang-Mendel - 📉 Can overfit on small datasets - ⚙️ Requires tuning hyperparameters - 📦 Sugeno output (functions, not linguistic terms)

When to use: - Accuracy is the priority - You have enough training data (100+ samples) - Regression or function approximation task - You can afford complex models

PSO Parameters:

```
learner.optimize(
    X, y,
    method='pso',
    n_particles=30,      # Population size (20-50 typical)
    n_iterations=100,    # Generations (50-200 typical)
    inertia=0.7,         # Velocity decay (0.4-0.9)
    cognitive=1.5,       # Personal best influence
    social=1.5           # Global best influence
)
```

Tuning guide: - **Exploration** (diverse search): High inertia (0.8-0.9), low social (1.0-1.5) - **Exploitation** (refine best): Low inertia (0.4-0.6), high social (2.0-2.5)

```
learner.optimize(
    X, y,
    method='ga',
    crossover_prob=0.8,   # High crossover (0.7-0.9)
    mutation_prob=0.1,   # Low mutation (0.01-0.1)
    selection_method='tournament', # or 'roulette', 'rank'
    tournament_size=3,   # Tournament selection size
    elitism=True          # Preserve best solutions
)
```

Comparing Metaheuristics

```
methods = ['pso', 'de', 'ga']
results = {}

for method in methods:
    system, history = learner.optimize(
        X, y,
        method=method,
        n_iterations=100,
        verbose=False
    )

    y_pred = system.evaluate_batch(X)
    mse = np.mean((y - y_pred['output'])**2)
    results[method] = mse

    print(f"{method.upper():10s} MSE = {mse:.4f}")

# Plot convergence
import matplotlib.pyplot as plt
for method in methods:
    _, history = learner.optimize(X, y, method=method,
                                  n_iterations=100, verbose=False)
    plt.plot(history['fitness'], label=method.upper())

plt.xlabel('Iteration')
plt.ylabel('Fitness (MSE)')
plt.legend()
plt.yscale('log')
plt.show()
```

Differential Evolution (DE)

DE uses difference vectors to mutate solutions.

```
optimized_system, history = learner.optimize(
    X, y,
    method='de',
    population_size=40,
    n_iterations=100,
    mutation_factor=0.8, # F: controls mutation strength
    crossover_prob=0.7,  # CR: controls recombination
    strategy='best1bin', # Mutation strategy
    verbose=True
)
```

DE Parameters:

Parameter	Range	Effect
mutation_factor (F)	0.4-1.0	Higher → more exploration
crossover_prob (CR)	0.5-0.9	Higher → faster convergence
strategy	'best1bin', 'rand1bin', 'best2bin'	Mutation scheme

Strategies: - 'best1bin': Exploits best solution (fast convergence) - 'rand1bin': More exploration (avoid local optima) - 'best2bin': Balanced (good default)

Performance comparison:

Method	Speed	Exploration	Stability
PSO	⚡⚡⚡ Fast	☆☆ Moderate	☆☆
DE	⚡⚡ Moderate	☆☆☆ High	☆☆☆
GA	⚡ Slow	☆☆☆ High	☆☆☆

Genetic Algorithm (GA)

GA uses selection, crossover, and mutation.

```
optimized_system, history = learner.optimize(
    X, y,
    method='ga',
    population_size=50,
    n_iterations=100,
    crossover_prob=0.8,
    mutation_prob=0.1,
    selection_method='tournament',
    tournament_size=3,
    elitism=True, # Keep best individuals
    verbose=True
)
```

GA Parameters:

Rule of thumb: - Start with PSO (fastest, stable) - Switch to DE if PSO gets stuck - Use GA for discrete optimization (e.g., rule selection)

What Gets Optimized?

You can control what parameters are optimized:

```
learner = MamdaniLearning(
    n_inputs=1,
    n_outputs=1,
    n_terms=5,
    optimize_mf=True,      # Optimize membership function parameters
    optimize_rules=True,   # Optimize rule weights
    optimize_defuzz=False  # Keep defuzzification method fixed
)
```

Typical configurations:**Configuration 1: MF parameters only**

```
optimize_mf=True, optimize_rules=False
```

- Fastest - Good if rule structure is already known - Fine-tunes MF shapes

Configuration 2: Full optimization

```
optimize_mf=True, optimize_rules=True
```

- Slowest but most flexible - Optimizes everything - Best accuracy potential

Configuration 3: Rules only

```
optimize_mf=False, optimize_rules=True
```

- Medium speed - Good if MFs are well-designed - Tunes rule weights and operators

Strengths and Limitations

Strengths: - ⭐⭐⭐ Highly interpretable (Mamdani output) - 🔧 No gradients needed - 🌐 Can optimize discrete and continuous parameters - 🌐 Global search (avoids local optima)

Limitations: - ⌚ Very slow (minutes to hours) - 🎲 Stochastic (results vary between runs) - ⚙️ Many hyperparameters to tune - 💾 Memory-intensive for large populations

When to use: - Interpretability is critical - You have time for optimization - Gradient-based methods don't work - You need linguistic outputs (Mamdani)

6.6 Choosing a Learning Method

Decision Tree

```

Do you have labeled data?
├ No → Use expert knowledge (manual design)
└ Yes → Continue

How important is interpretability?
├ Critical → Wang-Mendel or Mamdani Learning
└ Less important → ANFIS

How much data do you have?
├ Small (<100 samples) → Wang-Mendel
├ Medium (100-1000) → ANFIS
└ Large (>1000) → ANFIS or Mamdani + PSO

How much time can you spend?
├ Minutes → Wang-Mendel
├ Hours → ANFIS
└ Hours to days → Mamdani Learning

```

Practical Comparison

```
from fuzzy_systems.learning import WangMendelLearning, ANFIS, MamdaniLearning
import time
```

```

# Prepare data
X_train, y_train = ... # Your data here

# Method 1: Wang-Mendel
start = time.time()
wm = WangMendelLearning(n_inputs=2, n_outputs=1, n_terms=5)
wm_system = wm.fit(X_train, y_train)
wm_time = time.time() - start
wm_pred = wm.predict(X_test)
wm_mse = np.mean((y_test - wm_pred)**2)
print(f"Wang-Mendel: MSE={wm_mse:.4f}, Time={wm_time:.2f}s")

# Method 2: ANFIS
start = time.time()
anfis = ANFIS(n_inputs=2, n_terms=3)
anfis.fit(X_train, y_train, epochs=50, verbose=False)
anfis_time = time.time() - start
anfis_pred = anfis.predict(X_test)
anfis_mse = np.mean((y_test - anfis_pred)**2)
print(f"ANFIS: MSE={anfis_mse:.4f}, Time={anfis_time:.2f}s")

# Method 3: Mamdani + PSO
start = time.time()
ml = MamdaniLearning(n_inputs=2, n_outputs=1, n_terms=5)
ml_system, _ = ml.optimize(X_train, y_train, method='pso',
                           n_iterations=50, verbose=False)
ml_time = time.time() - start
ml_pred = ml_system.evaluate_batch(X_test)['output']
ml_mse = np.mean((y_test - ml_pred)**2)
print(f"Mamdani+PSO: MSE={ml_mse:.4f}, Time={ml_time:.2f}s")

```

6.7 Advanced Topics

Transfer Learning

Start from a pre-trained system:

```

# Load pre-trained system
base_system = MamdaniSystem.load('pretrained.pkl')

# Fine-tune with new data
learner = MamdaniLearning.from_system(base_system)
optimized_system, _ = learner.optimize(
    X_new, y_new,
    method='pso',

```

Ensemble Learning

Combine multiple fuzzy systems:

```

from fuzzy_systems.learning import FuzzyEnsemble

# Train multiple systems
systems = []

```



```
for seed in range(5):
    np.random.seed(seed)
    learner = MamdaniLearning(n_inputs=2, n_outputs=1, n_terms=5)
    system, _ = learner.optimize(X_train, y_train, method='pso',
                                n_iterations=50, verbose=False)
    systems.append(system)

# Create ensemble
ensemble = FuzzyEnsemble(systems, method='average') # or 'weighted',
'voting'

# Predict
y_pred = ensemble.predict(X_test)
```

```
# Train ANFIS
anfis = ANFIS(n_inputs=2, n_terms=3)
anfis.fit(X_train, y_train, epochs=50, verbose=False)

# Evaluate
y_pred = anfis.predict(X_val)
mse = mean_squared_error(y_val, y_pred)
scores.append(mse)

print(f"Cross-validation MSE: {np.mean(scores):.4f} ± {np.std(scores):.4f}")
```

Cross-Validation

Evaluate generalization performance:

```
from sklearn.model_selection import KFold
from sklearn.metrics import mean_squared_error

kfold = KFold(n_splits=5, shuffle=True, random_state=42)
scores = []

for train_idx, val_idx in kfold.split(X):
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]
```

Regularization

Prevent overfitting in ANFIS:

```
anfis = ANFIS(n_inputs=2, n_terms=3)
anfis.fit(
    X, y,
    epochs=100,
    learning_rate=0.01,
    l2_penalty=0.001, # L2 regularization on consequent parameters
    dropout=0.1, # Dropout on rule activations
    validation_split=0.2
)
```

6.8 Design Guidelines

1. Data Preparation

Normalization:

```
from sklearn.preprocessing import StandardScaler

scaler_X = StandardScaler()
scaler_y = StandardScaler()

X_scaled = scaler_X.fit_transform(X)
y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).ravel()

# Train on scaled data
anfis.fit(X_scaled, y_scaled, ...)

# Predict and inverse transform
y_pred_scaled = anfis.predict(X_test_scaled)
y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1)).ravel()
```

Why normalize: - Improves convergence speed - Balances influence of different features - Prevents numerical instability

2. Number of Terms vs Dataset Size

Dataset Size	Recommended n_terms	Total Rules (2 inputs)
< 50 samples	3	9
50-200 samples	3-5	9-25
200-1000 samples	5-7	25-49
> 1000 samples	7-9	49-81

Rule of thumb: Total rules should be $\leq N/10$ where N is dataset size.

3. Avoiding Overfitting

Symptoms: - Training error very low, test error high - Validation loss starts increasing after some epochs - System output is jagged or oscillates

Solutions:

Reduce model complexity:

```
# Fewer terms
anfis = ANFIS(n_inputs=2, n_terms=3) # instead of 5 or 7

# Simpler MF types
learner = MamdaniLearning(mf_type='triangular') # instead of gaussian
```

Early stopping:

```
anfis.fit(X, y, epochs=200, early_stopping=True, patience=15)
```

Regularization:

```
anfis.fit(X, y, l2_penalty=0.01)
```

Get more data: - Collect more samples - Use data augmentation (carefully!) - Use cross-validation to detect overfitting

4. Hyperparameter Tuning

Use grid search or random search:

```
from sklearn.model_selection import ParameterGrid
from sklearn.metrics import mean_squared_error

# Define parameter grid
```

```
param_grid = {
    'n_terms': [3, 5, 7],
    'learning_rate': [0.001, 0.005, 0.01],
    'l2_penalty': [0, 0.001, 0.01]
}

best_score = float('inf')
best_params = None

for params in ParameterGrid(param_grid):
    anfis = ANFIS(n_inputs=2, n_terms=params['n_terms'])
    anfis.fit(X_train, y_train,
              epochs=50,
              learning_rate=params['learning_rate'],
              l2_penalty=params['l2_penalty'],
```

```
verbose=False)

y_pred = anfis.predict(X_val)
score = mean_squared_error(y_val, y_pred)

if score < best_score:
    best_score = score
    best_params = params

print(f"Best params: {best_params}")
print(f"Best MSE: {best_score:.4f}")
```

6.9 Troubleshooting

Problem: ANFIS loss is NaN

Causes: - Learning rate too high - Numerical overflow

Solutions:

```
# Reduce learning rate
anfis.fit(X, y, learning_rate=0.001)

# Normalize data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Use gradient clipping
anfis.fit(X, y, clip_gradients=True, max_grad_norm=1.0)
```

Problem: Wang-Mendel generates too many rules

Cause: Too many terms or sparse data distribution.

Solutions:

Reduce n_terms:

```
learner = WangMendelLearning(n_terms=3) # instead of 5 or 7
```

Prune rules after learning:

```
system = learner.fit(X, y)

# Remove rules with low activation
learner.prune_rules(min_activation=0.1)

# Or keep only top K rules
learner.keep_top_rules(k=20)
```

Problem: Metaheuristic optimization is stuck

Symptoms: - Fitness doesn't improve after many iterations - All particles/individuals converge to same solution

Solutions:

Increase diversity:

```
# PSO: increase inertia
learner.optimize(X, y, method='pso', inertia=0.9)

# DE: increase mutation factor
learner.optimize(X, y, method='de', mutation_factor=0.9)

# GA: increase mutation probability
learner.optimize(X, y, method='ga', mutation_prob=0.2)
```

Increase population:

```
learner.optimize(X, y, method='pso', n_particles=50) # instead of 30
```

Try different method:

```
# If PSO stuck, try DE
learner.optimize(X, y, method='de')
```

Problem: Learning is too slow

For ANFIS:

```
# Reduce epochs
anfis.fit(X, y, epochs=30) # instead of 100

# Increase batch size
anfis.fit(X, y, batch_size=128) # instead of 32

# Use hybrid learning
anfis = ANFIS(learning_method='hybrid')
```

For metaheuristics:

```
# Reduce population and iterations
learner.optimize(X, y, method='pso',
                 n_particles=20, n_iterations=50)

# Parallelize (if available)
learner.optimize(X, y, method='pso', n_jobs=-1)
```

6.10 Next Steps

- **Inference Systems:** Build systems manually before learning
- **API Reference: Learning:** Complete method documentation

- **Examples: Learning:** Interactive notebooks
-

6.11 Further Reading

- **Wang, L. X., & Mendel, J. M. (1992):** "Generating fuzzy rules by learning from examples". *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6), 1414-1427.
- **Jang, J. S. (1993):** "ANFIS: adaptive-network-based fuzzy inference system". *IEEE Transactions on Systems, Man, and Cybernetics*, 23(3), 665-685.
- **Eberhart, R., & Kennedy, J. (1995):** "Particle swarm optimization". *Proceedings of ICNN'95*, Vol. 4, 1942-1948.
- **Storn, R., & Price, K. (1997):** "Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces". *Journal of Global Optimization*, 11(4), 341-359.

7. User Guide: Fuzzy Dynamical Systems

This guide covers how to model and solve dynamical systems with fuzzy parameters and initial conditions.

7.1 What are Fuzzy Dynamical Systems?

Classical dynamical systems: - Crisp parameters ($r = 0.5$) - Crisp initial conditions ($y_0 = 10$) - Deterministic evolution

Fuzzy dynamical systems: - Uncertain parameters ($r \approx$ "around 0.5") - Uncertain initial conditions ($y_0 \approx$ "approximately 10") - Prediction bands instead of single trajectories

Why use fuzzy dynamics: - 📊 Model parameter uncertainty - 🧑 Handle measurement errors - 🔄 Propagate uncertainty through time - 🌐 Capture expert knowledge about ranges

7.2 Overview of Methods

Method	Type	Best For	Output
Fuzzy ODE	Continuous-time	ODEs with fuzzy IVPs	α -level trajectories
p-Fuzzy (Discrete)	Discrete-time	Difference equations	Fuzzy number sequences
p-Fuzzy (Continuous)	Continuous-time	Continuous dynamics	Fuzzy trajectories

Quick decision: - Have a differential equation? → Fuzzy ODE - Have a difference equation? → p-Fuzzy Discrete - Need interactive dynamics? → p-Fuzzy Continuous

7.3 Fuzzy Numbers

Before solving fuzzy dynamical systems, we need to represent uncertainty.

Creating Fuzzy Numbers

```
from fuzzy_systems.dynamics import FuzzyNumber
import matplotlib.pyplot as plt

# Triangular fuzzy number: "approximately 10"
y0 = FuzzyNumber.triangular(center=10, spread=2)

# Trapezoidal: "between 8 and 12, most likely 9-11"
y0 = FuzzyNumber.trapezoidal(a=8, b=9, c=11, d=12)

# Gaussian: "around 10 with standard deviation 1"
y0 = FuzzyNumber.gaussian(center=10, sigma=1)

# Plot
y0.plot()
plt.xlabel('Value')
plt.ylabel('Membership')
plt.title('Fuzzy Initial Condition')
plt.show()
```

```
b = FuzzyNumber.triangular(center=3, spread=0.5)

# Addition
c = a + b # Approximately 8 ± 1.5

# Subtraction
d = a - b # Approximately 2 ± 1.5

# Multiplication
e = a * b # Approximately 15 ± ...

# Scalar operations
f = 2 * a # Approximately 10 ± 2
g = a + 5 # Approximately 10 ± 1

# Plot results
import matplotlib.pyplot as plt
fig, axes = plt.subplots(2, 3, figsize=(12, 6))
a.plot(ax=axes[0, 0], title='a')
b.plot(ax=axes[0, 1], title='b')
c.plot(ax=axes[0, 2], title='a + b')
d.plot(ax=axes[1, 0], title='a - b')
e.plot(ax=axes[1, 1], title='a * b')
f.plot(ax=axes[1, 2], title='2 * a')
plt.tight_layout()
plt.show()
```

Operations with Fuzzy Numbers

Fuzzy numbers support arithmetic operations:

```
# Create fuzzy numbers
a = FuzzyNumber.triangular(center=5, spread=1)
```

Operations use α -level arithmetic: - Addition: $[a, b] + [c, d] = [a+c, b+d]$ - Multiplication: $[a, b] \times [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$

α-levels

Access specific confidence intervals:

```
y0 = FuzzyNumber.triangular(center=10, spread=2)

# Get 0.5-level (50% confidence)
lower, upper = y0.alpha_cut(alpha=0.5)
```

```
print(f"0.5-level: [{lower:.2f}, {upper:.2f}]") # [9.0, 11.0]

# Get support (0-level)
lower, upper = y0.alpha_cut(alpha=0)
print(f"Support: [{lower:.2f}, {upper:.2f}]") # [8.0, 12.0]

# Get core (1-level)
lower, upper = y0.alpha_cut(alpha=1)
print(f"Core: [{lower:.2f}, {upper:.2f}]") # [10.0, 10.0]
```

7.4 Fuzzy ODE Solver

Solve ordinary differential equations with fuzzy initial conditions using the **α-level method**.

How It Works

- 1. **Choose α-levels:** 0, 0.25, 0.5, 0.75, 1.0
- 2. **For each α:**
- 3. Extract interval $[y_{\text{lower}}(\alpha), y_{\text{upper}}(\alpha)]$
- 4. Solve ODE twice: once with y_{lower} , once with y_{upper}
- 5. **Reconstruct fuzzy solution** from intervals at each time point

Example 1: Logistic Growth

Model population growth with uncertain initial population:

$$\frac{dy}{dt} = r \cdot y \cdot \left(1 - \frac{y}{K}\right)$$

```
from fuzzy_systems.dynamics import FuzzyODESolver, FuzzyNumber
import numpy as np
import matplotlib.pyplot as plt

# Define logistic equation
def logistic(t, y, r, K):
    """
    y: list of fuzzy numbers [y(t)]
    Returns: dy/dt
    """
    return r * y[0] * (1 - y[0] / K)

# Fuzzy initial condition: "approximately 10"
y0 = FuzzyNumber.triangular(center=10, spread=2)

# Solve
solver = FuzzyODESolver(
    f=logistic,
    t_span=(0, 20),
    y0_fuzzy=[y0],
    params={'r': 0.3, 'K': 100},
    n_alpha=11 # 11 α-levels
)

solution = solver.solve()

# Plot
solver.plot()
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Logistic Growth with Fuzzy Initial Condition')
plt.show()
```

Interpretation: - Dark region: High confidence (α = 1.0) - Light region: Low confidence (α = 0.0) - Uncertainty **increases** over time (characteristic of fuzzy ODEs)

Example 2: Predator-Prey (Lotka-Volterra)

Two-dimensional system with fuzzy initial conditions:

$$\begin{aligned} \frac{dx}{dt} &= \alpha x - \beta x y \\ \frac{dy}{dt} &= \delta x y - \gamma y \end{aligned}$$

```
def predator_prey(t, y, alpha, beta, delta, gamma):
    """
    y[0]: prey population
    y[1]: predator population
    """
    x, y_pred = y
    dx_dt = alpha * x - beta * x * y_pred
    dy_dt = delta * x * y_pred - gamma * y_pred
    return [dx_dt, dy_dt]

# Fuzzy initial conditions
x0 = FuzzyNumber.triangular(center=40, spread=5) # Prey
y0 = FuzzyNumber.triangular(center=9, spread=1) # Predator

# Solve
solver = FuzzyODESolver(
    f=predator_prey,
    t_span=(0, 30),
    y0_fuzzy=[x0, y0],
    params={'alpha': 0.1, 'beta': 0.02, 'delta': 0.01, 'gamma': 0.1},
    n_alpha=11
)

solution = solver.solve()

# Plot both populations
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

solver.plot(variable_index=0, ax=axes[0])
axes[0].set_ylabel('Prey Population')
axes[0].set_title('Prey')

solver.plot(variable_index=1, ax=axes[1])
axes[1].set_ylabel('Predator Population')
axes[1].set_title('Predator')

plt.tight_layout()
plt.show()

# Phase portrait
solver.plot_phase(variable_indices=(0, 1))
plt.xlabel('Prey')
plt.ylabel('Predator')
plt.title('Phase Portrait')
plt.show()
```

Solver Parameters

```
solver = FuzzyODESolver(
    f=equation,
    t_span=(t_start, t_end),
    y0_fuzzy=[y0_1, y0_2, ...], # List of FuzzyNumber
    params={...},                # Dictionary of crisp parameters
    n_alpha=11,                  # Number of α-levels (odd number)
    method='RK45',                # Integration method
)
```

```

    rtol=1e-6,          # Relative tolerance
    atol=1e-9          # Absolute tolerance
)

```

Integration methods: - 'RK45': Runge-Kutta 4(5) (default, good balance)
 - 'RK23': Runge-Kutta 2(3) (faster, less accurate) - 'DOP853': Runge-Kutta 8 (slower, very accurate) - 'BDF': Backward differentiation (for stiff problems)

Number of α -levels: - 5-7: Fast, coarse uncertainty bands - 11-21: Good balance (recommended) - 51+: Slow, smooth bands

Accessing Solutions

```

solution = solver.solve()

# Time points
t = solution['t']

# Fuzzy solution at each time point
y_fuzzy = solution['y'] # List of lists of FuzzyNumber

# Get specific  $\alpha$ -level trajectory
alpha = 0.5
y_lower, y_upper = solver.get_alpha_trajectory(alpha, variable_index=0)

# Plot custom  $\alpha$ -levels
fig, ax = plt.subplots()
for alpha in [0, 0.25, 0.5, 0.75, 1.0]:
    lower, upper = solver.get_alpha_trajectory(alpha, 0)
    ax.fill_between(t, lower, upper, alpha=0.3, label=f' $\alpha$ {alpha}')
ax.legend()
ax.set_xlabel('Time')
ax.set_ylabel('y(t)')
plt.show()

```

Fuzzy Parameters

Parameters can also be fuzzy:

```

# Fuzzy growth rate: "approximately 0.3"
r_fuzzy = FuzzyNumber.triangular(center=0.3, spread=0.05)

# Convert to crisp samples for Monte Carlo
n_samples = 100
r_samples = [r_fuzzy.sample() for _ in range(n_samples)]

# Solve for each sample
trajectories = []
for r_val in r_samples:
    solver = FuzzyODESolver(
        f=logistic,
        t_span=(0, 20),
        y0_fuzzy=[y0],
        params={'r': r_val, 'K': 100},
        n_alpha=5 # Fewer  $\alpha$ -levels for speed
    )
    solution = solver.solve()
    trajectories.append(solution['y'][0]) # First variable

# Plot envelope
import numpy as np
t = solution['t']
y_array = np.array([traj for traj in trajectories])
y_mean = y_array.mean(axis=0)
y_std = y_array.std(axis=0)

plt.fill_between(t, y_mean - 2*y_std, y_mean + 2*y_std, alpha=0.3)
plt.plot(t, y_mean, 'r-', linewidth=2)
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Logistic Growth with Fuzzy Parameter')
plt.show()

```

7.5 p-Fuzzy Systems (Discrete-Time)

p-Fuzzy systems use fuzzy rules to define dynamical systems.

Basic Concept

Instead of equations, use **linguistic rules**:

```

IF x is LOW THEN x_next is MEDIUM
IF x is MEDIUM THEN x_next is HIGH
IF x is HIGH THEN x_next is LOW

```

Example: Population Dynamics

```

from fuzzy_systems.dynamics import PFuzzyDiscrete
import numpy as np
import matplotlib.pyplot as plt

# Create discrete p-fuzzy system
system = PFuzzyDiscrete(n_variables=1)

# Add input (current state)
system.add_input('x', (0, 100))
system.add_term('x', 'low', 'trapezoidal', (0, 0, 20, 40))
system.add_term('x', 'medium', 'triangular', (30, 50, 70))
system.add_term('x', 'high', 'trapezoidal', (60, 80, 100, 100))

# Add output (next state)
system.add_output('x_next', (0, 100))
system.add_term('x_next', 'low', 'trapezoidal', (0, 0, 20, 40))
system.add_term('x_next', 'medium', 'triangular', (30, 50, 70))
system.add_term('x_next', 'high', 'trapezoidal', (60, 80, 100, 100))

# Add rules (discrete map)

```

```

system.add_rules([
    {'x': 'low', 'x_next': 'medium'}, # Low pop → grows to medium
    {'x': 'medium', 'x_next': 'high'}, # Medium → grows to high
    {'x': 'high', 'x_next': 'low'}    # High → collapses to low
])

# Simulate
x0 = 10 # Initial population
trajectory = system.simulate(x0=x0, n_steps=20)

# Plot
plt.plot(trajectory['x'], 'o-')
plt.xlabel('Time Step')
plt.ylabel('Population')
plt.title('Discrete p-Fuzzy Population Dynamics')
plt.grid(True)
plt.show()

# Phase diagram (x_t vs x_{t+1})
plt.plot(trajectory['x'][:-1], trajectory['x'][1:], 'o-')
plt.plot([0, 100], [0, 100], 'k--', alpha=0.3) # Identity line
plt.xlabel('x(t)')
plt.ylabel('x(t+1)')
plt.title('Discrete Map')
plt.grid(True)
plt.show()

```

Example: Predator-Prey (Discrete)

Two-variable system:

```

system = PFuzzyDiscrete(n_variables=2)

```

```
# Prey (x)
system.add_input('x', (0, 100))
system.add_term('x', 'low', 'triangular', (0, 0, 50))
system.add_term('x', 'high', 'triangular', (50, 100, 100))

system.add_output('x_next', (0, 100))
system.add_term('x_next', 'low', 'triangular', (0, 0, 50))
system.add_term('x_next', 'medium', 'triangular', (25, 50, 75))
system.add_term('x_next', 'high', 'triangular', (50, 100, 100))

# Predator (y)
system.add_input('y', (0, 50))
system.add_term('y', 'low', 'triangular', (0, 0, 25))
system.add_term('y', 'high', 'triangular', (25, 50, 50))

system.add_output('y_next', (0, 50))
system.add_term('y_next', 'low', 'triangular', (0, 0, 25))
system.add_term('y_next', 'medium', 'triangular', (12.5, 25, 37.5))
system.add_term('y_next', 'high', 'triangular', (25, 50, 50))

# Rules
system.add_rules([
    # When prey low, predator low → both grow
    {'x': 'low', 'y': 'low', 'x_next': 'medium', 'y_next': 'low'},

    # When prey low, predator high → prey recovers, predator declines
    {'x': 'low', 'y': 'high', 'x_next': 'medium', 'y_next': 'medium'},

    # When prey high, predator low → prey stays high, predator grows
    {'x': 'high', 'y': 'low', 'x_next': 'high', 'y_next': 'medium'},

    # When prey high, predator high → prey declines, predator stays high
    {'x': 'high', 'y': 'high', 'x_next': 'medium', 'y_next': 'high'},
])

# Simulate
trajectory = system.simulate(x0={'x': 40, 'y': 9}, n_steps=50)

# Plot time series
fig, axes = plt.subplots(2, 1, figsize=(10, 6), sharex=True)

axes[0].plot(trajectory['x'], 'b-o', label='Prey')
axes[0].set_ylabel('Prey')
axes[0].legend()
axes[0].grid(True)

axes[1].plot(trajectory['y'], 'r-o', label='Predator')
axes[1].set_ylabel('Predator')
axes[1].set_xlabel('Time Step')
```

```
axes[1].legend()
axes[1].grid(True)

plt.tight_layout()
plt.show()

# Phase portrait
plt.plot(trajectory['x'], trajectory['y'], 'o-')
plt.plot(trajectory['x'][0], trajectory['y'][0], 'go', markersize=10, label='Start')
plt.plot(trajectory['x'][-1], trajectory['y'][-1], 'ro', markersize=10, label='End')
plt.xlabel('Prey')
plt.ylabel('Predator')
plt.title('Phase Portrait')
plt.legend()
plt.grid(True)
plt.show()
```

Multiple Initial Conditions

Explore different starting points:

```
initial_conditions = [
    {'x': 10, 'y': 5},
    {'x': 30, 'y': 15},
    {'x': 70, 'y': 30},
    {'x': 90, 'y': 45}
]

plt.figure(figsize=(10, 6))
for x0, y0 in initial_conditions:
    traj = system.simulate(x0={'x': x0, 'y': y0}, n_steps=30)
    plt.plot(traj['x'], traj['y'], 'o-', alpha=0.6)
    plt.plot(x0, y0, 'o', markersize=10)

plt.xlabel('Prey')
plt.ylabel('Predator')
plt.title('Phase Portrait from Multiple Initial Conditions')
plt.grid(True)
plt.show()
```

7.6 p-Fuzzy Systems (Continuous-Time)

Continuous-time p-fuzzy systems use rules to define **derivatives**.

Example: Logistic Growth

```
from fuzzy_systems.dynamics import PFuzzyContinuous
import numpy as np
import matplotlib.pyplot as plt

# Create continuous p-fuzzy system
system = PFuzzyContinuous(n_variables=1)

# Add input (current population)
system.add_input('x', (0, 100))
system.add_term('x', 'low', 'trapezoidal', (0, 0, 20, 40))
system.add_term('x', 'medium', 'triangular', (30, 50, 70))
system.add_term('x', 'high', 'trapezoidal', (60, 80, 100, 100))

# Add output (growth rate dx/dt)
system.add_output('dx_dt', (-10, 10))
system.add_term('dx_dt', 'negative', 'trapezoidal', (-10, -10, -5, 0))
system.add_term('dx_dt', 'zero', 'triangular', (-2, 0, 2))
system.add_term('dx_dt', 'positive', 'trapezoidal', (0, 5, 10, 10))

# Add rules
system.add_rules([
    {'x': 'low', 'dx_dt': 'positive'},      # Low pop → grows
    {'x': 'medium', 'dx_dt': 'positive'},   # Medium → still grows
    {'x': 'high', 'dx_dt': 'negative'}     # High → declines (carrying capacity)
])

# Simulate
t_span = (0, 20)
```

```
x0 = 10
solution = system.simulate(x0=x0, t_span=t_span, method='RK45')

# Plot
plt.plot(solution['t'], solution['x'])
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Continuous p-Fuzzy Logistic Growth')
plt.grid(True)
plt.show()
```

Example: Predator-Prey (Continuous)

```
system = PFuzzyContinuous(n_variables=2)

# Prey (x)
system.add_input('x', (0, 100))
system.add_term('x', 'low', 'triangular', (0, 0, 50))
system.add_term('x', 'medium', 'triangular', (25, 50, 75))
system.add_term('x', 'high', 'triangular', (50, 100, 100))

system.add_output('dx_dt', (-20, 20))
system.add_term('dx_dt', 'decrease', 'triangular', (-20, -20, 0))
system.add_term('dx_dt', 'stable', 'triangular', (-5, 0, 5))
system.add_term('dx_dt', 'increase', 'triangular', (0, 20, 20))

# Predator (y)
system.add_input('y', (0, 50))
system.add_term('y', 'low', 'triangular', (0, 0, 25))
system.add_term('y', 'medium', 'triangular', (12.5, 25, 37.5))
```

```

system.add_term('y', 'high', 'triangular', (25, 50, 50))

system.add_output('dy_dt', (-10, 10))
system.add_term('dy_dt', 'decrease', 'triangular', (-10, -10, 0))
system.add_term('dy_dt', 'stable', 'triangular', (-2, 0, 2))
system.add_term('dy_dt', 'increase', 'triangular', (0, 10, 10))

# Rules based on ecology
system.add_rules([
    # Low prey → prey can grow, predator declines
    {'x': 'low', 'y': 'low', 'dx_dt': 'increase', 'dy_dt': 'stable'},
    {'x': 'low', 'y': 'high', 'dx_dt': 'decrease', 'dy_dt': 'decrease'}
],

    # Medium prey → balanced
    {'x': 'medium', 'y': 'low', 'dx_dt': 'increase', 'dy_dt': 'increase'},
    {'x': 'medium', 'y': 'medium', 'dx_dt': 'stable', 'dy_dt': 'stable'},
    {'x': 'medium', 'y': 'high', 'dx_dt': 'decrease', 'dy_dt': 'increase'},

    # High prey → prey declines, predator grows
    {'x': 'high', 'y': 'low', 'dx_dt': 'stable', 'dy_dt': 'increase'},
    {'x': 'high', 'y': 'high', 'dx_dt': 'decrease', 'dy_dt': 'stable'},
])

# Simulate
solution = system.simulate(
    x0={'x': 40, 'y': 9},
    t_span=(0, 50),

```

```

        method='RK45'
    )

    # Plot time series
    fig, axes = plt.subplots(2, 1, figsize=(10, 6), sharex=True)

    axes[0].plot(solution['t'], solution['x'], 'b-')
    axes[0].set_ylabel('Prey')
    axes[0].grid(True)

    axes[1].plot(solution['t'], solution['y'], 'r-')
    axes[1].set_ylabel('Predator')
    axes[1].set_xlabel('Time')
    axes[1].grid(True)

    plt.tight_layout()
    plt.show()

    # Phase portrait
    plt.plot(solution['x'], solution['y'])
    plt.plot(solution['x'][0], solution['y'][0], 'go', markersize=10,
              label='Start')
    plt.xlabel('Prey')
    plt.ylabel('Predator')
    plt.title('Continuous p-Fuzzy Phase Portrait')
    plt.legend()
    plt.grid(True)
    plt.show()

```

7.7 Comparing Methods

Let's compare all three methods on the same problem (logistic growth):

```

import numpy as np
import matplotlib.pyplot as plt
from fuzzy_systems.dynamics import (
    FuzzyODESolver, FuzzyNumber,
    PFuzzyDiscrete, PFuzzyContinuous
)

# Parameters
r, K = 0.3, 100
y0_value = 10
t_max = 20

# Method 1: Fuzzy ODE
def logistic_ode(t, y, r, K):
    return r * y[0] * (1 - y[0] / K)

y0_fuzzy = FuzzyNumber.triangular(center=y0_value, spread=2)
solver_ode = FuzzyODESolver(
    f=logistic_ode,
    t_span=(0, t_max),
    y0_fuzzy=[y0_fuzzy],
    params={'r': r, 'K': K},
    n_alpha=11
)
solution_ode = solver_ode.solve()

# Method 2: p-Fuzzy Discrete
system_discrete = PFuzzyDiscrete(n_variables=1)
system_discrete.add_input('x', (0, K))
system_discrete.add_term('x', 'low', 'trapezoidal', (0, 0, K*0.3, K*0.5))
system_discrete.add_term('x', 'medium', 'triangular', (K*0.4, K*0.6, K*0.8))
system_discrete.add_term('x', 'high', 'trapezoidal', (K*0.7, K*0.9, K, K))

system_discrete.add_output('x_next', (0, K))
system_discrete.add_term('x_next', 'low', 'trapezoidal', (0, 0, K*0.3, K*0.5))
system_discrete.add_term('x_next', 'medium', 'triangular', (K*0.4, K*0.6, K*0.8))
system_discrete.add_term('x_next', 'high', 'trapezoidal', (K*0.7, K*0.9, K, K))

# Approximating continuous dynamics with discrete map
system_discrete.add_rules([
    {'x': 'low', 'x_next': 'medium'},
    {'x': 'medium', 'x_next': 'high'},
    {'x': 'high', 'x_next': 'high'}
])

traj_discrete = system_discrete.simulate(x0=y0_value, n_steps=int(t_max))

# Method 3: p-Fuzzy Continuous
system_continuous = PFuzzyContinuous(n_variables=1)
system_continuous.add_input('x', (0, K))
system_continuous.add_term('x', 'low', 'trapezoidal', (0, 0, K*0.3, K*0.5))
system_continuous.add_term('x', 'medium', 'triangular', (K*0.4, K*0.6, K*0.8))
system_continuous.add_term('x', 'high', 'trapezoidal', (K*0.7, K*0.9, K, K))

```



```
system_continuous.add_output('dx_dt', (-K, K))
system_continuous.add_term('dx_dt', 'negative', 'triangular', (-K, -K, 0))
system_continuous.add_term('dx_dt', 'zero', 'triangular', (-K*0.1, 0, K*0.1))
system_continuous.add_term('dx_dt', 'positive', 'triangular', (0, K, K))

system_continuous.add_rules([
    {'x': 'low', 'dx_dt': 'positive'},
    {'x': 'medium', 'dx_dt': 'positive'},
    {'x': 'high', 'dx_dt': 'zero'}
])

solution_continuous = system_continuous.simulate(
    x0=y0_value,
    t_span=(0, t_max),
    method='RK45'
)

# Compare
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Fuzzy ODE
solver_ode.plot(ax=axes[0])
axes[0].set_title('Fuzzy ODE ( $\alpha$ -levels)')
axes[0].set_ylabel('Population')

# p-Fuzzy Discrete
axes[1].plot(range(len(traj_discrete['x'])), traj_discrete['x'], 'o-')
axes[1].set_title('p-Fuzzy Discrete')
axes[1].set_xlabel('Time Step')
axes[1].grid(True)

# p-Fuzzy Continuous
axes[2].plot(solution_continuous['t'], solution_continuous['x'])
axes[2].set_title('p-Fuzzy Continuous')
axes[2].set_xlabel('Time')
axes[2].grid(True)

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

Observations: - **Fuzzy ODE:** Shows uncertainty bands growing over time - **p-Fuzzy Discrete:** Step-wise evolution, good for discrete events - **p-Fuzzy Continuous:** Smooth trajectories, rule-based dynamics

7.8 Design Guidelines

1. Choosing α -levels

Trade-off: accuracy vs speed

```
# Fast (3-5 levels)
solver = FuzzyODESolver(..., n_alpha=5)

# Balanced (11-21 levels)
solver = FuzzyODESolver(..., n_alpha=11) # Recommended

# Smooth (51+ levels)
solver = FuzzyODESolver(..., n_alpha=51) # Slow
```

Use fewer α -levels when: - Prototyping or exploring - Computational budget is limited - Rough uncertainty estimates are sufficient

Use more α -levels when: - Creating publication-quality figures - Precise uncertainty quantification needed - Computational resources available

2. Fuzzy Number Shapes

Triangular vs Trapezoidal vs Gaussian:

```
# Triangular: "approximately X"
y0 = FuzzyNumber.triangular(center=10, spread=2)

# Trapezoidal: "between A and B"
y0 = FuzzyNumber.trapezoidal(a=8, b=9, c=11, d=12)
```

```
# Gaussian: "normally distributed"
y0 = FuzzyNumber.gaussian(center=10, sigma=1)
```

Guidelines: - Use **triangular** for symmetric uncertainty - Use **trapezoidal** for ranges with plateaus - Use **gaussian** for measurement errors

3. Rule Design for p-Fuzzy

Principle: Rules should reflect domain knowledge

Good rules (ecologically sound):

```
system.add_rules([
    {'prey': 'low', 'predator': 'high', 'dprey_dt': 'decrease'},
    {'prey': 'high', 'predator': 'low', 'dprey_dt': 'increase'}
])
```

Bad rules (contradictory):

```
system.add_rules([
    {'prey': 'low', 'dprey_dt': 'increase'},
    {'prey': 'low', 'dprey_dt': 'decrease'} # Conflict!
])
```

Check rule coverage:

```
# Visualize rule activation
system.plot_rule_matrix() # For 2D systems
```

4. Integration Method Selection

Method	Speed	Accuracy	Best for
'RK23'	⚡⚡⚡	☆☆	Fast prototyping, smooth problems
'RK45'	⚡⚡	☆☆☆	Default, most problems
'DOP853'	⚡	☆☆☆☆☆	High precision needed
'BDF'	⚡⚡	☆☆☆	Stiff equations

Stiff equations? Try 'BDF' or 'Radau':

```
solver = FuzzyODESolver(..., method='BDF')
```

7.9 Advanced Topics

Sensitivity Analysis

How sensitive is the solution to initial conditions?

```
from fuzzy_systems.dynamics import FuzzyODESolver, FuzzyNumber
import numpy as np
import matplotlib.pyplot as plt

def logistic(t, y, r, K):
    return r * y[0] * (1 - y[0] / K)

# Test different spreads
spreads = [1, 2, 5, 10]
fig, ax = plt.subplots(figsize=(10, 6))

for spread in spreads:
    y0 = FuzzyNumber.triangular(center=10, spread=spread)
    solver = FuzzyODESolver(
        f=logistic,
        t_span=(0, 20),
        y0_fuzzy=y0,
        params={'r': 0.3, 'K': 100},
        n_alpha=11
    )
    solution = solver.solve()

    # Plot envelope
    t = solution['t']
    y_lower, y_upper = solver.get_alpha_trajectory(alpha=0, variable_in
dex=0)
    ax.fill_between(t, y_lower, y_upper, alpha=0.3, label=f'spread={spr
ead}')

ax.set_xlabel('Time')
ax.set_ylabel('Population')
ax.set_title('Sensitivity to Initial Uncertainty')
ax.legend()
ax.grid(True)
plt.show()
```

```
# Perturbed trajectory
x0_perturbed = x0 + epsilon
traj2 = system.simulate(x0=x0_perturbed, n_steps=n_steps)['x']

# Compute divergence
divergence = [np.log(abs(traj2[i] - traj1[i]) / epsilon)
               for i in range(1, n_steps)]

lyapunov = np.mean(divergence)
return lyapunov

# Test
system = PFuzzyDiscrete(n_variables=1)
# ... configure system ...

lambda_max = lyapunov_exponent(system, x0=10)
print(f"Lyapunov exponent: {lambda_max:.4f}")

if lambda_max > 0:
    print("System is chaotic!")
elif lambda_max < 0:
    print("System is stable.")
else:
    print("System is at the edge of chaos.")
```

Bifurcation Diagrams (p-Fuzzy)

Explore parameter space:

```
# Vary a parameter and observe long-term behavior
parameters = np.linspace(0.1, 0.5, 50)
final_states = []

for param in parameters:
    # Modify rule or parameter
    system = create_system_with_param(param)

    # Simulate and discard transient
    traj = system.simulate(x0=10, n_steps=500)
    final_states.append(traj['x'][-100:]) # Last 100 steps

# Plot bifurcation diagram
for i, param in enumerate(parameters):
    plt.plot([param]*len(final_states[i]), final_states[i],
             'k', alpha=0.5)

plt.xlabel('Parameter')
plt.ylabel('Long-term Population')
plt.title('Bifurcation Diagram')
plt.show()
```

Lyapunov Exponents (p-Fuzzy Discrete)

Measure chaos in discrete systems:

```
def lyapunov_exponent(system, x0, n_steps=1000, epsilon=1e-8):
    """Estimate largest Lyapunov exponent."""
    traj1 = system.simulate(x0=x0, n_steps=n_steps)['x']
```

7.10 Troubleshooting

Problem: Fuzzy ODE solution "explodes"

Symptoms: - Solution bands become extremely wide - Values go to infinity

Causes: - Unstable dynamics - Tolerance too loose

Solutions:

```
# Tighten tolerances
solver = FuzzyODESolver(..., rtol=1e-9, atol=1e-12)

# Use more stable integrator
solver = FuzzyODESolver(..., method='BDF')

# Check classical solution first
def check_stability(f, t_span, y0, params):
    from scipy.integrate import solve_ivp
    sol = solve_ivp(lambda t, y: f(t, [y[0]], **params),
                    t_span, [y0], method='RK45')
    plt.plot(sol.t, sol.y[0])
    plt.show()

check_stability(logistic, (0, 20), 10, {'r': 0.3, 'K': 100})
```

```
# Fix: adjust term coverage
system.plot_variables() # Visual check
```

Problem: Discrete p-Fuzzy is stuck in a loop

Symptoms: - Trajectory oscillates between same values - Phase portrait shows closed loop

Explanation: - This may be intentional (limit cycle) - Or rules create an attractor

To verify:

```
# Test multiple initial conditions
for x0 in [10, 30, 50, 70, 90]:
    traj = system.simulate(x0=x0, n_steps=50)
    plt.plot(traj['x'], alpha=0.6)
plt.xlabel('Time Step')
plt.ylabel('x')
plt.title('Trajectories from Different ICs')
plt.show()

# If all converge to same cycle -> it's an attractor
```

Problem: p-Fuzzy system has no output

Symptoms: - `simulate()` returns NaN or constant values - No rules are activating

Solutions:

```
# Debug: check fuzzification
x_test = 50
input_degrees = system.inputs['x'].fuzzify(x_test)
print(f"Input memberships at x={x_test}: {input_degrees}")
# Should be non-zero for at least one term

# Debug: check rule activations
details = system.evaluate_detailed(x=x_test)
print(f"Rule activations: {details['rule_activations']}")
# Should have at least one non-zero activation
```

Problem: Continuous p-Fuzzy doesn't reach equilibrium

Causes: - Rules don't allow convergence - No "zero growth" rules

Solutions:

```
# Add equilibrium rules
system.add_rules([
    {'x': 'medium', 'dx_dt': 'zero'}, # Equilibrium at medium
])

# Or increase tolerance
solution = system.simulate(..., method='RK45', rtol=1e-3)
```

7.11 Next Steps

- **Fundamentals:** Review fuzzy logic basics
- **API Reference: Dynamics:** Complete method documentation
- **Examples: Dynamics:** Interactive notebooks

7.12 Further Reading

- Puri, M. L., & Ralescu, D. A. (1983): "Differentials of fuzzy functions". *Journal of Mathematical Analysis and Applications*, 91(2), 552-558.
- Buckley, J. J., & Feuring, T. (2000): "Fuzzy differential equations". *Fuzzy Sets and Systems*, 110(1), 43-54.
- Barros, L. C., Bassanezi, R. C., & Lodwick, W. A. (2017): *A First Course in Fuzzy Logic, Fuzzy Dynamical Systems, and Biomathematics*. Springer.
- Jafelice, R. M., et al. (2015): "Fuzzy parameter in a prey-predator model". *Nonlinear Analysis: Real World Applications*, 16, 59-71.

8. Core API Reference

The `fuzzy_systems.core` module provides fundamental components for fuzzy logic:

- **Membership functions:** Define fuzzy set shapes
- **Fuzzy sets:** `FuzzySet` and `LinguisticVariable` classes
- **Operators:** AND, OR, NOT operations
- **Defuzzification:** Convert fuzzy to crisp values

8.1 Membership Functions

`triangular(x, params)`

Triangular membership function.

Parameters:

- `x` (float | ndarray): Input value(s)
- `params` (tuple): `(a, b, c)` where:
 - `a`: Left foot
 - `b`: Peak ($\mu = 1$)
 - `c`: Right foot

Returns: `float | ndarray` - Membership degree(s) in `[0, 1]`

Example:

```
import numpy as np
from fuzzy_systems.core import triangular

x = np.linspace(0, 10, 100)
mu = triangular(x, (2, 5, 8))
```

`trapezoidal(x, params)`

Trapezoidal membership function.

Parameters:

- `x` (float | ndarray): Input value(s)
- `params` (tuple): `(a, b, c, d)` where:
 - `a`: Left foot
 - `b`: Left shoulder ($\mu = 1$ starts)
 - `c`: Right shoulder ($\mu = 1$ ends)
 - `d`: Right foot

Returns: `float | ndarray` - Membership degree(s) in `[0, 1]`

Example:

```
from fuzzy_systems.core import trapezoidal

mu = trapezoidal(x, (1, 3, 7, 9))
```

`gaussian(x, params)`

Gaussian (bell-shaped) membership function.

Parameters:

- `x` (float | ndarray): Input value(s)
- `params` (tuple): `(mean, sigma)` where:
 - `mean`: Center of the curve
 - `sigma`: Standard deviation (controls width)

Returns: `float | ndarray` - Membership degree(s) in `[0, 1]`

Example:

```
from fuzzy_systems.core import gaussian

mu = gaussian(x, (5, 1.5))
```

`sigmoid(x, params)`

Sigmoid membership function.

Parameters:

- `x` (float | ndarray): Input value(s)
- `params` (tuple): `(a, c)` where:
 - `a`: Slope parameter
 - `c`: Inflection point (where $\mu = 0.5$)

Returns: `float | ndarray` - Membership degree(s) in `[0, 1]`

Example:

```
from fuzzy_systems.core import sigmoid

mu = sigmoid(x, (1, 5))
```

generalized_bell(x, params)

Generalized bell-shaped membership function.

Parameters:

- `x` (float | ndarray): Input value(s)
- `params` (tuple): (`a`, `b`, `c`) where:
 - `a` : Width parameter
 - `b` : Slope parameter
 - `c` : Center

Returns: float | ndarray - Membership degree(s) in [0, 1]

8.2 Classes

FuzzySet

Represents a fuzzy set with its membership function.

Constructor

FuzzySet(name, mf_type, params, mf_func=None)

Parameters:

- `name` (str): Name of the fuzzy set (e.g., "low", "medium", "high")
- `mf_type` (str): Membership function type ("triangular", "trapezoidal", "gaussian", etc.)
- `params` (tuple): Parameters for the membership function
- `mf_func` (callable, optional): Custom membership function

Example:

```
from fuzzy_systems.core import FuzzySet

fs = FuzzySet(
    name="warm",
    mf_type="triangular",
    params=(15, 22.5, 30)
)
```

Methods

`.membership(x)`

Calculate membership degree of value(s) in this fuzzy set.

Parameters: - `x` (float | ndarray): Input value(s)

Returns: float | ndarray - Membership degree(s)

Example:

```
mu = fs.membership(20) # Returns: 0.727...
```

LinguisticVariable

Represents a linguistic variable with multiple fuzzy terms.

Constructor

LinguisticVariable(name, universe)

Parameters:

- `name` (str): Variable name (e.g., "temperature", "speed")
- `universe` (tuple): Range (`min`, `max`) of the variable

Example:

```
from fuzzy_systems.core import LinguisticVariable

temperature = LinguisticVariable(
    name="temperature",
    universe=(0, 50)
)
```

Methods

`.add_term(name, mf_type, params, mf_func=None)`

Add a fuzzy term to the variable.

Parameters: - `name` (str): Term name (e.g., "cold", "warm", "hot") -
`mf_type` (str): Membership function type - `params` (tuple): Function
parameters - `mf_func` (callable, optional): Custom function

Example:

```
temperature.add_term("cold", "trapezoidal", (0, 0, 10, 20))
temperature.add_term("warm", "triangular", (15, 25, 35))
temperature.add_term("hot", "trapezoidal", (30, 40, 50, 50))
```

Alternative (pass FuzzySet):

```
from fuzzy_systems.core import FuzzySet

cold_set = FuzzySet("cold", "triangular", (0, 0, 20))
temperature.add_term(cold_set)
```

```
.fuzzify(value)
```

Convert a crisp value to fuzzy membership degrees.

Parameters: - `value` (float): Crisp input value

Returns: `dict` - Membership degrees for all terms: `{term_name: degree}`

Example:

```
degrees = temperature.fuzzify(28)
# Returns: {'cold': 0.0, 'warm': 0.143, 'hot': 0.333}
```

```
.plot(ax=None, show=True, figsize=(10, 6), **kwargs)
```

Plot all fuzzy terms of the variable.

Parameters: - `ax` (matplotlib.axes.Axes, optional): Axes to plot on - `show` (bool): Whether to call `plt.show()` - `figsize` (tuple): Figure size if creating new figure - `**kwargs`: Additional matplotlib styling options

Returns: `tuple` - (`fig`, `ax`) matplotlib objects

Example:

```
temperature.plot()
```

8.3 Fuzzy Operators

AND Operators (T-norms)

```
fuzzy_and_min(a, b)
```

Minimum t-norm (standard fuzzy AND).

Parameters: - `a`, `b` (float | ndarray): Membership degrees

Returns: `float` | `ndarray` - `min(a, b)`

Example:

```
from fuzzy_systems.core import fuzzy_and_min
result = fuzzy_and_min(0.7, 0.5) # Returns: 0.5
```

```
fuzzy_and_product(a, b)
```

Product t-norm.

Returns: `float` | `ndarray` - `a * b`

OR Operators (S-norms)

```
fuzzy_or_max(a, b)
```

Maximum s-norm (standard fuzzy OR).

Parameters: - `a`, `b` (float | ndarray): Membership degrees

Returns: `float` | `ndarray` - `max(a, b)`

Example:

```
from fuzzy_systems.core import fuzzy_or_max
result = fuzzy_or_max(0.7, 0.5) # Returns: 0.7
```

```
fuzzy_or_probabilistic(a, b)
```

Probabilistic s-norm.

Returns: `float` | `ndarray` - `a + b - a*b`

NOT Operators

```
fuzzy_not(a)
```

Standard fuzzy negation.

Parameters: - `a` (float | ndarray): Membership degree(s)

Returns: `float` | `ndarray` - `1 - a`

Example:

```
from fuzzy_systems.core import fuzzy_not
result = fuzzy_not(0.7) # Returns: 0.3
```

8.4 Defuzzification

```
centroid(x, mu)
```

Centroid (center of gravity) defuzzification method.

Parameters: - `x` (ndarray): Universe of discourse values - `mu` (ndarray): Aggregated membership degrees

Returns: `float` - Crisp output value

Formula: $\int x \cdot \mu(x) dx / \int \mu(x) dx$

Example:

```
from fuzzy_systems.core import centroid
import numpy as np

x = np.linspace(0, 100, 500)
mu = np.maximum(0.5 * triangular(x, (0, 0, 50)),
               0.8 * triangular(x, (50, 100, 100)))

crisp_value = centroid(x, mu)
```

`bisector(x, mu)`

Bisector defuzzification method (divides area in half).

Parameters: - `x` (ndarray): Universe of discourse values - `mu` (ndarray): Aggregated membership degrees

Returns: `float` - Crisp output value

`mean_of_maximum(x, mu)`

Mean of Maximum (MOM) defuzzification method.

Parameters: - `x` (ndarray): Universe of discourse values - `mu` (ndarray): Aggregated membership degrees

Returns: `float` - Mean of x values where μ is maximum

8.5 Complete Example

```
import numpy as np
from fuzzy_systems.core import (
    LinguisticVariable,
    triangular,
    fuzzy_and_min,
    fuzzy_or_max,
    fuzzy_not
)

# Create linguistic variable
temperature = LinguisticVariable("temperature", (0, 50))
temperature.add_term("cold", "trapezoidal", (0, 0, 10, 20))
temperature.add_term("warm", "triangular", (15, 25, 35))
temperature.add_term("hot", "trapezoidal", (30, 40, 50, 50))

# Fuzzify a value
current_temp = 28
degrees = temperature.fuzzify(current_temp)
print(degrees) # {'cold': 0.0, 'warm': 0.143, 'hot': 0.333}

# Apply fuzzy operations
mu_warm = degrees['warm']
mu_hot = degrees['hot']

comfort = fuzzy_and_min(mu_warm, fuzzy_not(mu_hot))
print(f"Comfort level: {comfort:.3f}")

# Plot the variable
temperature.plot()
```

8.6 See Also

- [Inference API](#) - Build complete fuzzy inference systems
- [User Guide: Fundamentals](#) - Learn fuzzy logic concepts
- [Examples](#) - Practical examples

9. Inference API Reference

The `fuzzy_systems.inference` module provides complete fuzzy inference systems:

- **MamdaniSystem**: Classic fuzzy inference with linguistic outputs
- **SugenoSystem**: TSK systems with functional outputs (order 0 and 1)

9.1 MamdaniSystem

Classic Mamdani fuzzy inference system with linguistic rule base.

Constructor

```
MamdaniSystem(name="Mamdani FIS", t_norm='min', s_norm='max',
               implication='min', aggregation='max', defuzz_method='centroid')
```

Parameters:

- `name` (str): System name (default: "Mamdani FIS")
- `t_norm` (str): T-norm for AND operation: 'min', 'product', etc. (default: 'min')
- `s_norm` (str): S-norm for OR operation: 'max', 'probabilistic', etc. (default: 'max')
- `implication` (str): Implication method: 'min' (Mamdani), 'product' (Larsen) (default: 'min')
- `aggregation` (str): Aggregation method: 'max', 'sum', 'probabilistic' (default: 'max')
- `defuzz_method` (str): Defuzzification: 'centroid', 'bisector', 'mom', 'som', 'lom' (default: 'centroid')

Example:

```
from fuzzy_systems import MamdaniSystem

system = MamdaniSystem(name="Temperature Control")
```

Methods

`.add_input(name, universe)`

Add an input variable to the system.

Parameters: - `name` (str): Variable name (e.g., "temperature") - `universe` (tuple): Range (min, max) of the variable

Returns: `LinguisticVariable` - The created variable

Example:

```
system.add_input('temperature', (0, 40))
system.add_input('humidity', (0, 100))
```

Alternative (pass `LinguisticVariable`):

```
from fuzzy_systems.core import LinguisticVariable

temp_var = LinguisticVariable('temperature', (0, 40))
system.add_input(temp_var)
```

`.add_output(name, universe)`

Add an output variable to the system.

Parameters: - `name` (str): Variable name (e.g., "fan_speed") - `universe` (tuple): Range (min, max) of the variable

Returns: `LinguisticVariable` - The created variable

Example:

```
system.add_output('fan_speed', (0, 100))
```

`.add_term(variable_name, term_name, mf_type, params, mf_func=None)`

Add a fuzzy term to an input or output variable.

Parameters: - `variable_name` (str): Name of the variable (input or output) - `term_name` (str): Name of the term (e.g., "cold", "hot") - `mf_type` (str): Membership function type: 'triangular', 'trapezoidal', 'gaussian', etc. - `params` (tuple): Parameters for the membership function - `mf_func` (callable, optional): Custom membership function

Example:

```
# Add terms to input
system.add_term('temperature', 'cold', 'triangular', (0, 0, 20))
system.add_term('temperature', 'warm', 'triangular', (10, 20, 30))
system.add_term('temperature', 'hot', 'triangular', (20, 40, 40))

# Add terms to output
system.add_term('fan_speed', 'slow', 'triangular', (0, 0, 50))
system.add_term('fan_speed', 'fast', 'triangular', (50, 100, 100))
```

`.add_rule(rule_dict, operator='AND', weight=1.0)`

Add a single fuzzy rule to the system.

Parameters: - `rule_dict` (dict | list | tuple): Rule specification -
`operator` (str): 'AND' or 'OR' (default: 'AND') - `weight` (float):
 Rule weight in [0, 1] (default: 1.0)

Rule Formats:

Format 1 - Dictionary (Recommended):

```
system.add_rule({
    'temperature': 'cold',
    'humidity': 'high',
    'fan_speed': 'slow'
})

# With operator and weight
system.add_rule({
    'temperature': 'hot',
    'humidity': 'low',
    'fan_speed': 'fast',
    'operator': 'OR',
    'weight': 0.9
})
```

Format 2 - Tuple (Compact):

```
# (input1_term, input2_term, ..., output1_term, ...)
system.add_rule(('cold', 'high', 'slow'))
system.add_rule(('hot', 'low', 'fast'))
```

Format 3 - Tuple with indices:

```
# Use term indices instead of names
system.add_rule((0, 2, 0)) # First term of each variable
```

```
.add_rules(rules_list, operator='AND', weight=1.0)
```

Add multiple rules at once.

Parameters: - `rules_list` (list): List of rules in any supported format -
`operator` (str): Default operator for all rules - `weight` (float): Default
 weight for all rules

Example:

```
# Using tuples (simple)
system.add_rules([
    ('cold', 'slow'),
    ('warm', 'medium'),
    ('hot', 'fast')
])

# Using dictionaries (explicit)
system.add_rules([
    {'temperature': 'cold', 'fan_speed': 'slow'},
    {'temperature': 'hot', 'fan_speed': 'fast', 'operator': 'OR'}
])

# Mixed formats
system.add_rules([
    ('cold', 'slow'),
    {'temperature': 'hot', 'fan_speed': 'fast', 'weight': 0.8}
])
```

```
.evaluate(inputs, **kwargs)
```

Evaluate the fuzzy system for given inputs.

Parameters: - `inputs` (dict | list | tuple | scalar): Input values in various
 formats - `**kwargs`: Alternative way to pass inputs as keyword arguments

Returns: `dict` - Output values: {output_name: crisp_value}

Input Formats:

Format 1 - Dictionary:

```
result = system.evaluate({'temperature': 25, 'humidity': 60})
```

Format 2 - Keyword arguments:

```
result = system.evaluate(temperature=25, humidity=60)
```

Format 3 - List/tuple (order matches variable addition order):

```
result = system.evaluate([25, 60])
```

Format 4 - Scalar (for single input):

```
result = system.evaluate(25)
```

Example:

```
# Evaluate
result = system.evaluate(temperature=25)
print(f"Fan speed: {result['fan_speed']:.1f}%")
# Output: Fan speed: 62.5%
```

```
.evaluate_detailed(inputs, **kwargs)
```

Evaluate with detailed intermediate results.

Parameters: - `inputs` (dict | list | tuple | scalar): Input values

Returns: `dict` - Detailed results:

```
{
    'inputs': {...},          # Fuzzified inputs
    'rule_activations': [...], # Activation level of each rule
    'aggregated': {...},     # Aggregated output MFs
    'outputs': {...}         # Final crisp outputs
}
```

Example:

```
details = system.evaluate_detailed(temperature=25)

print("Input fuzzification:")
print(details['inputs'])
# {'temperature': {'cold': 0.25, 'warm': 0.5, 'hot': 0.25}}

print("\nRule activations:")
for i, activation in enumerate(details['rule_activations']):
    print(f"Rule {i+1}: {activation:.3f}")

print("\nFinal output:")
print(details['outputs'])
# {'fan_speed': 62.5}
```

```
.plot_variables(var_names=None, figsize=(12, 8), show=True)
```

Plot membership functions of variables.

Parameters: - `var_names` (list, optional): List of variable names to plot. If
 None, plots all. - `figsize` (tuple): Figure size (default: (12, 8)) - `show`
 (bool): Whether to call `plt.show()` (default: True)

Returns: tuple - (fig, axes) matplotlib objects

Example:

```
# Plot all variables
system.plot_variables()

# Plot specific variables
system.plot_variables(['temperature', 'fan_speed'])

# Get figure for customization
fig, axes = system.plot_variables(show=False)
axes[0].set_title("My Custom Title")
fig.savefig('variables.png')
```

`.plot_rule_matrix(figsize=(10, 8), cmap='RdYlGn', show=True)`

Plot rule matrix as a heatmap (for 2-input systems).

Parameters: - `figsize` (tuple): Figure size (default: (10, 8)) - `cmap` (str): Colormap name (default: 'RdYlGn') - `show` (bool): Whether to call `plt.show()` (default: True)

Returns: tuple - (fig, ax) matplotlib objects

Example:

```
system.plot_rule_matrix()
```

`.export_rules(filename, format='txt')`

Export rules to a file.

Parameters: - `filename` (str): Output file path - `format` (str): Format: 'txt', 'json', 'csv' (default: 'txt')

Example:

```
system.export_rules('rules.txt', format='txt')
system.export_rules('rules.json', format='json')
system.export_rules('rules.csv', format='csv')
```

`.import_rules(filename, format='txt')`

Import rules from a file.

Parameters: - `filename` (str): Input file path - `format` (str): Format: 'txt', 'json', 'csv' (default: 'txt')

Example:

```
system.import_rules('rules.json', format='json')
```

`.save(filename)`

Save complete system (variables + rules) to a file.

Parameters: - `filename` (str): Output file path (typically .pkl or .json)

Example:

```
system.save('my_system.pkl')
```

`.load(filename)`

Load complete system from a file (class method).

Parameters: - `filename` (str): Input file path

Returns: MamdaniSystem - Loaded system

Example:

```
system = MamdaniSystem.load('my_system.pkl')
```

9.2 SugenoSystem

Sugeno (TSK) fuzzy inference system with functional outputs.

Constructor

```
SugenoSystem(name="Sugeno FIS", t_norm='min', s_norm='max')
```

Parameters:

- `name` (str): System name (default: "Sugeno FIS")
- `t_norm` (str): T-norm for AND operation (default: 'min')
- `s_norm` (str): S-norm for OR operation (default: 'max')

Example:

```
from fuzzy_systems import SugenoSystem

system = SugenoSystem(name="Nonlinear Model")
```

Methods

Most methods are identical to

`MamdaniSystem: .add_input(), .add_term(), .evaluate(), etc.`

Key Differences

`.add_output(name, order=0)`

Add output variable with functional definition.

Parameters: - `name` (str): Variable name - `order` (int): Output order: -
`0`: Constant output (zero-order Sugeno) - `1`: Linear function (first-order Sugeno)

Example:

```
# Zero-order (constants)
system.add_output('y', order=0)

# First-order (linear functions)
system.add_output('y', order=1)
```

`.add_rule()` WITH FUNCTIONAL OUTPUTS

For Sugeno systems, consequents are numbers (order 0) or coefficient lists (order 1).

Order 0 - Constant outputs:

```
system.add_rules([
    ('low', 2.0),      # IF x is low THEN y = 2.0
    ('high', 8.0),     # IF x is high THEN y = 8.0
])
```

Order 1 - Linear outputs:

```
# For y = a*x + b, provide (a, b)
system.add_rules([
    ('low', 2.0, 1.0),  # IF x is low THEN y = 2.0*x + 1.0
    ('high', 0.5, 3.0)  # IF x is high THEN y = 0.5*x + 3.0
])
```

Multiple inputs (order 1):

```
# For y = a*x1 + b*x2 + c, provide (a, b, c)
system.add_rules([
    ('low', 'low', 1.0, 0.5, 2.0),  # y = 1.0*x1 + 0.5*x2 + 2.0
    ('high', 'high', 2.0, 1.0, 0.0)  # y = 2.0*x1 + 1.0*x2 + 0.0
])
```

9.3 Complete Examples

Example 1: Mamdani Tipping System

```
from fuzzy_systems import MamdaniSystem

# Create system
system = MamdaniSystem(name="Tipping System")

# Add inputs
system.add_input('service', (0, 10))
system.add_input('food', (0, 10))

# Add output
system.add_output('tip', (0, 25))

# Add terms to inputs
system.add_term('service', 'poor', 'triangular', (0, 0, 5))
system.add_term('service', 'good', 'triangular', (0, 5, 10))
system.add_term('service', 'excellent', 'triangular', (5, 10, 10))

system.add_term('food', 'poor', 'triangular', (0, 0, 5))
system.add_term('food', 'good', 'triangular', (0, 5, 10))
system.add_term('food', 'delicious', 'triangular', (5, 10, 10))

# Add terms to output
system.add_term('tip', 'low', 'triangular', (0, 0, 13))
system.add_term('tip', 'medium', 'triangular', (0, 13, 25))
system.add_term('tip', 'high', 'triangular', (13, 25, 25))

# Add rules
system.add_rules([
    ('service': 'poor', 'food': 'poor', 'tip': 'low'),
    ('service': 'good', 'food': 'good', 'tip': 'medium'),
    ('service': 'excellent', 'food': 'delicious', 'tip': 'high'),
])

# Evaluate
result = system.evaluate(service=7, food=8)
print(f"Tip: {result['tip']:.1f}%")

# Visualize
system.plot_variables()
system.plot_rule_matrix()
```

```
# Add input
system.add_input('x', (0, 10))
system.add_term('x', 'low', 'triangular', (0, 0, 5))
system.add_term('x', 'medium', 'triangular', (0, 5, 10))
system.add_term('x', 'high', 'triangular', (5, 10, 10))

# Add output (order 0 = constant)
system.add_output('y', order=0)

# Add rules with constant outputs
system.add_rules([
    ('low', 2.0),      # IF x is low THEN y = 2.0
    ('medium', 5.0),   # IF x is medium THEN y = 5.0
    ('high', 8.0),     # IF x is high THEN y = 8.0
])

# Evaluate
result = system.evaluate(x=6)
print(f"y = {result['y']:.2f}")
```

Example 3: Sugeno First-Order

```
from fuzzy_systems import SugenoSystem

# Create system
system = SugenoSystem()

# Add input
system.add_input('x', (0, 10))
system.add_term('x', 'low', 'triangular', (0, 0, 5))
system.add_term('x', 'high', 'triangular', (5, 10, 10))

# Add output (order 1 = linear function)
system.add_output('y', order=1)

# Add rules with linear functions: y = a*x + b
system.add_rules([
    ('low', 2.0, 1.0),  # IF x is low THEN y = 2.0*x + 1.0
    ('high', 0.5, 3.0)  # IF x is high THEN y = 0.5*x + 3.0
])

# Evaluate
result = system.evaluate(x=7)
print(f"y = {result['y']:.2f}")

# For x=7:
# - mu_low = 0.0, mu_high = 0.4
# - y_low = 2.0*7 + 1.0 = 15.0
# - y_high = 0.5*7 + 3.0 = 6.5
# - y_final = (0.0*15.0 + 0.4*6.5) / (0.0 + 0.4) = 6.5
```

Example 2: Sugeno Zero-Order

```
from fuzzy_systems import SugenoSystem

# Create system
system = SugenoSystem()
```

Example 4: Multiple Inputs & Complex Rules

```
system = MamdaniSystem()

# Multiple inputs
system.add_input('temp', (0, 40))
system.add_input('humidity', (0, 100))
system.add_output('comfort', (0, 10))

# Add terms
for var in ['temp', 'humidity']:
    system.add_term(var, 'low', 'triangular', (0, 0, 50))
    system.add_term(var, 'high', 'triangular', (50, 100, 100))

system.add_term('comfort', 'uncomfortable', 'triangular', (0, 0, 5))
system.add_term('comfort', 'comfortable', 'triangular', (5, 10, 10))

# Rules with OR operator
```

```
system.add_rules([
    {
        'temp': 'high',
        'humidity': 'high',
        'comfort': 'uncomfortable',
        'operator': 'OR',
        'weight': 0.9
    },
    {
        'temp': 'low',
        'humidity': 'low',
        'comfort': 'comfortable',
        'operator': 'AND'
    }
])

result = system.evaluate(temp=30, humidity=70)
print(f"Comfort: {result['comfort']:.1f}/10")
```

9.4 See Also

- [Core API](#) - Membership functions, fuzzy sets, operators
- [Learning API](#) - Automatic rule generation and optimization
- [User Guide: Inference](#) - Detailed tutorials
- [Examples](#) - Interactive notebooks

10. Learning API Reference

The `fuzzy_systems.learning` module provides algorithms for automatic rule generation and system optimization:

- **WangMendelLearning:** Automatic rule generation from data (single-pass algorithm)
- **ANFIS:** Adaptive Neuro-Fuzzy Inference System (gradient-based learning)
- **MamdaniLearning:** Mamdani system optimization with gradients and metaheuristics
- **Metaheuristics:** PSO, Differential Evolution, Genetic Algorithms

10.1 WangMendelLearning

Automatic fuzzy rule generation using the Wang-Mendel algorithm (1992).

Reference: Wang, L. X., & Mendel, J. M. (1992). "Generating fuzzy rules by learning from examples." *IEEE Transactions on Systems, Man, and Cybernetics*, 22(6), 1414-1427.

Algorithm Steps

1. **Partition** variable domains (use existing MFs)
2. **Generate** candidate rules from each data sample
3. **Assign** degree to each rule based on membership strengths
4. **Resolve** conflicts (keep rule with highest degree)
5. **Create** final fuzzy system with learned rules

Constructor

```
WangMendelLearning(system, X, y, task='auto',
                    scale_classification=True, verbose_init=False)
```

Parameters:

- `system` (MamdaniSystem): Pre-configured system with variables and terms (NO rules yet)
- `X` (ndarray): Input data, shape `(n_samples, n_features)`
- `y` (ndarray): Output data, shape `(n_samples,)` or `(n_samples, n_outputs)`
- `task` (str): 'auto' (detect), 'regression', or 'classification' (default: 'auto')
- `scale_classification` (bool): Scale classification outputs to [0, 1] (default: True)
- `verbose_init` (bool): Print initialization info (default: False)

Example:

```
import numpy as np
from fuzzy_systems import MamdaniSystem
from fuzzy_systems.learning import WangMendelLearning

# Prepare data
X_train = np.random.uniform(0, 10, (100, 2))
y_train = np.sin(X_train[:, 0]) + np.cos(X_train[:, 1])

# Create base system (with variables and terms, NO rules)
system = MamdaniSystem()
system.add_input('x1', (0, 10))
```

```
system.add_input('x2', (0, 10))
system.add_output('y', (-2, 2))

# Add partitions (e.g., 5 terms per variable)
for var in ['x1', 'x2']:
    for i in range(5):
        center = i * 2.5
        system.add_term(var, f'term_{i}', 'triangular',
                        (max(0, center-2.5), center, min(10, center+2.5)))

# Similar for output
for i in range(5):
    center = -2 + i * 1.0
    system.add_term('y', f'out_{i}', 'triangular',
                    (max(-2, center-1), center, min(2, center+1)))

# Learn rules from data
wm = WangMendelLearning(system, X_train, y_train)
wm.fit(verbose=True)
```

Methods

`.fit(verbose=False)`

Generate fuzzy rules from the training data.

Parameters: - `verbose` (bool): Print progress information (default: False)

Returns: `MamdaniSystem` - The trained fuzzy system

Example:

```
trained_system = wm.fit(verbose=True)
```

Output (verbose=True):

```
☑ Wang-Mendel Algorithm Starting...
  ✓ Generated 100 candidate rules
  ✓ Resolved 23 conflicts
  ✓ Final rule base: 77 rules
☑ Wang-Mendel training complete!
```

.predict(X)

Predict outputs for new inputs.

Parameters: - `X` (ndarray): Input data, shape

(`n_samples`, `n_features`)

Returns: ndarray - Predicted outputs

For Regression:

```
y_pred = wm.predict(X_test) # Shape: (n_samples, n_outputs)
```

For Classification:

```
y_pred_classes = wm.predict(X_test) # Shape: (n_samples,) - class
indices
```

.predict_proba(X) (Classification only)

Predict class probabilities.

Parameters: - `X` (ndarray): Input data

Returns: ndarray - Probability matrix, shape (`n_samples`, `n_classes`)

Example:

```
proba = wm.predict_proba(X_test)
print(proba[0]) # [0.1, 0.7, 0.2] for 3 classes
```

.get_training_stats()

Get statistics about the training process.

Returns: dict - Training statistics:

```
{
  'candidate_rules': 100,    # Rules generated from data
  'final_rules': 77,        # Rules after conflict resolution
  'conflicts_resolved': 23, # Number of conflicts
  'task': 'regression'      # Task type
}
```

Example:

```
stats = wm.get_training_stats()
print(f"Generated {stats['candidate_rules']} rules")
print(f"Final: {stats['final_rules']} rules")
```

Complete Example: Regression

```
import numpy as np
from fuzzy_systems import MamdaniSystem
from fuzzy_systems.learning import WangMendelLearning

# Generate nonlinear data
X_train = np.linspace(0, 2*np.pi, 50).reshape(-1, 1)
y_train = np.sin(X_train) + 0.1*X_train

# Create system with 11 partitions
system = MamdaniSystem()
```

```
system.add_input('x', (0, 2*np.pi))
system.add_output('y', (-2, 2))

# Add 11 triangular terms to input and output
n_terms = 11
for i in range(n_terms):
    # Input terms
    center_x = i * (2*np.pi) / (n_terms - 1)
    width = (2*np.pi) / (n_terms - 1)
    system.add_term('x', f'x_{i}', 'triangular',
                    (max(0, center_x - width),
                     center_x,
                     min(2*np.pi, center_x + width)))

    # Output terms
    center_y = -2 + i * 4 / (n_terms - 1)
    width_y = 4 / (n_terms - 1)
    system.add_term('y', f'y_{i}', 'triangular',
                    (max(-2, center_y - width_y),
                     center_y,
                     min(2, center_y + width_y)))

# Train Wang-Mendel
wm = WangMendelLearning(system, X_train, y_train)
wm.fit(verbose=True)

# Predict
X_test = np.linspace(0, 2*np.pi, 200).reshape(-1, 1)
y_pred = wm.predict(X_test)

# Evaluate
from sklearn.metrics import mean_squared_error, r2_score
y_true = np.sin(X_test) + 0.1*X_test
mse = mean_squared_error(y_true, y_pred)
r2 = r2_score(y_true, y_pred)

print(f"MSE: {mse:.4f}")
print(f"R²: {r2:.4f}")
```

Complete Example: Classification

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from fuzzy_systems import MamdaniSystem
from fuzzy_systems.learning import WangMendelLearning

# Load Iris dataset
iris = load_iris()
X = iris.data[:, [2, 3]] # Use petal length and width
y = iris.target

# One-hot encode targets
encoder = OneHotEncoder(sparse_output=False)
y_onehot = encoder.fit_transform(y.reshape(-1, 1))

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y_onehot, test_size=0.3, random_state=42
)

# Create system with 3 terms per input, 1 output per class
system = MamdaniSystem()
system.add_input('petal_length', (X[:, 0].min(), X[:, 0].max()))
system.add_input('petal_width', (X[:, 1].min(), X[:, 1].max()))

# Add 3 binary outputs (one per class)
for i in range(3):
    system.add_output(f'class_{i}', (0, 1))

# Add terms (3 per variable)
for var in ['petal_length', 'petal_width']:
    universe = system.input_variables[var].universe
    for i in range(3):
        center = universe[0] + i * (universe[1] - universe[0]) / 2
        width = (universe[1] - universe[0]) / 3
        system.add_term(var, f'term_{i}', 'triangular',
                        (max(universe[0], center - width),
                         center,
                         min(universe[1], center + width)))

# Add output terms (2 per class: 0 and 1)
for i in range(3):
    system.add_term(f'class_{i}', 'no', 'triangular', (0, 0, 0.5))
    system.add_term(f'class_{i}', 'yes', 'triangular', (0.5, 1, 1))

# Train
wm = WangMendelLearning(system, X_train, y_train,
```

```
task='classification')
wm.fit(verbose=True)

# Predict
y_pred_classes = wm.predict(X_test)
y_pred_proba = wm.predict_proba(X_test)

# Evaluate
from sklearn.metrics import accuracy_score, classification_report
y_test_classes = y_test.argmax(axis=1)
```

```
accuracy = accuracy_score(y_test_classes, y_pred_classes)

print(f"Accuracy: {accuracy:.2%}")
print("\nClassification Report:")
print(classification_report(y_test_classes, y_pred_classes,
                           target_names=iris.target_names))
```

10.2 ANFIS

Adaptive Neuro-Fuzzy Inference System with gradient-based learning.

Constructor

```
ANFIS(n_inputs, n_terms, n_outputs=1, mf_type='gaussian')
```

Parameters:

- `n_inputs` (int): Number of input variables
- `n_terms` (int): Number of membership functions per input
- `n_outputs` (int): Number of outputs (default: 1)
- `mf_type` (str): Membership function type: 'gaussian', 'bell' (default: 'gaussian')

Example:

```
from fuzzy_systems.learning import ANFIS

anfis = ANFIS(n_inputs=2, n_terms=3, n_outputs=1)
```

Methods

```
.fit(X, y, epochs=100, learning_rate=0.01, batch_size=None,
validation_split=0.0, early_stopping=False, patience=10,
lyapunov_check=True, verbose=True)
```

Train ANFIS using gradient descent with backpropagation.

Parameters:

- `X` (ndarray): Input data, shape (n_samples, n_inputs)
- `y` (ndarray): Output data, shape (n_samples, n_outputs) or (n_samples,)
- `epochs` (int): Number of training epochs (default: 100)
- `learning_rate` (float): Learning rate (default: 0.01)
- `batch_size` (int, optional): Batch size for mini-batch gradient descent. If None, uses full batch
- `validation_split` (float): Fraction of data for validation (default: 0.0)
- `early_stopping` (bool): Stop if validation loss doesn't improve (default: False)
- `patience` (int): Epochs to wait before early stopping (default: 10)
- `lyapunov_check` (bool): Monitor Lyapunov stability (default: True)
- `verbose` (bool): Print training progress (default: True)

Returns: dict - Training history

Example:

```
history = anfis.fit(
    X_train, y_train,
    epochs=50,
    learning_rate=0.01,
    validation_split=0.2,
    early_stopping=True,
    verbose=True
)
```

Output (verbose=True):

```
Epoch 10/50 - Loss: 0.0234 - Val Loss: 0.0251 - Lyapunov: 0.98
Epoch 20/50 - Loss: 0.0156 - Val Loss: 0.0178 - Lyapunov: 0.99
...
✅ Training complete!
```

.predict(X)

Predict outputs for new inputs.

Parameters: - `X` (ndarray): Input data, shape (n_samples, n_inputs)

Returns: ndarray - Predictions, shape (n_samples, n_outputs)

Example:

```
y_pred = anfis.predict(X_test)
```

.get_training_history()

Get complete training history.

Returns: dict - History with keys:

```
{
    'epochs': [1, 2, 3, ...],
    'loss': [0.5, 0.3, 0.2, ...],
    'val_loss': [0.6, 0.4, 0.25, ...], # If validation_split > 0
    'lyapunov': [0.95, 0.97, 0.99, ...] # If lyapunov_check=True
}
```

Example:

```
history = anfis.get_training_history()

import matplotlib.pyplot as plt
plt.plot(history['epochs'], history['loss'], label='Training')
plt.plot(history['epochs'], history['val_loss'], label='Validation')
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

```
plt.legend()
plt.show()
```

Complete Example: ANFIS Regression

```
import numpy as np
from fuzzy_systems.learning import ANFIS

# Generate data
X_train = np.random.uniform(0, 10, (200, 2))
y_train = np.sin(X_train[:, 0]) + np.cos(X_train[:, 1])

X_test = np.random.uniform(0, 10, (50, 2))
y_true = np.sin(X_test[:, 0]) + np.cos(X_test[:, 1])

# Create and train ANFIS
anfis = ANFIS(n_inputs=2, n_terms=5, n_outputs=1, mf_type='gaussian')

history = anfis.fit(
    X_train, y_train,
    epochs=100,
    learning_rate=0.01,
    validation_split=0.2,
    early_stopping=True,
    patience=10,
    verbose=True
)
```

```
# Predict
y_pred = anfis.predict(X_test)

# Evaluate
from sklearn.metrics import mean_squared_error, r2_score
mse = mean_squared_error(y_true, y_pred)
r2 = r2_score(y_true, y_pred)

print(f"MSE: {mse:.4f}")
print(f"R²: {r2:.4f}")

# Plot training curve
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)
plt.plot(history['epochs'], history['loss'], label='Train')
plt.plot(history['epochs'], history['val_loss'], label='Validation')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Learning Curve')

plt.subplot(1, 2, 2)
plt.plot(history['epochs'], history['lyapunov'])
plt.xlabel('Epoch')
plt.ylabel('Lyapunov Stability')
plt.title('Stability Monitoring')
plt.tight_layout()
plt.show()
```

10.3 MamdaniLearning

Optimize Mamdani systems using gradients or metaheuristics.

Constructor

```
MamdaniLearning(system=None, X=None, y=None)
```

Parameters:

- `system` (MamdaniSystem, optional): Existing Mamdani system to optimize
- `X` (ndarray, optional): Training input data
- `y` (ndarray, optional): Training output data

Example:

```
from fuzzy_systems.learning import MamdaniLearning

# Create from existing system
learner = MamdaniLearning.from_mamdani(system, X_train, y_train)

# Or create new
learner = MamdaniLearning()
# ... configure ...
```

Class Methods

```
.from_mamdani(system, X, y)
```

Create MamdaniLearning from existing MamdaniSystem.

Parameters: - `system` (MamdaniSystem): Existing fuzzy system - `X` (ndarray): Training inputs - `y` (ndarray): Training outputs

Returns: `MamdaniLearning` - Learner instance

Example:

```
learner = MamdaniLearning.from_mamdani(system, X_train, y_train)
```

Methods

```
.fit(X, y, method='gradient', epochs=100, learning_rate=0.01,
**kwargs)
```

Optimize the fuzzy system.

Parameters:

- `X` (ndarray): Training input data
- `y` (ndarray): Training output data
- `method` (str): Optimization method:
 - `'gradient'`: Gradient descent
 - `'pso'`: Particle Swarm Optimization
 - `'de'`: Differential Evolution
 - `'ga'`: Genetic Algorithm
- `epochs` (int): Number of iterations (default: 100)
- `learning_rate` (float): Learning rate for gradient (default: 0.01)
- `**kwargs`: Method-specific parameters

Gradient-specific kwargs: - `batch_size` (int): Batch size for mini-batch - `momentum` (float): Momentum factor

PSO-specific kwargs: - `n_particles` (int): Number of particles (default: 30) - `inertia` (float): Inertia weight (default: 0.7) - `cognitive` (float): Cognitive parameter (default: 1.5) - `social` (float): Social parameter (default: 1.5)

DE-specific kwargs: - `population_size` (int): Population size (default: 50) - `mutation_factor` (float): Mutation factor F (default: 0.8) - `crossover_prob` (float): Crossover probability (default: 0.9)

GA-specific kwargs: - `population_size` (int): Population size (default: 50) - `mutation_rate` (float): Mutation rate (default: 0.1) - `crossover_rate` (float): Crossover rate (default: 0.8)

Example:

```
# Gradient descent
learner.fit(X_train, y_train, method='gradient',
            epochs=100, learning_rate=0.01)

# PSO
learner.fit(X_train, y_train, method='pso',
            epochs=50, n_particles=30)

# Differential Evolution
learner.fit(X_train, y_train, method='de',
            epochs=100, population_size=50)
```

.predict(X)

Predict outputs using the optimized system.

Parameters: - `X` (ndarray): Input data

Returns: `ndarray` - Predictions

.to_mamdani()

Convert back to MamdaniSystem.

Returns: `MamdaniSystem` - Optimized fuzzy system

Example:

```
optimized_system = learner.to_mamdani()
optimized_system.plot_variables()
```

Complete Example: Optimization with PSO

```
import numpy as np
from fuzzy_systems import MamdaniSystem
from fuzzy_systems.learning import MamdaniLearning

# Generate data
X_train = np.linspace(-5, 5, 100).reshape(-1, 1)
y_train = -2 * X_train + 5 + np.random.normal(0, 0.5, X_train.shape)

# Create initial system
system = MamdaniSystem()
system.add_input('x', (-5, 5))
system.add_output('y', (-15, 15))

# Add terms with suboptimal initial parameters
system.add_term('x', 'low', 'triangular', (-5, -2, 1))
system.add_term('x', 'high', 'triangular', (-1, 2, 5))
system.add_term('y', 'low', 'triangular', (-15, -7, 1))
system.add_term('y', 'high', 'triangular', (-1, 7, 15))

# Initial rules
system.add_rules([(('low', 'high'), ('high', 'low'))])

# Create learner
learner = MamdaniLearning.from_mamdani(system, X_train, y_train)

# Optimize with PSO
history = learner.fit(
    X_train, y_train,
    method='pso',
    epochs=100,
    n_particles=30,
    verbose=True
)

# Predict
y_pred = learner.predict(X_train)

# Evaluate
from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_train, y_pred)
print(f"MSE after optimization: {mse:.4f}")

# Get optimized system
optimized_system = learner.to_mamdani()
optimized_system.save('optimized_system.pkl')
```

10.4 Metaheuristics

Direct access to optimization algorithms.

PSO

Particle Swarm Optimization.

```
from fuzzy_systems.learning import PSO

optimizer = PSO(
    objective_func,
    bounds,
    n_particles=30,
    max_iter=100,
    inertia=0.7,
    cognitive=1.5,
    social=1.5
)

best_params, best_cost = optimizer.optimize()
```

DE

Differential Evolution.

```
from fuzzy_systems.learning import DE

optimizer = DE(
    objective_func,
    bounds,
    population_size=50,
    max_iter=100,
    mutation_factor=0.8,
    crossover_prob=0.9
)

best_params, best_cost = optimizer.optimize()
```

GA

Genetic Algorithm.

```
from fuzzy_systems.learning import GA
```

```
optimizer = GA(
    objective_func,
    bounds,
    population_size=50,
    max_iter=100,
    mutation_rate=0.1,
    crossover_rate=0.8
)

best_params, best_cost = optimizer.optimize()
```

10.5 Comparison Table

Method	Type	Speed	Accuracy	Best For
Wang-Mendel	Rule generation	⚡⚡⚡ Fast	☆☆ Good	Quick prototyping, interpretable rules
ANFIS	Neuro-fuzzy	⚡⚡ Medium	☆☆☆ Excellent	Precise approximation, differentiable problems
MamdaniLearning (Gradient)	Gradient	⚡⚡ Medium	☆☆☆ Excellent	Fine-tuning existing systems
MamdaniLearning (PSO)	Metaheuristic	⚡ Slow	☆☆☆ Excellent	Non-differentiable, global search
MamdaniLearning (DE)	Metaheuristic	⚡ Slow	☆☆☆ Excellent	Robust optimization, fewer parameters
MamdaniLearning (GA)	Metaheuristic	⚡ Slow	☆☆ Good	Discrete/combinatorial optimization

10.6 See Also

- [Core API](#) - Fuzzy sets and operators
- [Inference API](#) - Mamdani and Sugeno systems
- [Dynamics API](#) - Dynamic fuzzy systems
- [User Guide: Learning](#) - Detailed tutorials
- [Examples](#) - Interactive notebooks

11. Dynamics API Reference

The `fuzzy_systems.dynamics` module provides tools for dynamic systems with fuzzy uncertainty:

- **FuzzyODE**: Solve ODEs with fuzzy parameters/initial conditions (α -level method)
- **PFuzzyDiscrete**: Discrete dynamical systems with fuzzy rule-based evolution
- **PFuzzyContinuous**: Continuous dynamical systems with fuzzy rule-based evolution

Reference: Barros, L. C., Bassanezi, R. C., & Lodwick, W. A. (2017). *A First Course in Fuzzy Logic, Fuzzy Dynamical Systems, and Biomathematics*.

11.1 Fuzzy ODEs

FuzzyNumber

Represent fuzzy numbers for use as initial conditions or parameters.

Class Methods

`.triangular(center, spread, name=None)`

Create a triangular fuzzy number.

Parameters: - `center` (float): Center (peak) of the triangle - `spread` (float): Half-width at base - `name` (str, optional): Name of the fuzzy number

Returns: `FuzzyNumber`

Example:

```
from fuzzy_systems.dynamics import FuzzyNumber

# Triangular:  $\mu(x)$  peaks at 10, base from 8 to 12
y0_fuzzy = FuzzyNumber.triangular(center=10, spread=2)
```

`.trapezoidal(a, b, c, d, name=None)`

Create a trapezoidal fuzzy number.

Parameters: - `a`, `b`, `c`, `d` (float): Trapezoidal parameters ($a \leq b \leq c \leq d$) - `name` (str, optional): Name of the fuzzy number

Returns: `FuzzyNumber`

`.gaussian(mean, sigma, name=None)`

Create a Gaussian fuzzy number.

Parameters: - `mean` (float): Mean (center) - `sigma` (float): Standard deviation - `name` (str, optional): Name

Returns: `FuzzyNumber`

Constructor

```
FuzzyODESolver(ode_func, t_span, y0_fuzzy=None, params=None,
               alpha_levels=None, method='RK45', **options)
```

Parameters:

- `ode_func` (callable): ODE function $f(t, y, *params) \rightarrow dydt$
- `t_span` (tuple): Time interval (t_0, t_f)
- `y0_fuzzy` (list, optional): List of `FuzzyNumber` objects for initial conditions
- `params` (dict, optional): Crisp or fuzzy parameters: `{name: value_or_FuzzyNumber}`
- `alpha_levels` (list, optional): α -cut levels (default: `[0, 0.25, 0.5, 0.75, 1.0]`)
- `method` (str): Integration method: 'RK45', 'RK23', 'DOP853', 'Radau', 'BDF', 'LSODA' (default: 'RK45')
- `**options`: Additional options for `scipy.integrate.solve_ivp`

Example:

```
from fuzzy_systems.dynamics import FuzzyODESolver, FuzzyNumber
import numpy as np

# Define ODE:  $dy/dt = r*y*(1 - y/K)$  (Logistic growth)
def logistic(t, y, r, K):
    return r * y[0] * (1 - y[0] / K)

# Fuzzy initial condition
y0 = FuzzyNumber.triangular(center=10, spread=2)

# Fuzzy parameters
r_fuzzy = FuzzyNumber.triangular(center=1.0, spread=0.2)
K_fuzzy = FuzzyNumber.triangular(center=100, spread=10)

# Create solver
solver = FuzzyODESolver(
    ode_func=logistic,
    t_span=(0, 20),
    y0_fuzzy=[y0],
    params={'r': r_fuzzy, 'K': K_fuzzy},
    alpha_levels=[0, 0.5, 1.0]
)
```

FuzzyODESolver

Solve ordinary differential equations with fuzzy uncertainty.

Methods

`.solve(n_points=100, parallel=True, n_jobs=-1)`

Solve the fuzzy ODE using α -level method.

Parameters: - `n_points` (int): Number of time points (default: 100) -
`parallel` (bool): Use parallel processing (default: True) - `n_jobs` (int):
 Number of parallel jobs. -1 uses all CPUs (default: -1)

Returns: `FuzzySolution` - Solution object

Example:

```
solution = solver.solve(n_points=200, parallel=True)
```

`.plot_envelope(variables=None, alpha_colors=None, figsize=(12, 6), show=True)`

Plot fuzzy envelope showing uncertainty bands.

Parameters: - `variables` (list, optional): Variable indices to plot. If None,
 plots all - `alpha_colors` (dict, optional): Custom colors for α -levels:
`{alpha: color}` - `figsize` (tuple): Figure size (default: (12, 6)) -
`show` (bool): Whether to call `plt.show()` (default: True)

Returns: tuple - (fig, axes) matplotlib objects

Example:

```
solver.plot_envelope(
    variables=[0],
    alpha_colors={0: 'lightblue', 0.5: 'blue', 1.0: 'darkblue'})
```

```
import numpy as np
import matplotlib.pyplot as plt
from fuzzy_systems.dynamics import FuzzyODESolver, FuzzyNumber

# Define ODE: dy/dt = r*y*(1 - y/K)
def logistic(t, y, r, K):
    """Logistic growth model."""
    return r * y[0] * (1 - y[0] / K)

# Fuzzy initial condition: population around 10 ± 2
y0 = FuzzyNumber.triangular(center=10, spread=2)

# Fuzzy parameters
r = FuzzyNumber.triangular(center=0.5, spread=0.1) # Growth rate
K = FuzzyNumber.triangular(center=100, spread=10) # Carrying capacity

# Solve fuzzy ODE
solver = FuzzyODESolver(
    ode_func=logistic,
    t_span=(0, 30),
    y0_fuzzy=[y0],
    params={'r': r, 'K': K},
    alpha_levels=[0, 0.25, 0.5, 0.75, 1.0],
    method='RK45'
)

solution = solver.solve(n_points=200)

# Plot
solver.plot_envelope(
    variables=[0],
    figsize=(12, 6),
    alpha_colors={
        0: 'lightblue',
        0.5: 'blue',
        1.0: 'darkblue'
    }
)
plt.xlabel('Time')
plt.ylabel('Population')
plt.title('Fuzzy Logistic Growth with Uncertain Parameters')
plt.show()
```

Complete Example: Fuzzy Holling-Tanner (Predator-Prey)

```
from fuzzy_systems.dynamics import FuzzyODESolver, FuzzyNumber

# Holling-Tanner predator-prey model
def holling_tanner(t, y, r, K, a, b, c, d):
    """
    Predator-prey with Holling Type II functional response.

    y[0] = prey (x)
    y[1] = predator (z)
    """
    x, z = y
    dx = r * x * (1 - x/K) - (a*x*z)/(b + x)
    dz = c * z * (1 - d*z/x) if x > 0 else 0
    return [dx, dz]

# Initial conditions (fuzzy)
x0 = FuzzyNumber.triangular(center=40, spread=5)
z0 = FuzzyNumber.triangular(center=15, spread=3)

# Parameters (some fuzzy, some crisp)
params = {
    'r': FuzzyNumber.triangular(1.0, 0.1), # Fuzzy
    'K': 100, # Crisp
    'a': 1.0,
    'b': 10,
    'c': 0.5,
    'd': 0.1
}

# Solve
solver = FuzzyODESolver(
    ode_func=holling_tanner,
    t_span=(0, 100),
    y0_fuzzy=[x0, z0],
    params=params,
    alpha_levels=[0, 0.5, 1.0]
)

solution = solver.solve()

# Plot both variables
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

solver.plot_envelope(variables=[0], show=False)
```

FuzzySolution

Solution object returned by `FuzzyODESolver.solve()`.

Attributes

- `t` (ndarray): Time points
- `alpha_levels` (list): α -cut levels used
- `envelopes` (dict): Fuzzy envelopes: `{alpha: {'lower': array, 'upper': array}}`

Methods

`.plot(variables=None, **kwargs)`

Plot the fuzzy solution.

Parameters: - `variables` (list, optional): Variables to plot - `**kwargs`:
 Additional plotting options

Complete Example: Fuzzy Logistic Growth

```
ax1.set_title('Prey Population (x)')
ax1.set_xlabel('Time')
ax1.set_ylabel('Population')

solver.plot_envelope(variables=[1], show=False)
ax2.set_title('Predator Population (z)')
ax2.set_xlabel('Time')
ax2.set_ylabel('Population')
```

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

# Phase space plot
# (Would need to extract and plot lower/upper bounds in x-z plane)
```

11.2 p-Fuzzy Systems

Systems where evolution is defined by fuzzy rules instead of equations.

PFuzzyDiscrete

Discrete dynamical system with fuzzy rule-based evolution.

Discrete evolution: - **Absolute mode:** $x_{n+1} = x_n + f(x_n)$ - **Relative mode:** $x_{n+1} = x_n * (1 + f(x_n))$

Constructor

```
PFuzzyDiscrete(fis, mode='absolute', state_vars=None)
```

Parameters:

- `fis` (MamdaniSystem | SugenoSystem): Fuzzy inference system
- `mode` (str): Evolution mode: 'absolute' or 'relative' (default: 'absolute')
- `state_vars` (list, optional): State variable names. If None, uses all FIS inputs

Example:

```
from fuzzy_systems import MamdaniSystem
from fuzzy_systems.dynamics import PFuzzyDiscrete

# Create FIS with rules
fis = MamdaniSystem()
fis.add_input('prey', (0, 100))
fis.add_input('predator', (0, 100))
fis.add_output('var_pre', (-5, 5))
fis.add_output('var_predator', (-5, 5))

# Add terms and rules...
# (See examples below)

# Create p-fuzzy system
pfuzzy = PFuzzyDiscrete(
    fis=fis,
    mode='absolute',
    state_vars=['prey', 'predator']
)
```

Methods

```
.simulate(x0, n_steps, return_time=True)
```

Simulate the discrete system.

Parameters: - `x0` (dict | ndarray): Initial conditions: {var_name: value} or array - `n_steps` (int): Number of time steps - `return_time` (bool): If True, returns (time, trajectory). Otherwise, only trajectory (default: True)

Returns: tuple | ndarray - Time array and trajectory matrix, or just trajectory

Example:

```
# Using dictionary
time, trajectory = pfuzzy.simulate(
    x0={'prey': 50, 'predator': 40},
    n_steps=200
)

# Using array (order matches state_vars)
time, trajectory = pfuzzy.simulate(
    x0=[50, 40],
    n_steps=200
)
```

```
.plot_trajectory(variables=None, figsize=(12, 6), show=True)
```

Plot time evolution of state variables.

Parameters: - `variables` (list, optional): Variable names to plot. If None, plots all - `figsize` (tuple): Figure size - `show` (bool): Whether to call `plt.show()`

Returns: tuple - (fig, ax)

Example:

```
pfuzzy.plot_trajectory(variables=['prey', 'predator'])
```

```
.plot_phase_space(var_x, var_y, figsize=(8, 8), show=True)
```

Plot phase space (2D trajectory).

Parameters: - `var_x` (str): Variable for x-axis - `var_y` (str): Variable for y-axis - `figsize` (tuple): Figure size - `show` (bool): Whether to call `plt.show()`

Returns: tuple - (fig, ax)

Example:

```
pfuzzy.plot_phase_space('prey', 'predator')
```

```
.to_csv(filename, include_time=True)
```

Export trajectory to CSV file.

Parameters: - `filename` (str): Output file path - `include_time` (bool): Include time column (default: True)

Example:

```
pfuzzy.to_csv('trajectory.csv')
```

PFuzzyContinuous

Continuous dynamical system with fuzzy rule-based evolution.

Continuous evolution: - **Absolute mode:** $dx/dt = f(x)$ - **Relative mode:** $dx/dt = x * f(x)$

Constructor

```
PFuzzyContinuous(fis, mode='absolute', state_vars=None)
```

Parameters: Same as `PFuzzyDiscrete`

Methods

```
.simulate(x0, t_span, dt=0.1, method='RK4', return_time=True)
```

Simulate the continuous system.

Parameters: - `x0` (dict | ndarray): Initial conditions - `t_span` (tuple): Time interval (`t0`, `tf`) - `dt` (float): Time step for integration (default: `0.1`) - `method` (str): Integration method: `'Euler'`, `'RK4'` (default: `'RK4'`) - `return_time` (bool): Return time array (default: `True`)

Returns: tuple | ndarray - (time, trajectory) or just trajectory

Example:

```
time, trajectory = pfuzzy.simulate(
    x0={'prey': 50, 'predator': 40},
    t_span=(0, 100),
    dt=0.05,
    method='RK4'
)
```

Other methods

(`.plot_trajectory()`, `.plot_phase_space()`, `.to_csv()`) are identical to `PFuzzyDiscrete`.

Complete Example: Discrete Predator-Prey

```
from fuzzy_systems import MamdaniSystem
from fuzzy_systems.dynamics import PFuzzyDiscrete
import numpy as np
import matplotlib.pyplot as plt

# Create FIS
fis = MamdaniSystem(name="Predator-Prey Discrete")

# Define variables
fis.add_input('prey', (0, 100))
fis.add_input('predator', (0, 100))
fis.add_output('var_prey', (-2, 2))
fis.add_output('var_predator', (-2, 2))

# Add 4 linguistic terms per variable (Low, Medium-Low, Medium-High, High)
```

```
for var in ['prey', 'predator']:
    fis.add_term(var, 'B', 'gaussian', (0, 12)) # Low
    fis.add_term(var, 'MB', 'gaussian', (33, 12)) # Medium-Low
    fis.add_term(var, 'MA', 'gaussian', (67, 12)) # Medium-High
    fis.add_term(var, 'A', 'gaussian', (100, 12)) # High

# Add output terms (8 per variable: 4 positive, 4 negative)
lrg = 0.5
for out_var in ['var_prey', 'var_predator']:
    # Negative variations
    fis.add_term(out_var, 'A_n', 'trapezoidal', (-4*lrg, -4*lrg, -3*lrg, -2*lrg))
    fis.add_term(out_var, 'MA_n', 'triangular', (-3*lrg, -2*lrg, -lrg))
    fis.add_term(out_var, 'MB_n', 'triangular', (-2*lrg, -lrg, 0))
    fis.add_term(out_var, 'B_n', 'triangular', (-lrg, 0, 0))
    # Positive variations
    fis.add_term(out_var, 'B_p', 'triangular', (0, 0, lrg))
    fis.add_term(out_var, 'MB_p', 'triangular', (0, lrg, 2*lrg))
    fis.add_term(out_var, 'MA_p', 'triangular', (lrg, 2*lrg, 3*lrg))
    fis.add_term(out_var, 'A_p', 'trapezoidal', (2*lrg, 3*lrg, 4*lrg, 4*lrg))

# Define 16 rules (4x4 matrix)
rules = [
    # Prey=B (Low)
    ('B', 'B', 'MB_p', 'MB_n'), # Few prey, few predators - prey increase
    ('B', 'MB', 'B_p', 'MB_n'),
    ('B', 'MA', 'B_n', 'MA_n'),
    ('B', 'A', 'MB_n', 'A_n'),

    # Prey=MB (Medium-Low)
    ('MB', 'B', 'MA_p', 'B_n'),
    ('MB', 'MB', 'MB_p', 'B_n'),
    ('MB', 'MA', 'B_n', 'MB_n'),
    ('MB', 'A', 'MB_n', 'MA_n'),

    # Prey=MA (Medium-High)
    ('MA', 'B', 'MB_p', 'MA_p'),
    ('MA', 'MB', 'B_p', 'MB_p'),
    ('MA', 'MA', 'MB_n', 'B_p'),
    ('MA', 'A', 'MA_n', 'B_p'),

    # Prey=A (High)
    ('A', 'B', 'B_n', 'A_p'),
    ('A', 'MB', 'MB_n', 'MA_p'),
    ('A', 'MA', 'MA_n', 'MB_p'),
    ('A', 'A', 'A_n', 'B_p')
]

fis.add_rules(rules)

# Create p-fuzzy system
pfuzzy = PFuzzyDiscrete(
    fis=fis,
    mode='absolute',
    state_vars=['prey', 'predator']
)

# Simulate
time, trajectory = pfuzzy.simulate(
    x0={'prey': 50, 'predator': 40},
    n_steps=250
)

# Plot results
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# Time series
pfuzzy.plot_trajectory(show=False)
ax1 = plt.gca()
ax1.set_title('Population Dynamics')

# Phase space
pfuzzy.plot_phase_space('prey', 'predator', show=False)
ax2 = plt.gca()
ax2.set_title('Phase Space')

plt.tight_layout()
plt.show()

# Export
pfuzzy.to_csv('predator_prey_discrete.csv')
```

Complete Example: Continuous Population Growth

```
from fuzzy_systems import MamdaniSystem
from fuzzy_systems.dynamics import PFuzzyContinuous
```

```
# Create FIS for continuous population growth
fis = MamdaniSystem()

# Single variable: population
fis.add_input('population', (0, 150))
fis.add_output('growth_rate', (-5, 5))

# Terms: Low, Medium, High population
fis.add_term('population', 'low', 'triangular', (0, 0, 50))
fis.add_term('population', 'medium', 'triangular', (25, 75, 125))
fis.add_term('population', 'high', 'triangular', (100, 150, 150))

# Growth rates: negative, zero, positive
fis.add_term('growth_rate', 'negative', 'triangular', (-5, -2.5, 0))
fis.add_term('growth_rate', 'zero', 'triangular', (-1, 0, 1))
fis.add_term('growth_rate', 'positive', 'triangular', (0, 2.5, 5))

# Rules (logistic-like behavior)
fis.add_rules([
    ('low', 'positive'),      # Low population → growth
    ('medium', 'zero'),      # Medium population → equilibrium
    ('high', 'negative')     # High population → decline
])
```

```
# Create continuous p-fuzzy system
pfuzzy = PFuzzyContinuous(
    fis=fis,
    mode='absolute',
    state_vars=['population']
)

# Simulate
time, trajectory = pfuzzy.simulate(
    x0={'population': 10},
    t_span=(0, 50),
    dt=0.1,
    method='RK4'
)

# Plot
pfuzzy.plot_trajectory()
plt.axhline(y=75, color='r', linestyle='--', label='Equilibrium (~75)')
plt.legend()
plt.show()
```

11.3 Comparison: Fuzzy ODE vs p-Fuzzy

Feature	Fuzzy ODE	p-Fuzzy
Evolution	Mathematical equation: $dy/dt = f(t, y)$	Fuzzy rules: IF...THEN
Uncertainty	Parameters & initial conditions	Rule-based behavior
Method	α -level cuts + ODE solver	Direct FIS evaluation
Output	Fuzzy envelope (bands)	Deterministic trajectory
Best for	Models with known equations but uncertain params	Models defined by expert rules
Interpretability	Medium (equation-based)	High (linguistic rules)

11.4 See Also

- [Core API](#) - Fuzzy sets and membership functions
- [Inference API](#) - Build fuzzy systems for p-fuzzy
- [Learning API](#) - Learn fuzzy rules from data
- [User Guide: Dynamics](#) - Detailed tutorials
- [Examples](#) - Interactive notebooks

12. Examples Gallery

Explore practical examples through interactive Colab notebooks organized by topic and difficulty.

12.1 📖 Fundamentals (Beginner)

Learn the basics of fuzzy logic.

Membership Functions

 [Open in Colab](#)

What you'll learn: - Triangular, trapezoidal, gaussian, sigmoid functions - `FuzzySet` and `LinguisticVariable` classes - Fuzzification process - Fuzzy operators (AND, OR, NOT)

Estimated time: 45-60 min

Thermal Comfort System

 [Open in Colab](#)

What you'll learn: - Model multiple variables (temperature + humidity) - Combine variables with fuzzy operators - Implement simple IF-THEN rules - Create 2D comfort maps

Estimated time: 40-50 min

12.2 🧩 Inference Systems (Intermediate)

Build complete fuzzy inference systems.

Mamdani Tipping System

 [Open in Colab](#)

What you'll learn: - Complete Mamdani inference system - 5 Mamdani steps: fuzzification → rules → implication → aggregation → defuzzification - Multiple inputs (service + food quality) - 3D control surfaces

Estimated time: 60-75 min

Sugeno First-Order System

 [Open in Colab](#)

What you'll learn: - Sugeno with linear output functions: $y = ax + b$ - Function approximation - Comparison with zero-order

Estimated time: 40-50 min

Sugeno Zero-Order System

 [Open in Colab](#)

What you'll learn: - Sugeno system with constant outputs - Difference between Mamdani and Sugeno - Weighted average defuzzification

Estimated time: 45-60 min

Voting Prediction

 [Open in Colab](#)

What you'll learn: - Real-world application - Complex rule base - Multiple inputs (income + education)

Estimated time: 50-70 min

12.3 🧠 Learning & Optimization (Advanced)

Automatic rule generation and system optimization.

Wang-Mendel: Nonlinear Approximation

 [Open in Colab](#)

What you'll learn: - Automatic rule generation from data - Single-pass learning algorithm - Function approximation: $f(x) = \sin(x) + 0.1x$ - Rule conflict resolution

Estimated time: 60-75 min

Wang-Mendel: Linear Function

 [Open in Colab](#)

What you'll learn: - Simple case study - Effect of number of partitions - Performance metrics (MSE, RMSE, R^2)

Estimated time: 40-50 min

Wang-Mendel: Iris Classification

 [Open in Colab](#)

What you'll learn: - Classification with Wang-Mendel - Multi-class fuzzy classification - Interpretable fuzzy rules

Estimated time: 50-65 min

ANFIS: Iris Classification

 [Open in Colab](#)

What you'll learn: - Adaptive Neuro-Fuzzy Inference System - Gradient-based learning (backpropagation) - Membership function refinement - Lyapunov stability monitoring

Estimated time: 60-75 min

ANFIS: Regression

 [Open in Colab](#)

What you'll learn: - ANFIS for regression problems - Nonlinear function approximation - Comparison with neural networks

Estimated time: 50-65 min

Rules Optimization with PSO

 [Open in Colab](#)

What you'll learn: - Particle Swarm Optimization (PSO) - Metaheuristic optimization - Optimize membership function parameters

Estimated time: 50-65 min

Rules Optimization: Iris

 [Open in Colab](#)

What you'll learn: - Comparison: PSO vs DE vs GA - Classification optimization - Best practices

Estimated time: 55-70 min

12.4 🌊 Dynamic Systems (Advanced)

Fuzzy systems with time evolution.

p-Fuzzy Discrete: Predator-Prey

 [Open in Colab](#)

What you'll learn: - Discrete p-fuzzy systems: $x_{n+1} = x_n + f(x_n)$ - Population dynamics with fuzzy rules - Phase space analysis - Multiple initial conditions

Estimated time: 50-65 min

p-Fuzzy Continuous: Predator-Prey

 [Open in Colab](#)

What you'll learn: - Continuous p-fuzzy: $dx/dt = f(x)$ - ODE integration (Euler, RK4) - Oscillatory dynamics - Vector fields

Estimated time: 60-75 min

p-Fuzzy Discrete: Population Growth

 [Open in Colab](#)

What you'll learn: - Single population model - Logistic-like fuzzy dynamics - Bifurcation analysis

Estimated time: 45-60 min

What you'll learn: - ODEs with fuzzy parameters/initial conditions - α -level method for uncertainty propagation - Fuzzy envelopes

Estimated time: 55-70 min

Fuzzy ODE: Holling-Tanner

 [Open in Colab](#)

What you'll learn: - System of ODEs with fuzzy uncertainty - Multi-dimensional envelopes - Phase space with uncertainty

Estimated time: 60-75 min

Fuzzy ODE: Logistic Growth

 [Open in Colab](#)

12.5 By Difficulty Level

Beginner (0-2 notebooks recommended)

- Membership Functions
- Thermal Comfort

Intermediate (After fundamentals)

- All Inference Systems (Mamdani, Sugeno, Voting)

Advanced (Requires ML/math background)

- All Learning notebooks (Wang-Mendel, ANFIS, PSO)
- All Dynamics notebooks (p-fuzzy, Fuzzy ODEs)

12.6 Running the Examples

On Google Colab (Recommended)

1. Click any "Open in Colab" badge
2. Run the first cell to install: `!pip install pyfuzzy-toolbox`
3. Execute cells sequentially

Locally

```
# Clone repository
git clone https://github.com/lmoi6/pyfuzzy-toolbox.git
cd pyfuzzy-toolbox/notebooks_colab

# Install dependencies
pip install pyfuzzy-toolbox jupyter

# Launch Jupyter
jupyter notebook
```

12.7 Need Help?

- **API Reference:** Detailed documentation of all methods
- **User Guide:** Conceptual explanations and tutorials
- **GitHub Issues:** Report problems or ask questions