Comprehensive Instructions

Author: [Author Name] **Date:** November 4, 2025

Table of Contents

API_REFERENCE.md

DASE Engine API Reference

Complete Python & C++ API Documentation

Version: 1.0 **Last Updated**: October 23, 2025 **Performance**: 3.57 ns/op, 280M ops/sec, 4,344x speedup

---

Table of Contents

- Analog Cellular Engine

2. REST API
3. WebSocket API
4. C++ API
5. Examples

Python API

Installation

```
# Build the C++ extension
cd src/python
python setup.py build_ext --inplace

# Verify installation
python -c "import dase_engine as de; print('DASE Engine loaded')"
```

Import

```
import dase_engine as de
import numpy as np
```

CPU Features

Check hardware capabilities before using AVX2 features.

`CPUFeatures`

Static class for CPU capability detection.

Methods

`has_avx2() → bool` Check if CPU supports AVX2 instructions.

```
if de.CPUFeatures.has_avx2():
    print("AVX2 supported.")
else:
    print("AVX2 not available - performance will be limited")
```

Returns: `True` if AVX2 is available, `False` otherwise

`has_fma() → bool` Check if CPU supports FMA (Fused Multiply-Add) instructions.

```
if de.CPUFeatures.has_fma():
    print("FMA supported - extra performance boost!")
```

Returns: `True` if FMA is available, `False` otherwise

`print_capabilities() → None` Print all CPU capabilities to console.

```
de.CPUFeatures.print_capabilities()
# Output:
#   CPU Features:
#     AVX2: Supported
#     FMA:  Supported
```

Engine Metrics

Performance monitoring and statistics.

`EngineMetrics`

Class containing performance metrics from C++ engine.

Attributes

Attribute	Type	Description
`total_operations`	`int`	Total operations executed
`avx2_operations`	`int`	Operations using AVX2 SIMD
`node_processes`	`int`	Number of node processing calls
`harmonic_generations`	`int`	Harmonic generation count
`current_ns_per_op`	`float`	Current nanoseconds per operation
`current_ops_per_second`	`float`	Current operations per second
`speedup_factor`	`float`	Speedup vs baseline (15,500 ns)
`avg_time_ns`	`float`	Average time in nanoseconds
`throughput_gflops`	`float`	Throughput in GFLOPS

Methods

`update_performance() → None` Recalculate performance metrics.

```
metrics = engine.get_metrics()
metrics.update_performance()
```

`print_metrics()` → **None** Print metrics to console in formatted output.

```
metrics = engine.get_metrics()
metrics.print_metrics()
# Output:
#   D-ASE AVX2 ENGINE METRICS
#   ================================
#   Current Performance: 3.57 ns/op
#   Speedup Factor:     4344.48x
#   ...
```

`reset()` → **None** Reset all metrics to zero.

```
metrics = engine.get_metrics()
metrics.reset()
```

Analog Universal Node

Single processing node with multiple operations.

`AnalogUniversalNodeAVX2`

A versatile analog processing node with AVX2 acceleration.

Constructor

```
node = de.AnalogUniversalNodeAVX2()
```

Creates a new node with default state (all zeros).

Attributes

Attribute	Type	Description
`x`	`int16`	X coordinate for node positioning
`y`	`int16`	Y coordinate for node positioning
`z`	`int16`	Z coordinate for node positioning
`node_id`	`int`	Unique node identifier
`integrator_state`	`float`	Current integrator accumulator value
`previous_input`	`float`	Previous input signal value
`current_output`	`float`	Current output signal value
`feedback_gain`	`float`	Feedback coefficient

Basic Operations

`amplify(input_signal, gain)` → **float** Multiply input by gain factor.

```
output = node.amplify(input_signal=2.0, gain=0.5)
# output = 1.0
```

Parameters: - `input_signal` (float): Input signal value - `gain` (float): Amplification factor

Returns: Amplified signal (input × gain)

Performance: ~0.5 ns per call

`integrate(input_signal, time_constant)` → **float** Perform time-domain integration.

```
# Accumulate signal over time
for i in range(100):
    output = node.integrate(input_signal=1.0, time_constant=0.1)
```

Parameters: - `input_signal` (float): Input signal to integrate - `time_constant` (float): Integration time constant (seconds)

Returns: Integrated output

Performance: ~1.0 ns per call

Note: Uses thread-local accumulator for state

`apply_feedback(input_signal, feedback_gain)` → **float** Apply feedback from integrator state.

```
output = node.apply_feedback(input_signal=1.0, feedback_gain=0.5)
```

Parameters: - `input_signal` (float): Input signal - `feedback_gain` (float): Feedback coefficient

Returns: Input + (integrator_state × feedback_gain)

Performance: ~0.5 ns per call

Advanced Operations

`process_signal_avx2(input_signal, control_signal, aux_signal)` → **float** Complete signal processing chain with AVX2 optimization.

```
output = node.process_signal_avx2(
    input_signal=1.0,
```

```
    control_signal=0.5,
    aux_signal=0.1
)
```

Parameters: - `input_signal` (float): Primary input signal - `control_signal` (float): Control/modulation signal - `aux_signal` (float): Auxiliary signal

Returns: Processed output signal

Performance: ~3.57 ns per call (with AVX2)

Processing Chain: 1. Amplify input by control 2. Integrate result 3. Blend with auxiliary 4. Apply spectral processing (AVX2) 5. Apply feedback 6. Clamp output to [-10, +10]

Waveform Generation

`oscillate(frequency_hz, duration_seconds)` → List[float] Generate sinusoidal waveform (standard version).

```
# Generate 440 Hz tone for 0.1 seconds
waveform = node.oscillate(frequency_hz=440.0, duration_seconds=0.1)
# Returns 4,800 samples at 48kHz sample rate
```

Parameters: - `frequency_hz` (float): Frequency in Hertz - `duration_seconds` (float): Duration in seconds

Returns: List of float samples (48kHz sample rate)

Performance: ~0.027 µs per sample (37M samples/sec)

Sample Rate: 48,000 Hz

Range: [-1.0, +1.0] normalized

`oscillate_np(frequency_hz, duration_seconds)` → np.ndarray Generate waveform with NumPy zero-copy (more efficient).

```
# NumPy zero-copy version - more efficient
waveform = node.oscillate_np(frequency_hz=440.0, duration_seconds=0.1)
# Returns NumPy array directly, no Python list conversion
```

Parameters: - `frequency_hz` (float): Frequency in Hertz - `duration_seconds` (float): Duration in seconds

Returns: NumPy array of float32 samples

Performance: 2-3x faster than `oscillate()` (zero-copy)

Benefit: No Python list conversion overhead

Signal Processing

`process_block_frequency_domain(input_block)` → **List[float]** Process signal block using FFT bandpass filter.

```
import numpy as np

# Create test signal
signal = np.sin(np.linspace(0, 2*np.pi, 256))

# Process with FFT bandpass filter
filtered = node.process_block_frequency_domain(signal.tolist())
```

Parameters: - `input_block` (List[float]): Input signal samples

Returns: Filtered signal samples

Performance: ~0.07 ms for 256 samples

Filter: Bandpass (keeps middle 50% of spectrum)

Method: FFT → filter → inverse FFT

`process_block_frequency_domain_np(data)` → **np.ndarray** Process signal with NumPy zero-copy (in-place, faster).

```
import numpy as np

# Create test signal (float32 array)
signal = np.sin(np.linspace(0, 2*np.pi, 256)).astype(np.float32)

# Process in-place (modifies signal array)
filtered = node.process_block_frequency_domain_np(signal)
# filtered and signal are the same object
```

Parameters: - `data` (np.ndarray): NumPy array of float32 samples (modified in-place)

Returns: Same NumPy array (modified)

Performance: Faster than standard version (zero-copy)

Note: Modifies input array in-place!

Batch Processing

`process_batch(input_signals, control_signals, aux_signals)` → **List[float]** Process multiple samples in one call (5-10x faster than loop).

```
# Process 1000 samples at once
inputs = [1.0] * 1000
controls = [0.5] * 1000
aux = [0.1] * 1000

outputs = node.process_batch(inputs, controls, aux)
# Returns list of 1000 processed values
```

Parameters: - `input_signals` (List[float]): Batch of input signals - `control_signals` (List[float]): Batch of control signals - `aux_signals` (List[float]): Batch of auxiliary signals

Returns: List of processed output values

Performance: 5-10x faster than calling `process_signal_avx2()` in a loop

Requirement: All input lists must have the same length

Benefit: Reduces Python function call overhead

State Management

`set_feedback(feedback_coefficient)` → **None** Set feedback gain coefficient.

```
node.set_feedback(0.5)  # 50% feedback
```

Parameters: - `feedback_coefficient` (float): Feedback gain (clamped to [-2.0, +2.0])

`get_output()` → **float** Get current output value.

```
output = node.get_output()
```

Returns: Current output signal value

`get_integrator_state()` → **float** Get current integrator accumulator value.

```
state = node.get_integrator_state()
```

Returns: Current integrator state

`reset_integrator() → None` Reset integrator state to zero.

```
node.reset_integrator()
```

Analog Cellular Engine

Multi-node engine for large-scale parallel processing.

`AnalogCellularEngineAVX2`

High-performance engine managing multiple nodes with OpenMP parallelization.

Constructor

```
engine = de.AnalogCellularEngineAVX2(num_nodes=1024)
```

Parameters: - `num_nodes` (int): Number of processing nodes (default: 1024)

Memory: ~64 bytes per node (64-byte aligned for cache optimization)

Initialization: All nodes initialized with default state

Execution

`run_mission(cycles) → None` Execute processing mission with OpenMP parallelization.

```
engine = de.AnalogCellularEngineAVX2(num_nodes=1024)
engine.run_mission(cycles=3000)
# Executes 92,160,000 operations (1024 nodes × 30 iterations × 3000 cycles)
```

Parameters: - `cycles` (int): Number of processing cycles

Performance: 3.57 ns per operation (280M ops/sec)

Parallelization: OpenMP across all CPU cores

Output: Prints metrics to console after completion

`run_builtin_benchmark(iterations) → None` Run built-in benchmark test.

```
engine.run_builtin_benchmark(iterations=1000)
```

Parameters: - `iterations` (int): Number of benchmark iterations

Signal Processing

`process_signal_wave_avx2(input_signal, control_pattern)` → float
Process single signal with all nodes.

```
output = engine.process_signal_wave_avx2(
    input_signal=1.0,
    control_pattern=0.5
)
```

Parameters: - `input_signal` (float): Input signal value - `control_pattern` (float): Control pattern value

Returns: Processed output signal

`perform_signal_sweep_avx2(frequency)` → float Perform frequency sweep across nodes.

```
output = engine.perform_signal_sweep_avx2(frequency=440.0)
```

Parameters: - `frequency` (float): Sweep frequency in Hz

Returns: Sweep result

`process_block_frequency_domain(signal_block)` → None Process entire signal block with FFT (in-place).

```
signal = [1.0] * 256
engine.process_block_frequency_domain(signal)  # Modifies signal in-place
```

Parameters: - `signal_block` (List[float]): Signal samples (modified in-place)

Note: Modifies input list in-place!

Utility

`calculate_inter_node_coupling(node_index)` → float Calculate coupling factor for specific node.

```
coupling = engine.calculate_inter_node_coupling(node_index=10)
```

Parameters: - `node_index` (int): Index of node to analyze

Returns: Coupling factor

`generate_noise_signal()` → **float** Generate random noise signal.

```
noise = engine.generate_noise_signal()
```

Returns: Random noise value

Metrics

`get_metrics()` → **EngineMetrics** Get performance metrics object.

```
metrics = engine.get_metrics()
print(f"ns/op: {metrics.current_ns_per_op}")
print(f"Speedup: {metrics.speedup_factor}x")
```

Returns: `EngineMetrics` object with current statistics

`print_live_metrics()` → **None** Print current metrics to console.

```
engine.print_.metrics()
```

`reset_metrics()` → **None** Reset all metrics to zero.

```
engine.reset_metrics()
```

REST API

Flask-based REST API for web integration.

Server Setup

```
cd src/python
python bridge_server_improved.py
# Server runs on http://127.0.0.1:5000
```

Endpoints

GET / Serve main web interface.

Response: HTML page

URL: http://127.0.0.1:5000/

GET `/api/status` Get server and engine status.

Response:

```
{
  "server": "running",
  "engine_available": true,
  "timestamp": "2025-10-23T06:50:01",
  "cpu_capabilities": {
    "avx2": true,
    "fma": true
  },
  "performance": {
    "total_requests": 42,
    "active_simulations": 0,
    "total_operations": 92160000
  }
}
```

GET `/api/metrics` Get performance metrics.

Response:

```
{
  "timestamp": "2025-10-23T06:50:01",
  "server_metrics": {
    "total_requests": 42,
    "active_simulations": 0
  },
  "engine_available": true
}
```

POST `/api/benchmark` Run benchmark test.

Request Body:

```
{
  "nodes": 1024,
  "cycles": 1000
}
```

Response:

```
{
  "success": true,
  "benchmark_results": {
```

```
    "nodes": 1024,
    "cycles": 1000,
    "elapsed_seconds": 0.329,
    "cpp_metrics": {
      "total_operations": 30720000,
      "avx2_operations": 30720000,
      "current_ns_per_op": 3.57,
      "current_ops_per_second": 280289072,
      "speedup_factor": 4344.48
    }
  }
}
```

POST `/api/validate_formula` Validate DVSL formula syntax.

Request Body:

```
{
  "formula": "=DELTA(A1, gain=2.0)"
}
```

Response:

```
{
  "valid": true,
  "formula": "=DELTA(A1, gain=2.0)",
  "parsed": {
    "type": "formula",
    "expression": "DELTA(A1, gain=2.0)"
  }
}
```

WebSocket API

Real-time bidirectional communication.

Connection

```
const ws = new WebSocket('ws://127.0.0.1:5000/ws');

ws.onopen = () => {
    console.log('Connected to DASE engine');
};

ws.onmessage = (event) => {
```

```
    const data = JSON.parse(event.data);
    console.log('Received:', data);
};
```

Messages

Client → Server

`ping` Heartbeat check.

```
ws.send(JSON.stringify({ command: "ping" }));
```

Response: `{ "type": "pong" }`

`status` Get engine status.

```
ws.send(JSON.stringify({ command: "status" }));
```

Response:

```
{
  "type": "status",
  "engine_available": true,
  "active_simulations": 0
}
```

`run` Run simulation.

```
ws.send(JSON.stringify({
    command: "run",
    nodes: 1024,
    cycles: 3000
}));
```

Response (start):

```
{
  "type": "run_started",
  "nodes": 1024,
  "cycles": 3000
}
```

Response (done):

```
{
  "type": "run_done",
  "metrics": {
```

```
    "total_operations": 92160000,
    "current_ns_per_op": 3.57,
    "current_ops_per_second": 280289072,
    "speedup_factor": 4344.48
  },
  "timestamp": "2025-10-23T06:50:01"
}
```

Server → Client

`hello` Initial connection message.

```
{
  "type": "hello",
  "client_id": 140234567890,
  "caps": {
    "avx2": true,
    "fma": true
  },
  "server_time": "2025-10-23T06:50:01"
}
```

`error` Error notification.

```
{
  "type": "error",
  "error": "Engine not available"
}
```

C++ API

Direct C++ usage (advanced users).

Header Include

```
#include "analog_universal_node_engine_avx2.h"
```

Basic Usage

```
#include "analog_universal_node_engine_avx2.h"
#include <iostream>
```

```cpp
int main() {
    // Check CPU capabilities
    if (!CPUFeatures::hasAVX2()) {
        std::cerr << "AVX2 not available!" << std::endl;
        return 1;
    }

    // Create engine
    AnalogCellularEngineAVX2 engine(1024);

    // Run mission
    engine.runMission(3000);

    // Get metrics
    EngineMetrics metrics = engine.getMetrics();
    std::cout << "ns/op: " << metrics.current_ns_per_op << std::endl;

    return 0;
}
```

Compilation

```bash
# Compile with AVX2 and OpenMP
g++ -o my_program my_program.cpp \
    analog_universal_node_engine_avx2.cpp \
    -std=c++17 -O2 -mavx2 -fopenmp -lfftw3

# Or with MSVC
cl /EHsc /std:c++17 /O2 /Ob3 /arch:AVX2 /openmp \
   my_program.cpp analog_universal_node_engine_avx2.cpp \
   libfftw3-3.lib
```

Examples

Example 1: Basic Signal Processing

```python
import dase_engine as de

# Create node
node = de.AnalogUniversalNodeAVX2()

# Process single signal
output = node.process_signal_avx2(
    input_signal=1.0,
    control_signal=0.5,
```

```
    aux_signal=0.1
)
print(f"Output: {output}")
```

Example 2: Waveform Generation

```
import dase_engine as de
import numpy as np
import matplotlib.pyplot as plt

# Create node
node = de.AnalogUniversalNodeAVX2()

# Generate 440 Hz tone (A4 note)
waveform = node.oscillate_np(frequency_hz=440.0, duration_seconds=0.1)

# Plot
plt.plot(waveform[:1000])  # First 1000 samples
plt.title("440 Hz Sine Wave")
plt.xlabel("Sample")
plt.ylabel("Amplitude")
plt.show()
```

Example 3: Batch Processing

```
import dase_engine as de
import numpy as np

# Create node
node = de.AnalogUniversalNodeAVX2()

# Prepare batch data (1000 samples)
inputs = np.random.randn(1000).tolist()
controls = [0.5] * 1000
aux = [0.1] * 1000

# Process batch (more efficient than a loop)
outputs = node.process_batch(inputs, controls, aux)
print(f"Processed {len(outputs)} samples")
```

Example 4: FFT Filtering

```
import dase_engine as de
import numpy as np

# Create node
node = de.AnalogUniversalNodeAVX2()
```

```python
# Create noisy signal
t = np.linspace(0, 1, 1000)
signal = np.sin(2 * np.pi * 10 * t)  # 10 Hz sine
noise = np.random.randn(1000) * 0.1
noisy_signal = (signal + noise).astype(np.float32)

# Filter with FFT (in-place)
filtered = node.process_block_frequency_domain_np(noisy_signal)
print(f"Signal filtered (energy reduced by bandpass)")
```

Example 5: Large-Scale Simulation

```python
import dase_engine as de
import time

# Create large engine
engine = de.AnalogCellularEngineAVX2(num_nodes=4096)

# Run mission
start = time.time()
engine.run_mission(cycles=5000)
elapsed = time.time() - start

# Get metrics
metrics = engine.get_metrics()
print(f"Elapsed: {elapsed:.2f}s")
print(f"Operations: {metrics.total_operations}")
print(f"Throughput: {metrics.current_ops_per_second/1e6:.1f}M ops/sec")
print(f"Speedup: {metrics.speedup_factor:.1f}x")
```

Example 6: WebSocket Integration

```javascript
// Connect to engine
const ws = new WebSocket('ws://127.0.0.1:5000/ws');

ws.onopen = () => {
    // Run simulation
    ws.send(JSON.stringify({
        command: "run",
        nodes: 1024,
        cycles: 1000
    }));
};

ws.onmessage = (event) => {
```

```
    const data = JSON.parse(event.data);

    if (data.type === 'run_done') {
        console.log(`Performance: ${data.metrics.current_ns_per_op} ns/op`);
        console.log(`Speedup: ${data.metrics.speedup_factor}x`);
    }
};
```

Performance Tips

1. Use NumPy Zero-Copy Methods

```
# Slow (Python list conversion)
waveform = node.oscillate(440.0, 0.1)

# Fast (NumPy zero-copy, more efficient)
waveform = node.oscillate_np(440.0, 0.1)
```

2. Use Batch Processing

```
# Slow (Python function call overhead)
outputs = [node.process_signal_avx2(x, c, a) for x, c, a in zip(inputs, controls, aux)]

# Fast (more efficient)
outputs = node.process_batch(inputs, controls, aux)
```

3. Verify AVX2 Support

```
if not de.CPUFeatures.has_avx2():
    print("Warning: AVX2 not available, performance will be limited!")
```

4. Use Appropriate Node Count

```
# Too few: underutilizes CPU cores
engine = de.AnalogCellularEngineAVX2(num_nodes=10)

# Good: matches CPU thread count × workload
engine = de.AnalogCellularEngineAVX2(num_nodes=1024)

# Too many: memory overhead
engine = de.AnalogCellularEngineAVX2(num_nodes=100000)
```

Error Handling

```python
import dase_engine as de

try:
    # Create node
    node = de.AnalogUniversalNodeAVX2()

    # Process signal
    output = node.process_signal_avx2(1.0, 0.5, 0.1)

except ImportError as e:
    print(f"Engine not available: {e}")
    print("Build with: python setup.py build_ext --inplace")

except RuntimeError as e:
    print(f"Runtime error: {e}")

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

Version Information

Current Version: 1.0

Performance Metrics: - Operation Time: 3.57 ns - Throughput: 280M ops/sec - Speedup: 4,344x vs baseline - AVX2 Usage: 100%

Compiler Flags: - `/O2` - Optimization level 2 - `/Ob3` - Aggressive inlining - `/Oi` - Intrinsic functions - `/Ot` - Favor speed - `/arch:AVX2` - AVX2 instructions - `/fp:fast` - Fast floating-point - `/GL` - Whole program optimization - `/LTCG` - Link-time code generation

---

Support

Issues: Report at project repository **Documentation**: See `docs/` directory **Examples**: See `tests/` directory

---

End of API Reference

...and so on for all the other files.