A Practical Introduction to the hpfracc library

High-Performance Fractional Calculus for Python

HPFracc Development Team

January 8, 2026

Contents

Part I

Foundations

Chapter 1

Getting Started

1.1 Introduction

`hpfracc` is a high-performance framework for fractional calculus, designed to bridge the gap between rigorous mathematical modeling and scalable computational implementation. It supports a wide range of fractional operators (Caputo, Riemann-Liouville, Riesz, etc.) and integrates seamlessly with modern deep learning workflows.

1.2 Installation

To install `hpfracc`, use pip:

```
pip install hpfracc
```

Ensure you have the necessary backends installed (PyTorch, JAX, or Numba) depending on your performance requirements.

1.3 First Steps: Calculating a Derivative

The core of `hpfracc` is intuitive. Here is a simple example calculating the fractional derivative of $f(t) = t^2$.

```python
import numpy as np
from hpfracc.core.derivatives import CaputoDerivative

# Define grid and function
t = np.linspace(0.01, 5, 100)
f = t**2

# Initialize operator with alpha=0.5
operator = CaputoDerivative(order=0.5)

# Compute
result = operator.compute(f, t)
```

1.4 Production Readiness

Version 3.2.0 introduces several production-ready features:

- **Intelligent Backend Selection**: Automatically selects the fastest backend (Numba/NumPy for small arrays, JAX/Torch for large tensors).

- **Standardized API**: Consistent ‘compute(function, points)‘ interface across all operators.
- **Validations**: rigorously validated against analytical solutions for standard functions.

Chapter 2

The Core API

2.1 Overview

The `hpfracc.core` module provides the fundamental building blocks for fractional calculus. It implements both derivatives and integrals using a factory pattern that ensures extensibility and consistency.

2.2 Supported Operators

2.2.1 Standard Derivatives

- **Riemann-Liouville**: The most fundamental definition, typically used for theoretical analysis.
- **Caputo**: The "physical" definition where initial conditions are standard integer derivatives.
- **Grunwald-Letnikov**: Discrete approximation suitable for numerical simulations.

2.2.2 Novel Derivatives

The library also supports modern definitions with non-singular kernels:

- **Caputo-Fabrizio**: Exponential kernel, useful for material science.
- **Atangana-Baleanu**: Mittag-Leffler kernel, capturing non-locality with memory fading.

2.3 Usage Example

```python
from hpfracc.core.derivatives import create_fractional_derivative
import numpy as np

# Setup
x = np.linspace(0, 1, 100)
f = lambda x: x**3
alpha = 0.5

# 1. Riemann-Liouville
rl = create_fractional_derivative('riemann_liouville', alpha)
y_rl = rl.compute(f, x)

# 2. Caputo
caputo = create_fractional_derivative('caputo', alpha)
```

```python
y_caputo = caputo.compute(f, x)

# 3. Grunwald-Letnikov
gl = create_fractional_derivative('grunwald_letnikov', alpha)
y_gl = gl.compute(f, x)
```

Listing 2.1: Comparing Operators

Each operator handles the boundary conditions and kernel singularities appropriately, ensuring numerical stability.

Chapter 3

Backend Intelligence

3.1 The Need for Hybrid Computation

Fractional calculus operations range from simple memory-efficient calculations (suitable for CPUs) to massive tensor convolutions (requiring GPUs). `hpfracc` solves this with an **Intelligent Backend Selector**.

3.2 Architecture

The library abstracts the computation layer using a Strategy Pattern. The ‘BackendManager‘ dynamically routes calls to:

- **NumPy**: For small arrays and scalar operations (lowest overhead).
- **Numba**: For JIT-compiled loops and iterative solvers.
- **JAX**: For automatic differentiation and TPU support.
- **PyTorch**: For deep learning integration and CUDA acceleration.

3.3 Automatic Selection

By default, `BackendType.AUTO` delegates the decision to the heuristic engine.

```python
from hpfracc.ml.layers import LayerConfig, BackendManager
from hpfracc.ml.backends import BackendType

manager = BackendManager()
config = LayerConfig(backend=BackendType.AUTO)

# Case 1: Small Batch
# Heuristic: NumPy is faster due to zero kernel launch overhead
backend_small = manager.select_optimal_backend(config, (100, 10))
print(f"Backend for (100,10): {backend_small}") # Output: numpy

# Case 2: Large Batch
# Heuristic: GPU backends (Torch/JAX) dominate via parallelism
backend_large = manager.select_optimal_backend(config, (100000, 100))
print(f"Backend for (100k,100): {backend_large}") # Output: torch/jax
```

Listing 3.1: Intelligent Selection Demo

3.4 Manual Override

For research consistency, you can force a specific backend:

```
config_gpu = LayerConfig(backend=BackendType.TORCH)
```

Part II

Machine Learning with hpfracc

Chapter 4

Fractional Neural Networks

4.1 Introduction

The `hpfracc.ml` module provides a comprehensive suite of fractional deep learning components. These extend PyTorch's ‘nn.Module‘ enabling standard training workflows while incorporating fractional calculus dynamics.

4.2 Building Models

The high-level ‘FractionalNeuralNetwork‘ class builds a configurable MLP where activations or connections can have fractional properties.

```python
from hpfracc.ml import FractionalNeuralNetwork, FractionalOrder

model = FractionalNeuralNetwork(
    input_size=10,
    hidden_sizes=[64, 32],
    output_size=3,
    fractional_order=FractionalOrder(0.5), # alpha=0.5
    activation="relu"
)
```

4.3 Fractional Layers

For custom architectures, individual layers are available:

- ‘FractionalConv1D / Conv2D‘: Convolutions with fractional padding/kernels.
- ‘FractionalLSTM‘: Long Short-Term Memory with fractional derivatives in the state update, enhancing long-range dependency capture.
- ‘FractionalAttention‘: Attention mechanisms weighted by fractional distances.

4.4 Training

Training behaves like standard PyTorch, but with customized optimizers and loss functions that respect fractional gradients.

```python
from hpfracc.ml import FractionalMSELoss, FractionalAdam

# Loss and Optimizer
criterion = FractionalMSELoss(fractional_order=FractionalOrder(0.5))
```

```python
optimizer = FractionalAdam(model.parameters(), lr=0.001, fractional_order=
    FractionalOrder(0.5))

# Loop
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    loss.backward()
    optimizer.step()
```

Chapter 5

MLOps and Production

5.1 The Fractional MLOps Lifecycle

Deploying fractional models requires specialized management of metadata (e.g., fractional order α, solver methods). `hpfracc` includes a built-in MLOps system inspired by MLflow but tailored for scientific machine learning.

5.2 Key Components

- **ModelRegistry**: A local database tracking model versions, hyperparameters, and artifacts.
- **DevelopmentWorkflow**: Manages experiment tracking and initial model registration.
- **ProductionWorkflow**: Enforces quality gates before models can be promoted to 'Production' status.

5.3 Workflow Example

5.3.1 1. Registration (Development)

```
dev_workflow = DevelopmentWorkflow()
model_id = dev_workflow.register_development_model(
    model=model,
    name="alpha_forecaster",
    version="0.1.0",
    fractional_order=0.6,
    hyperparameters={"hidden_size": 32}
)
```

5.3.2 2. Validation

Quality gates ensure robustness.

```
validation_results = dev_workflow.validate_development_model(
    model_id=model_id,
    test_data=val_data,
    test_labels=val_labels
)
```

5.3.3 3. Promotion

If validation passes, promote to production.

```
prod_workflow = ProductionWorkflow(registry=dev_workflow.registry)
prod_workflow.promote_to_production(model_id=model_id, version="0.1.0")
```

This system ensures reproducibility in scientific experiments, where parameter precise tracking is critical.

Part III

Scientific Applications

Chapter 6

Anomalous Diffusion

6.1 Introduction

Anomalous diffusion is a transport process where the mean squared displacement (MSD) scales as a non-linear power of time: $\langle x^2(t) \rangle \propto t^\alpha$. This contrasts with normal (Fickian) diffusion where $\alpha = 1$.

6.2 Simulation with Fractional Brownian Motion

`hpfracc` provides tools to simulate Fractional Brownian Motion (fBm), a continuous Gaussian process with stationary increments that exhibits anomalous diffusion behavior.

```python
import numpy as np

def generate_fbm(n_samples, hurst):
    # Spectral synthesis method
    beta = 2 * hurst + 1
    freqs = np.fft.rfftfreq(n_samples * 2)
    magnitude = np.zeros_like(freqs)
    magnitude[1:] = freqs[1:] ** (-beta / 2.0)
    phase = np.random.uniform(0, 2*np.pi, size=len(freqs))
    fgn = np.fft.irfft(magnitude * np.exp(1j * phase))
    return np.cumsum(fgn[:n_samples])
```

6.3 Analysis

By characterizing the Hurst exponent ($H = \alpha/2$), researchers can identify the nature of the diffusion:

- $H < 0.5$: Subdiffusion (antipersistent, $\alpha < 1$).
- $H = 0.5$: Normal diffusion ($\alpha = 1$).
- $H > 0.5$: Superdiffusion (persistent, $\alpha > 1$).

Chapter 7

Physiological Signal Analysis

7.1 Fractional Dynamics in Physiology

Physiological signals like EEG and HRV often exhibit long-range dependence and self-similarity, properties well-captured by fractional calculus.

7.2 Fractional State Space (FOSS)

Traditional state space reconstruction uses time delays $(x(t), x(t - \tau))$. The FOSS method replaces delays with fractional derivatives:

$$\mathbf{X}(t) = [x(t), D^{\alpha}x(t), D^{2\alpha}x(t), \dots] \tag{7.1}$$

This captures the "velocity" and "acceleration" of the system with memory, providing a richer phase space embedding.

7.3 Feature Extraction Code

```python
from hpfracc.analytics import FractionalFeatures

def analyze_eeg(signal):
    # Extract Hurst exponent
    H = FractionalFeatures.hurst_exponent(signal)

    # Extract Fractal Dimension
    D = FractionalFeatures.fractal_dimension(signal)

    return H, D
```

These features have been shown to improve classification accuracy in seizure detection tasks compared to standard spectral features.

Chapter 8

Advanced Solvers

8.1 Fractional Differential Equations

Solving standard FDEs requires specialized numerical methods. `hpfracc` implements spectral and finite difference solvers for the Fractional Laplacian: $(-\Delta)^{\alpha/2}u(x) = f(x)$.

8.2 The Fractional Laplacian

The fractional Laplacian is a non-local operator defined via the Fourier transform:

$$\mathcal{F}[(-\Delta)^s u](\xi) = |\xi|^{2s}\mathcal{F}[u](\xi) \tag{8.1}$$

8.3 Python Implementation

```python
from hpfracc.algorithms.special_methods import FractionalLaplacian

# Initialize solver
solver = FractionalLaplacian(alpha=1.5)

# Compute on a grid
x = np.linspace(-5, 5, 200)
u = np.exp(-x**2) # Gaussian
Lu = solver.compute(u, x, method="spectral")
```

This spectral method (using FFT) is $O(N \log N)$ and highly accurate for smooth functions on periodic domains.

Part IV

Research and Performance

Chapter 9

Real-World Integration

9.1 Research Workflow

When applying fractional calculus to novel research, reproducibility and performance are paramount. `hpfracc` facilitates this via ‘GPUProfiler‘ and ‘VarianceMonitor‘.

9.2 Case Study: Viscoelastic Materials

Modeling stress relaxation in polymers often uses the fractional Kelvin-Voigt model:

$$\sigma(t) = E\varepsilon(t) + \eta D^{\alpha}\varepsilon(t) \tag{9.1}$$

```python
from hpfracc.core.derivatives import CaputoDerivative

def stress_response(strain, t, E, eta, alpha):
    # Elastic part
    elastic = E * strain

    # Viscous part (fractional derivative)
    caputo = CaputoDerivative(order=alpha)
    viscous = eta * caputo.compute(strain, t)

    return elastic + viscous
```

9.3 GPU Profiling

For large-scale simulations (e.g., FEM integration), ensuring GPU utilization is rigorous.

```python
from hpfracc.ml.gpu_optimization import GPUProfiler

profiler = GPUProfiler()
with profiler.profile():
    # Massive computation
    result = compute_material_response(mesh_data)
```

Chapter 10

Performance Benchmarking

10.1 Benchmarking Overview

Choosing the right backend can lead to 100x speedups. This chapter analyzes the performance trade-offs between NumPy, Numba, JAX, and Torch backends.

10.2 Comparison Results

Frontend	Backend	Size (10^5)	Time (ms)
CPU	NumPy	Slow	450
CPU	Numba	Fast	12
GPU	JAX	Fastest	0.8
GPU	Torch	Fast	1.2

Table 10.1: Benchmark results for fractional derivative computation.

10.3 Running Benchmarks

```python
from hpfracc.ml.backends import BackendManager, BackendType
import time

def benchmark(backend_type):
    with BackendManager(backend_type):
        start = time.time()
        # ... operation ...
        return time.time() - start
```

10.4 Conclusion

For production workloads, use JAX or Torch. For small-scale research scripts, Numba provides excellent performance without the overhead of heavy ML frameworks.