Turbulence and Machine Learning

A Comprehensive Guide for the AI Engineer

From Fluid Dynamics Fundamentals to Neural Network Implementation

Sreekanth Pannala

January 5, 2026

Contents

I Theoretical Foundations **1**

1 Fundamentals of Fluid Dynamics **3**
1.1 What is a Fluid? . . . 3
1.1.1 Key Distinction from Solids . . . 3
1.2 The Continuum Hypothesis . . . 3
1.2.1 The Problem of Scale . . . 3
1.2.2 Mathematical Consequence . . . 3
1.2.3 When the Continuum Breaks Down . . . 4
1.3 Fundamental Fluid Properties . . . 4
1.3.1 Density (ρ) . . . 4
1.3.2 Pressure (p) . . . 4
1.3.3 Viscosity (μ, ν) . . . 4
1.4 The Material Derivative . . . 5
1.4.1 Following the Fluid . . . 5
1.4.2 The Material Derivative . . . 5
1.5 Conservation Laws . . . 5
1.5.1 Conservation of Mass (Continuity) . . . 5
1.5.2 Conservation of Momentum (Newton's Second Law) . . . 6
1.6 Dimensionless Numbers . . . 6
1.6.1 Why Non-Dimensionalize? . . . 6
1.6.2 The Reynolds Number (Most Important!) . . . 6
1.7 Boundary Conditions . . . 7
1.7.1 No-Slip Condition . . . 7
1.7.2 Periodic Boundaries . . . 7
1.8 Summary for ML Engineers . . . 7
1.8.1 Key Takeaways . . . 7
1.8.2 Mapping to Code . . . 8
1.9 Further Reading . . . 8
1.9.1 Textbooks . . . 8
1.10 Exercises . . . 8

2 The Navier-Stokes Equations **9**
2.1 Historical Context . . . 9
2.1.1 The Problem Worth $1 Million . . . 9
2.1.2 Timeline . . . 9
2.2 The Incompressible Navier-Stokes Equations . . . 9
2.2.1 The Complete System . . . 9
2.2.2 Component Form (Cartesian Coordinates) . . . 10
2.3 Physical Interpretation of Each Term . . . 10
2.3.1 The Momentum Equation Dissected . . . 10
2.3.2 Local Acceleration: $\partial \mathbf{u}/\partial t$ . . . 10

2.3.3 Convective Acceleration: $(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}$ . . . 11
2.3.4 Pressure Gradient: $-\boldsymbol{\nabla} p/\rho$ . . . 11
2.3.5 Viscous Diffusion: $\nu \nabla^2 \mathbf{u}$ . . . 11
2.4 Non-Dimensionalization . . . 12
2.4.1 Scaling the Equations . . . 12
2.4.2 The Non-Dimensional Navier-Stokes . . . 12
2.5 Vorticity Formulation . . . 12
2.5.1 Why Vorticity? . . . 12
2.5.2 Physical Interpretation . . . 12
2.5.3 Vortex Stretching . . . 13
2.6 Energy Equation . . . 13
2.6.1 Kinetic Energy Budget . . . 13
2.6.2 Statistically Steady Turbulence . . . 13
2.7 Spectral Form of Navier-Stokes . . . 13
2.7.1 Fourier Transform . . . 13
2.7.2 Navier-Stokes in Fourier Space . . . 14
2.7.3 Key Insights . . . 14
2.8 Types of Solutions . . . 14
2.8.1 Exact Solutions (Rare!) . . . 14
2.8.2 Numerical Solutions . . . 14
2.9 The Turbulence Closure Problem . . . 15
2.9.1 Reynolds Decomposition . . . 15
2.9.2 Reynolds-Averaged Navier-Stokes (RANS) . . . 15
2.9.3 The Closure Problem . . . 15
2.10 ML Approaches to Navier-Stokes . . . 16
2.10.1 Physics-Informed Neural Networks (PINNs) . . . 16
2.10.2 Neural Operators . . . 16
2.10.3 Hybrid Methods . . . 16
2.11 Connection to This Repository . . . 16
2.11.1 What We Do . . . 16
2.11.2 Key Code Mappings . . . 16
2.11.3 The Divergence-Free Projection . . . 16
2.12 Summary . . . 17
2.12.1 Key Equations to Remember . . . 17
2.12.2 For the ML Engineer . . . 17
2.13 Exercises . . . 17

3 Introduction to Turbulence **19**
3.1 What is Turbulence? . . . 19
3.1.1 A Working Definition . . . 19
3.1.2 Laminar vs Turbulent Flow . . . 19
3.2 The Reynolds Experiment (1883) . . . 19
3.2.1 The Classic Demonstration . . . 19
3.2.2 Transition to Turbulence . . . 20
3.3 Statistical Description . . . 20
3.3.1 Why Statistics? . . . 20
3.3.2 Reynolds Decomposition . . . 20
3.3.3 Key Statistical Quantities . . . 20
3.4 Correlation Functions . . . 21
3.4.1 Two-Point Correlation . . . 21
3.4.2 Integral Length Scale . . . 21

3.4.3 Taylor Microscale . . . 21
3.5 The Energy Spectrum . . . 22
3.5.1 From Correlation to Spectrum . . . 22
3.5.2 Energy Conservation . . . 22
3.5.3 Typical Energy Spectrum . . . 22
3.6 Isotropy and Homogeneity . . . 22
3.6.1 Definitions . . . 22
3.6.2 Why These Idealizations? . . . 22
3.6.3 Local Isotropy Hypothesis (Kolmogorov) . . . 23
3.7 Eddies and the Cascade Picture . . . 23
3.7.1 What is an "Eddy"? . . . 23
3.7.2 The Energy Cascade . . . 23
3.7.3 Cascade Mechanism: Vortex Stretching . . . 23
3.8 Intermittency . . . 23
3.8.1 What is Intermittency? . . . 23
3.8.2 Consequences . . . 23
3.8.3 Quantifying Intermittency . . . 24
3.9 Types of Turbulent Flows . . . 24
3.9.1 Free Shear Flows . . . 24
3.9.2 Wall-Bounded Flows . . . 24
3.9.3 Homogeneous Isotropic Turbulence (HIT) . . . 24
3.9.4 Stratified Turbulence . . . 24
3.9.5 Rotating Turbulence . . . 24
3.10 Turbulence Modeling Hierarchy . . . 25
3.10.1 Direct Numerical Simulation (DNS) . . . 25
3.10.2 Large Eddy Simulation (LES) . . . 25
3.10.3 Reynolds-Averaged Navier-Stokes (RANS) . . . 25
3.11 Why ML for Turbulence? . . . 25
3.11.1 The Perfect ML Problem . . . 25
3.11.2 What ML Can Do . . . 25
3.11.3 Our Approach . . . 26
3.12 Summary for ML Engineers . . . 26
3.12.1 Key Concepts . . . 26
3.12.2 Mapping to Code . . . 26
3.13 Further Reading . . . 26
3.13.1 Textbooks . . . 26
3.14 Exercises . . . 26

4 Kolmogorov Theory and Energy Cascade 27
4.1 Historical Context . . . 27
4.1.1 The State of Turbulence Before 1941 . . . 27
4.1.2 Kolmogorov's Breakthrough . . . 27
4.2 The Three Kolmogorov Hypotheses . . . 27
4.2.1 Hypothesis 1: Local Isotropy . . . 27
4.2.2 Hypothesis 2: Similarity (First Similarity Hypothesis) . . . 28
4.2.3 Hypothesis 3: Inertial Range (Second Similarity Hypothesis) . . . 28
4.3 Kolmogorov Scales . . . 28
4.3.1 Dimensional Analysis . . . 28
4.3.2 Physical Meaning . . . 28
4.3.3 Numerical Values . . . 29
4.4 The Scale Hierarchy . . . 29

4.4.1 Three Characteristic Scales . . . 29
4.4.2 Scale Ratios . . . 29
4.5 The Energy Spectrum: $k^{-5/3}$ Law . . . 29
4.5.1 Derivation . . . 29
4.5.2 The Kolmogorov Constant . . . 30
4.5.3 Experimental Verification . . . 30
4.5.4 Full Pope Spectrum Model . . . 30
4.6 Structure Functions . . . 30
4.6.1 Definition . . . 30
4.6.2 Kolmogorov's Prediction . . . 31
4.6.3 The 4/5 Law . . . 31
4.6.4 Structure Function Scaling . . . 31
4.7 The Energy Cascade Mechanism . . . 31
4.7.1 Physical Picture . . . 31
4.7.2 Energy Balance . . . 31
4.7.3 Eddy Turnover Time . . . 32
4.8 Spectral Energy Transfer . . . 32
4.8.1 The Spectral Energy Equation . . . 32
4.8.2 Properties of $T(k)$ . . . 32
4.8.3 Energy Flux . . . 32
4.9 Refined Similarity Hypothesis (K62) . . . 32
4.9.1 The Problem with K41 . . . 32
4.9.2 Kolmogorov's 1962 Refinement . . . 33
4.9.3 Log-Normal Model . . . 33
4.10 Inverse and Dual Cascades . . . 33
4.10.1 2D Turbulence: Inverse Cascade . . . 33
4.10.2 3D Turbulence: Forward Only . . . 33
4.11 Connection to Our Code . . . 33
4.11.1 TurbulenceParams Class . . . 33
4.11.2 Energy Spectrum Validation . . . 34
4.12 Summary: The K41 Framework . . . 34
4.12.1 Key Results . . . 34
4.12.2 Assumptions . . . 34
4.12.3 Limitations . . . 35
4.13 Exercises . . . 35

5 Spectral Methods and Fourier Analysis 37
5.1 Why Spectral Methods? . . . 37
5.1.1 Advantages for Turbulence Simulation . . . 37
5.1.2 Key Insight . . . 37
5.1.3 When to Use Spectral Methods . . . 37
5.2 Fourier Series Fundamentals . . . 38
5.2.1 Continuous Fourier Transform . . . 38
5.2.2 Discrete Fourier Transform (DFT) . . . 38
5.2.3 Wavenumbers . . . 38
5.3 Fast Fourier Transform (FFT) . . . 38
5.3.1 The Computational Breakthrough . . . 38
5.3.2 NumPy FFT Functions . . . 38
5.3.3 PyTorch FFT (GPU Accelerated) . . . 39
5.4 Properties of the Fourier Transform . . . 39
5.4.1 Linearity . . . 39

5.4.2 Differentiation . . . 39
5.4.3 Convolution Theorem . . . 39
5.4.4 Parseval's Theorem (Energy Conservation) . . . 40
5.5 3D Spectral Representation . . . 40
5.5.1 Velocity Field in Fourier Space . . . 40
5.5.2 Wavenumber Magnitude . . . 40
5.5.3 Hermitian Symmetry . . . 40
5.6 Spectral Derivatives . . . 40
5.6.1 First Derivative . . . 40
5.6.2 Second Derivative (Laplacian) . . . 41
5.6.3 Gradient, Divergence, Curl . . . 41
5.7 The Divergence-Free Projection . . . 41
5.7.1 Why It Matters . . . 41
5.7.2 Helmholtz Decomposition . . . 41
5.7.3 Projection Operator . . . 41
5.7.4 Energy Loss from Projection . . . 42
5.8 Spectral Filtering . . . 42
5.8.1 The Filtering Operation . . . 42
5.8.2 Common Filter Types . . . 42
5.9 Shell Averaging for $E(k)$ . . . 43
5.9.1 The Energy Spectrum . . . 43
5.9.2 Algorithm . . . 43
5.10 Pseudo-Spectral Methods . . . 44
5.10.1 The Nonlinear Term Problem . . . 44
5.10.2 Pseudo-Spectral Algorithm . . . 44
5.10.3 Aliasing Error . . . 44
5.10.4 Dealiasing (2/3 Rule) . . . 44
5.11 Connection to Machine Learning . . . 44
5.11.1 Fourier Features in Neural Networks . . . 44
5.11.2 Fourier Neural Operator (FNO) . . . 44
5.11.3 Our Approach: Hybrid . . . 45
5.12 Practical Considerations . . . 45
5.12.1 FFT Normalization . . . 45
5.12.2 Memory for 3D FFT . . . 45
5.13 Summary . . . 45
5.13.1 Key Formulas . . . 45
5.13.2 Code Quick Reference . . . 46
5.14 Exercises . . . 46

II Code Implementation 47

6 Synthetic Turbulence Generation 49
6.1 Overview . . . 49
6.1.1 Purpose of This Module . . . 49
6.1.2 Why Synthetic Data? . . . 49
6.2 Module Structure . . . 49
6.3 TurbulenceParams: Configuration Class . . . 50
6.3.1 The Dataclass . . . 50
6.3.2 Key Relationships . . . 50
6.3.3 Automatic Derivation in `__post_init__` . . . 50
6.3.4 Usage Example . . . 51

6.4 Wavenumber Generation . . . 51
6.4.1 The Function . . . 51
6.4.2 Wavenumber Convention . . . 51
6.4.3 3D Mesh Generation . . . 51
6.5 Energy Spectrum Models . . . 51
6.5.1 Pope's Model Spectrum . . . 51
6.5.2 Implementation . . . 52
6.6 Isotropic Turbulence Generation . . . 52
6.6.1 The Main Function . . . 52
6.6.2 Algorithm Steps . . . 52
6.7 Hermitian Symmetry . . . 54
6.7.1 Why Required? . . . 54
6.7.2 Implementation . . . 54
6.8 Diagnostic Tools . . . 54
6.8.1 Spectrum Diagnosis . . . 54
6.9 Common Usage Patterns . . . 55
6.9.1 Generate Single Field . . . 55
6.9.2 Verify Divergence-Free . . . 55
6.10 Limitations and Caveats . . . 55
6.10.1 What This Does NOT Capture . . . 55
6.10.2 When This Is Sufficient . . . 55
6.11 Summary . . . 55
6.11.1 Key Functions . . . 55
6.11.2 Key Concepts Implemented . . . 56

7 Spectral Filtering **57**
7.1 Overview . . . 57
7.1.1 The Fundamental Operation . . . 57
7.1.2 Why Filtering Matters . . . 57
7.2 Module Structure . . . 57
7.3 Filter Types . . . 58
7.3.1 Enum Definition . . . 58
7.3.2 Which to Use? . . . 58
7.4 Gaussian Filter . . . 58
7.4.1 Theory . . . 58
7.4.2 Implementation . . . 58
7.4.3 Properties . . . 58
7.5 Box (Top-Hat) Filter . . . 59
7.5.1 Theory . . . 59
7.5.2 Implementation . . . 59
7.6 Sharp Spectral Filter . . . 59
7.6.1 Theory . . . 59
7.6.2 Implementation . . . 59
7.6.3 Properties . . . 60
7.7 SpectralFilter Class . . . 60
7.7.1 Initialization . . . 60
7.7.2 Main Filter Method . . . 60
7.7.3 NumPy Implementation . . . 60
7.8 Multi-Scale Hierarchy . . . 61
7.8.1 Creating Scale Hierarchy . . . 61
7.8.2 Example Usage . . . 61

7.9 Sub-Filter Scale Quantities . . . 61
7.9.1 Sub-Filter Field . . . 61
7.9.2 Sub-Filter Stress Tensor . . . 62
7.10 Training Data Preparation . . . 62
7.10.1 The Key Function . . . 62
7.10.2 Usage Pattern . . . 63
7.11 Filter Size Selection . . . 63
7.11.1 Physical Interpretation . . . 63
7.11.2 Typical Values . . . 63
7.11.3 Choosing R_1 and R_2 . . . 64
7.12 Summary . . . 64
7.12.1 Key Concepts . . . 64
7.12.2 Key Functions . . . 64

8 Neural Network Architectures 65
8.1 Overview . . . 65
8.1.1 The Task . . . 65
8.1.2 Architecture Requirements . . . 65
8.2 Module Structure . . . 65
8.3 Building Blocks . . . 66
8.3.1 3D Convolutional Block . . . 66
8.3.2 Residual Block . . . 66
8.3.3 Attention Block . . . 67
8.4 Positional Encoding . . . 67
8.4.1 Fourier Features . . . 67
8.5 U-Net 3D Architecture . . . 68
8.5.1 Overview . . . 68
8.5.2 Implementation . . . 68
8.6 Scale Reconstruction Network . . . 69
8.7 Fourier Neural Operator . . . 70
8.7.1 Concept . . . 70
8.7.2 Spectral Convolution Layer . . . 70
8.8 Divergence-Free Output Layer . . . 71
8.8.1 Physical Constraint . . . 71
8.8.2 Projection Approach . . . 71
8.9 Model Selection Guide . . . 72
8.9.1 By Problem Size . . . 72
8.9.2 By Task . . . 72
8.10 Summary . . . 72
8.10.1 Key Architectures . . . 72
8.10.2 Key Layers . . . 72

9 Physics-Informed Loss Functions 73
9.1 Overview . . . 73
9.1.1 Why Physics-Informed Losses? . . . 73
9.1.2 Loss Function Structure . . . 73
9.2 Module Structure . . . 73
9.3 Data Fidelity Losses . . . 74
9.3.1 Mean Squared Error . . . 74
9.3.2 Relative Error . . . 74
9.4 Spectral Loss . . . 74
9.4.1 Motivation . . . 74

9.4.2 Implementation . . . 74
9.5 Divergence Loss . . . 75
9.5.1 Physical Constraint . . . 75
9.5.2 Implementation . . . 76
9.6 Gradient Loss . . . 76
9.6.1 Motivation . . . 76
9.6.2 Implementation . . . 76
9.7 Structure Function Loss . . . 77
9.7.1 Physics . . . 77
9.7.2 Implementation . . . 77
9.8 Energy Conservation Loss . . . 77
9.8.1 Physical Constraint . . . 77
9.9 Vorticity Loss . . . 78
9.10 Combined Physics Loss . . . 78
9.11 Loss Weight Scheduling . . . 79
9.11.1 Dynamic Weighting . . . 79
9.12 Best Practices . . . 80
9.12.1 Weight Selection Guidelines . . . 80
9.12.2 Common Mistakes . . . 80
9.13 Summary . . . 80
9.13.1 Loss Functions . . . 80
9.13.2 Key Equations . . . 80

10 Training Pipeline **81**
10.1 Overview . . . 81
10.1.1 Training Workflow . . . 81
10.2 Module Structure . . . 81
10.3 Dataset Class . . . 82
10.3.1 TurbulenceDataset . . . 82
10.3.2 Creating DataLoaders . . . 83
10.4 Trainer Class . . . 83
10.4.1 Initialization . . . 83
10.4.2 Training Epoch . . . 83
10.4.3 Validation . . . 84
10.4.4 Main Training Loop . . . 85
10.5 Metrics Computation . . . 85
10.6 Checkpointing . . . 86
10.7 Learning Rate Scheduling . . . 86
10.8 Early Stopping . . . 87
10.9 Complete Training Script . . . 87
10.10 Summary . . . 88
10.10.1 Key Components . . . 88
10.10.2 Training Checklist . . . 88

11 Analysis and Diagnostics **89**
11.1 Overview . . . 89
11.1.1 Purpose . . . 89
11.1.2 Key Diagnostics . . . 89
11.2 Energy Spectrum Analysis . . . 89
11.2.1 3D Energy Spectrum . . . 89
11.2.2 Spectrum Fitting . . . 90
11.3 Structure Functions . . . 91

11.3.1 Longitudinal Structure Function . . . 91
11.3.2 Kolmogorov's 4/5 Law . . . 91
11.4 Dissipation Estimation . . . 92
11.4.1 From Velocity Gradients . . . 92
11.4.2 From Spectrum . . . 92
11.5 TurbulenceAnalyzer Class . . . 92
11.6 Reconstruction Quality Metrics . . . 94
11.7 Visualization Helpers . . . 94
11.8 Summary Report . . . 95
11.9 Summary . . . 95
11.9.1 Key Functions . . . 95
11.9.2 Key Metrics for Model Validation . . . 95

12 Accelerated Computing **97**
12.1 Overview . . . 97
12.1.1 Why Acceleration Matters . . . 97
12.2 Module Structure . . . 97
12.3 GPU Detection and Setup . . . 98
12.4 Accelerated Spectral Filter . . . 98
12.5 Accelerated Turbulence Generation . . . 99
12.6 Accelerated Analysis . . . 100
12.7 Memory Management . . . 101
12.8 Benchmarking Tools . . . 102
12.9 Multi-GPU Support . . . 103
12.10 Performance Tips . . . 104
12.10.1 Best Practices . . . 104
12.10.2 Memory Optimization . . . 104
12.10.3 Typical Speedups . . . 104
12.11 Summary . . . 104
12.11.1 Key Classes . . . 104
12.11.2 Key Optimizations . . . 105

Quick Reference **107**

Part I

Theoretical Foundations

Chapter 1

Fundamentals of Fluid Dynamics

1.1 What is a Fluid?

A **fluid** is any substance that continuously deforms (flows) under an applied shear stress, no matter how small. This includes:

- **Liquids**: Water, oil, blood
- **Gases**: Air, steam, natural gas
- **Plasmas**: Ionized gases (stars, fusion reactors)

1.1.1 Key Distinction from Solids

Property	**Solid**	**Fluid**
Response to shear	Deforms to equilibrium, then stops	Continuously deforms
Shape	Maintains own shape	Takes shape of container
Molecular spacing	Fixed lattice	Free to move

ML Analogy: Think of a solid as having a "fixed embedding" while a fluid's molecules have "dynamic embeddings" that continuously update based on neighbors.

1.2 The Continuum Hypothesis

1.2.1 The Problem of Scale

Fluids are made of discrete molecules ($\sim 10^{23}$ per mole), but tracking each one is computationally impossible. The **continuum hypothesis** assumes:

> At scales much larger than molecular spacing ($\sim 10^{-9}$ m), fluid properties can be treated as continuous fields.

1.2.2 Mathematical Consequence

Instead of tracking N particles with positions $\mathbf{x}_i(t)$, we define **fields**:

$$\rho(\mathbf{x}, t) \quad \text{Density field (scalar)} \tag{1.1}$$

$$\mathbf{u}(\mathbf{x}, t) \quad \text{Velocity field (vector)} \tag{1.2}$$

$$p(\mathbf{x}, t) \quad \text{Pressure field (scalar)} \tag{1.3}$$

$$T(\mathbf{x}, t) \quad \text{Temperature field (scalar)} \tag{1.4}$$

ML Analogy: This is like moving from a particle-based representation to a continuous feature map. Your neural network will learn to predict these continuous fields.

1.2.3 When the Continuum Breaks Down

The **Knudsen number** determines validity:

$$\text{Kn} = \frac{\lambda}{L} \tag{1.5}$$

where λ = mean free path and L = characteristic length scale.

Kn Range	Regime	Example
$\text{Kn} < 0.01$	Continuum valid	Most engineering flows
$0.01 < \text{Kn} < 0.1$	Slip flow	Microchannels
$\text{Kn} > 0.1$	Rarefied gas	Spacecraft re-entry

1.3 Fundamental Fluid Properties

1.3.1 Density (ρ)

Definition: Mass per unit volume

$$\rho = \frac{dm}{dV} \quad [\text{kg/m}^3] \tag{1.6}$$

Typical values:

- Air at sea level: 1.225 kg/m^3
- Water: 1000 kg/m^3
- Mercury: 13,600 kg/m^3

1.3.2 Pressure (p)

Definition: Normal force per unit area

$$p = \frac{dF_n}{dA} \quad [\text{Pa} = \text{N/m}^2] \tag{1.7}$$

Pressure is **isotropic**—it acts equally in all directions at a point.

Key insight for ML: Pressure is the Lagrange multiplier that enforces incompressibility. In your neural network, pressure ensures the velocity field remains divergence-free.

1.3.3 Viscosity (μ, ν)

Dynamic viscosity (μ): Resistance to shear deformation

$$\tau = \mu \frac{du}{dy} \quad [\text{Pa} \cdot \text{s}] \tag{1.8}$$

This is **Newton's law of viscosity**—shear stress is proportional to velocity gradient.

Kinematic viscosity (ν): Viscosity normalized by density

$$\nu = \frac{\mu}{\rho} \quad [\text{m}^2/\text{s}] \tag{1.9}$$

Fluid	ν (m^2/s)
Air	1.5×10^{-5}
Water	1.0×10^{-6}
Honey	$\sim 10^{-2}$
Motor oil	$\sim 10^{-4}$

Why this matters for turbulence: Viscosity determines how quickly velocity gradients are smoothed out. Small ν means sharp gradients can persist $\rightarrow$ more turbulent.

1.4 The Material Derivative

1.4.1 Following the Fluid

There are two ways to describe fluid motion:

1. **Eulerian (Field View)**: Fix position in space, watch fluid flow past
   - Like a weather station measuring wind
   - Fields: $\mathbf{u}(x, y, z, t)$
2. **Lagrangian (Particle View)**: Follow a fluid parcel as it moves
   - Like a balloon drifting with the wind
   - Trajectories: $\mathbf{X}(t)$, where $d\mathbf{X}/dt = \mathbf{u}(\mathbf{X}, t)$

1.4.2 The Material Derivative

To write conservation laws, we need to track how properties change **following the fluid**. The **material derivative** (also called substantial derivative):

$$\frac{D}{Dt} = \frac{\partial}{\partial t} + \mathbf{u} \cdot \boldsymbol{\nabla} \tag{1.10}$$

Expanded in 3D:

$$\frac{Df}{Dt} = \frac{\partial f}{\partial t} + u\frac{\partial f}{\partial x} + v\frac{\partial f}{\partial y} + w\frac{\partial f}{\partial z} \tag{1.11}$$

where:

- $\partial f/\partial t$ = **local** rate of change (at fixed position)
- $\mathbf{u} \cdot \boldsymbol{\nabla} f$ = **convective** rate of change (due to fluid moving)

1.5 Conservation Laws

All of fluid dynamics comes from three fundamental conservation principles.

1.5.1 Conservation of Mass (Continuity)

Statement: Mass is neither created nor destroyed.

Differential form:

$$\frac{\partial \rho}{\partial t} + \boldsymbol{\nabla} \cdot (\rho \mathbf{u}) = 0 \tag{1.12}$$

For incompressible flow (constant ρ):

$$\boxed{\boldsymbol{\nabla} \cdot \mathbf{u} = 0} \tag{1.13}$$

This says the velocity field is **divergence-free** (solenoidal).
ML insight: In our neural network, we enforce $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$ either:

1. **Hard constraint**: Use stream function or project onto divergence-free space
2. **Soft constraint**: Add divergence penalty to loss function

1.5.2 Conservation of Momentum (Newton's Second Law)

Statement: $F = ma$ applied to fluid elements.

$$\rho \frac{D\mathbf{u}}{Dt} = -\boldsymbol{\nabla} p + \mu \nabla^2 \mathbf{u} + \rho \mathbf{g} \tag{1.14}$$

Terms:

- $\rho(D\mathbf{u}/Dt)$: Inertia (mass $\times$ acceleration)
- $-\boldsymbol{\nabla} p$: Pressure gradient force
- $\mu \nabla^2 \mathbf{u}$: Viscous forces
- $\rho \mathbf{g}$: Body forces (gravity)

This is the **Navier-Stokes equation** (see Chapter 2).

1.6 Dimensionless Numbers

1.6.1 Why Non-Dimensionalize?

1. **Reduce parameters**: Many combinations of $(L, U, \nu, \rho \ldots)$ give same physics
2. **Identify dominant effects**: Compare term magnitudes
3. **Enable scaling**: Lab results apply to full-scale systems

1.6.2 The Reynolds Number (Most Important!)

$$\boxed{\mathrm{Re} = \frac{UL}{\nu} = \frac{\text{Inertial forces}}{\text{Viscous forces}}} \tag{1.15}$$

where U = characteristic velocity, L = characteristic length, ν = kinematic viscosity.
Physical meaning:

Re	Regime	Characteristics
$\mathrm{Re} \ll 1$	Creeping flow	Viscosity dominates, reversible, laminar
$\mathrm{Re} \sim 1$–1000	Transitional	Neither dominates
$\mathrm{Re} \gg 1$	Turbulent	Inertia dominates, chaotic, irreversible

Examples:

Flow	Re
Bacteria swimming	10^{-4}
Blood in capillaries	10^{-2}
Pipe flow transition	~ 2300
Car on highway	10^{6}
Airplane	10^{7}
Atmospheric boundary layer	10^{8}

For this repository: We typically work with Taylor-scale Reynolds number $\text{Re}_\lambda \sim 100$–1000.

1.7 Boundary Conditions

1.7.1 No-Slip Condition

At a solid wall, fluid velocity equals wall velocity:

$$\mathbf{u}|_{\text{wall}} = \mathbf{u}_{\text{wall}} \tag{1.16}$$

Usually $\mathbf{u}_{\text{wall}} = 0$ (stationary wall).

1.7.2 Periodic Boundaries

For homogeneous turbulence simulations:

$$\mathbf{u}(\mathbf{x} + L) = \mathbf{u}(\mathbf{x}) \tag{1.17}$$

The domain wraps around—what exits one side enters the other.
Why we use this:

1. Simulates infinite domain without boundaries
2. Enables spectral (Fourier) methods
3. Removes artificial boundary effects

1.8 Summary for ML Engineers

1.8.1 Key Takeaways

1. **Fluids are continuous fields**—Perfect for CNN/neural field representations
2. **Conservation laws are PDEs**—Physics-informed loss functions enforce these
3. **Reynolds number determines turbulence**—High Re means chaotic, multi-scale behavior
4. **Incompressibility** ($\boldsymbol{\nabla} \cdot \mathbf{u} = 0$) is a hard constraint in our velocity predictions
5. **Periodic boundaries** enable Fourier/spectral methods (ideal for our architecture)

1.8.2 Mapping to Code

Physics Concept	Code Location
Velocity field	`velocity: np.ndarray` shape (N, N, N, 3)
Grid spacing	`dx = L / N`
Domain size	`L = 2π` (typical)
Reynolds number	`Re_lambda` in TurbulenceParams
Divergence-free	Enforced in `generate_isotropic_turbulence_spectral()`

1.9 Further Reading

1.9.1 Textbooks

1. **Kundu, Cohen, Dowling**—"Fluid Mechanics" (comprehensive intro)
2. **White**—"Fluid Mechanics" (engineering focus)
3. **Batchelor**—"An Introduction to Fluid Dynamics" (mathematical, classic)

1.10 Exercises

1. **Dimensional Analysis**: A pipe has diameter $D = 0.1$ m, flow velocity $U = 2$ m/s, and the fluid is water ($\nu = 10^{-6}$ m^2/s). Calculate Re and predict if flow is laminar or turbulent.
2. **Material Derivative**: If temperature field $T(x, t) = x^2 t$ and velocity $\mathbf{u} = (1, 0, 0)$, compute DT/Dt.
3. **Divergence**: For velocity field $\mathbf{u} = (x, -y, 0)$, verify it satisfies $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$.

Chapter 2

The Navier-Stokes Equations

"The Navier-Stokes equations are perhaps the most famous unsolved problem in mathematics—yet we use them every day to design aircraft, predict weather, and now, to train neural networks."

2.1 Historical Context

2.1.1 The Problem Worth $1 Million

The **Navier-Stokes existence and smoothness problem** is one of the seven Millennium Prize Problems. The question: Do solutions always exist, and do they remain smooth (non-singular) for all time?

For ML engineers, this means:

- We're working with equations that are fundamentally difficult
- Even mathematicians don't fully understand them
- This is why data-driven approaches are so promising

2.1.2 Timeline

- **1822**: Claude-Louis Navier derives equations using molecular theory
- **1845**: George Gabriel Stokes provides rigorous derivation
- **1883**: Osborne Reynolds discovers transition to turbulence
- **1941**: Kolmogorov develops statistical theory of turbulence
- **2000**: Clay Mathematics Institute offers $1M prize

2.2 The Incompressible Navier-Stokes Equations

2.2.1 The Complete System

For incompressible, Newtonian fluids:

Continuity (mass conservation):

$$\nabla \cdot \mathbf{u} = 0 \tag{2.1}$$

Momentum (Newton's second law):

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} = -\frac{\boldsymbol{\nabla} p}{\rho} + \nu \nabla^2 \mathbf{u} + \mathbf{f} \tag{2.2}$$

Where:

- $\mathbf{u}$ = velocity vector field (u, v, w)
- p = pressure field (divided by density)
- ρ = density (constant for incompressible)
- ν = kinematic viscosity
- $\mathbf{f}$ = body forces (gravity, etc.)

2.2.2 Component Form (Cartesian Coordinates)

Continuity:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{2.3}$$

x-momentum:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = -\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right) \tag{2.4}$$

y-momentum:

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = -\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2} + \frac{\partial^2 v}{\partial z^2}\right) \tag{2.5}$$

z-momentum:

$$\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = -\frac{\partial p}{\partial z} + \nu\left(\frac{\partial^2 w}{\partial x^2} + \frac{\partial^2 w}{\partial y^2} + \frac{\partial^2 w}{\partial z^2}\right) \tag{2.6}$$

2.3 Physical Interpretation of Each Term

2.3.1 The Momentum Equation Dissected

$$\underset{\substack{\downarrow \\ \text{Local} \\ \text{Acceleration}}}{\frac{\partial \mathbf{u}}{\partial t}} + \underset{\substack{\downarrow \\ \text{Convection} \\ \text{(Nonlinear)}}}{(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}} = \underset{\substack{\downarrow \\ \text{Pressure, Viscous,} \\ \text{Body Forces}}}{-\boldsymbol{\nabla} p/\rho + \nu \nabla^2 \mathbf{u} + \mathbf{f}}$$

2.3.2 Local Acceleration: $\partial \mathbf{u}/\partial t$

What it is: Rate of change of velocity at a fixed point in space.

Physical meaning: How quickly the flow is speeding up or slowing down at location (x, y, z).

ML relevance: This is what we're often predicting—the time evolution.

2.3.3 Convective Acceleration: $(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}$

What it is: Change in velocity due to fluid moving to a different location.

Physical meaning: Even in steady flow ($\partial\mathbf{u}/\partial t = 0$), a fluid particle accelerates as it moves through regions of different velocity.

The Nonlinearity: This term makes everything hard:

$$(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} = u\frac{\partial \mathbf{u}}{\partial x} + v\frac{\partial \mathbf{u}}{\partial y} + w\frac{\partial \mathbf{u}}{\partial z} \tag{2.7}$$

It's quadratic in velocity ($u \times \partial u$), creating:

- Energy transfer between scales
- Chaotic dynamics
- The turbulent cascade

ML insight: This nonlinearity is why turbulence is so hard to model. Linear models fail. Neural networks can capture nonlinear relationships.

2.3.4 Pressure Gradient: $-\boldsymbol{\nabla} p/\rho$

What it is: Force per unit mass due to pressure differences.

Physical meaning: Fluid accelerates from high to low pressure.

The Hidden Complexity: Pressure is not a prognostic variable—it adjusts instantaneously to enforce $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$. This is the **pressure-Poisson equation**:

$$\nabla^2 p = -\rho \boldsymbol{\nabla} \cdot [(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}] \tag{2.8}$$

ML insight: Pressure is a Lagrange multiplier enforcing incompressibility. Some neural networks predict pressure explicitly; others enforce divergence-free output.

2.3.5 Viscous Diffusion: $\nu\nabla^2\mathbf{u}$

What it is: Smoothing of velocity gradients by viscosity.

Physical meaning: Sharp velocity differences get "smeared out" over time.

The Laplacian: In 3D Cartesian:

$$\nabla^2 \mathbf{u} = \frac{\partial^2 \mathbf{u}}{\partial x^2} + \frac{\partial^2 \mathbf{u}}{\partial y^2} + \frac{\partial^2 \mathbf{u}}{\partial z^2} \tag{2.9}$$

This is a **diffusion operator**—same as heat equation.

Energy dissipation: This term always removes energy from the flow:

$$\varepsilon = \nu \int |\boldsymbol{\nabla}\mathbf{u}|^2 \, dV > 0 \tag{2.10}$$

ML insight: Viscosity sets the smallest scales in turbulence. The Kolmogorov scale $\eta \sim \nu^{3/4}$ is where kinetic energy dissipates.

2.4 Non-Dimensionalization

2.4.1 Scaling the Equations

Choose reference scales:

- Length: L (e.g., domain size, pipe diameter)
- Velocity: U (e.g., mean flow speed)
- Time: L/U (convective time scale)
- Pressure: ρU^2 (dynamic pressure)

Define dimensionless variables:

$$x^* = x/L, \quad u^* = u/U, \quad t^* = tU/L, \quad p^* = p/(\rho U^2) \tag{2.11}$$

2.4.2 The Non-Dimensional Navier-Stokes

$$\frac{\partial \mathbf{u}^*}{\partial t^*} + (\mathbf{u}^* \cdot \boldsymbol{\nabla}^*)\mathbf{u}^* = -\boldsymbol{\nabla}^* p^* + \frac{1}{\mathrm{Re}} \nabla^{2*} \mathbf{u}^* \tag{2.12}$$

The only parameter is Reynolds number:

$$\boxed{\mathrm{Re} = \frac{UL}{\nu}} \tag{2.13}$$

Profound implication: All flows with the same Re have the same non-dimensional behavior. A centimeter-scale lab experiment can represent a kilometer-scale atmospheric phenomenon if Re matches.

2.5 Vorticity Formulation

2.5.1 Why Vorticity?

Vorticity $\omega = \boldsymbol{\nabla} \times \mathbf{u}$ measures local rotation:

$$\omega = \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}, \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}, \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right) \tag{2.14}$$

Taking curl of Navier-Stokes eliminates pressure:

$$\frac{\partial \omega}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla})\omega = (\omega \cdot \boldsymbol{\nabla})\mathbf{u} + \nu \nabla^2 \omega \tag{2.15}$$

2.5.2 Physical Interpretation

Term	Meaning
$\partial\omega/\partial t$	Local rate of rotation change
$(\mathbf{u} \cdot \boldsymbol{\nabla})\omega$	Convection of vorticity
$(\omega \cdot \boldsymbol{\nabla})\mathbf{u}$	Vortex stretching (3D only!)
$\nu \nabla^2 \omega$	Vorticity diffusion

2.5.3 Vortex Stretching

The term $(\omega \cdot \boldsymbol{\nabla})\mathbf{u}$ exists only in 3D and is crucial for turbulence:

- When vortex tubes are stretched, they spin faster (like ice skater pulling in arms)
- Creates small-scale vorticity from large-scale flow
- Fundamental mechanism of energy cascade

2D vs 3D Turbulence:

- 2D: No vortex stretching → inverse cascade (energy goes UP in scale)
- 3D: Vortex stretching → forward cascade (energy goes DOWN in scale)

2.6 Energy Equation

2.6.1 Kinetic Energy Budget

Multiply momentum equation by $\mathbf{u}$ and integrate:

$$\frac{d}{dt}\left(\frac{1}{2}\int |\mathbf{u}|^2\, dV\right) = -\int p(\boldsymbol{\nabla}\cdot\mathbf{u})\, dV - \nu\int |\boldsymbol{\nabla}\mathbf{u}|^2\, dV + \int \mathbf{u}\cdot\mathbf{f}\, dV \tag{2.16}$$

For incompressible flow ($\boldsymbol{\nabla}\cdot\mathbf{u} = 0$):

$$\frac{dE}{dt} = -\varepsilon + P \tag{2.17}$$

Where:

- $E = \frac{1}{2}\langle|\mathbf{u}|^2\rangle$ = total kinetic energy
- $\varepsilon = \nu\langle|\boldsymbol{\nabla}\mathbf{u}|^2\rangle$ = dissipation rate
- $P = \langle\mathbf{u}\cdot\mathbf{f}\rangle$ = power input from forcing

2.6.2 Statistically Steady Turbulence

If $dE/dt = 0$ on average:

$$\varepsilon = P \tag{2.18}$$

Energy input at large scales = Energy dissipated at small scales.
This is the foundation of Kolmogorov's theory (Chapter 4).

2.7 Spectral Form of Navier-Stokes

2.7.1 Fourier Transform

For periodic domains, expand velocity in Fourier series:

$$\mathbf{u}(\mathbf{x}, t) = \sum_{\mathbf{k}} \hat{\mathbf{u}}(\mathbf{k}, t) e^{i\mathbf{k}\cdot\mathbf{x}} \tag{2.19}$$

Where $\mathbf{k} = (k_x, k_y, k_z)$ is the wavevector.

2.7.2 Navier-Stokes in Fourier Space

$$\frac{\partial \hat{\mathbf{u}}}{\partial t} = -i\mathbf{k}\hat{p} - \nu k^2 \hat{\mathbf{u}} + \hat{N}(\mathbf{k}) \tag{2.20}$$

Where $\hat{N}(\mathbf{k})$ represents the nonlinear convolution (mode interactions).

2.7.3 Key Insights

1. **Pressure projects onto divergence-free space**:

$$\hat{\mathbf{u}}_{\text{solenoidal}} = \left(\mathbf{I} - \frac{\mathbf{k}\mathbf{k}^T}{|\mathbf{k}|^2}\right) \hat{\mathbf{u}}_{\text{raw}} \tag{2.21}$$

2. **Viscosity acts as low-pass filter**:
   - High-k modes decay as $\exp(-\nu k^2 t)$
   - Sets the dissipation scale
3. **Nonlinearity couples modes**:
   - Mode at $\mathbf{k}_1$ interacts with mode at $\mathbf{k}_2$
   - Transfers energy to mode at $\mathbf{k}_1 + \mathbf{k}_2$

ML insight: This is why spectral methods are powerful—viscosity and pressure are simple in Fourier space. Our code uses spectral synthesis and filtering.

2.8 Types of Solutions

2.8.1 Exact Solutions (Rare!)

Poiseuille flow (pipe/channel):

$$u(y) = \frac{G}{2\mu}(h^2 - y^2) \tag{2.22}$$

Couette flow (shear between plates):

$$u(y) = \frac{Uy}{h} \tag{2.23}$$

Stokes flow ($\text{Re} \to 0$, creeping motion):

- Time-reversible
- No inertia
- $\boldsymbol{\nabla} p = \mu \nabla^2 \mathbf{u}$

2.8.2 Numerical Solutions

Method	Pros	Cons	When to Use
DNS	Exact (to grid resolution)	Expensive: $O(\text{Re}^3)$	Research, ground truth
LES	Resolves large scales	Needs subgrid model	Engineering
RANS	Fast	Loses all turbulence	Quick estimates

This repository: We use synthetic DNS-like data to train neural networks for scale reconstruction (a type of super-resolution).

2.9 The Turbulence Closure Problem

2.9.1 Reynolds Decomposition

Split velocity into mean and fluctuation:

$$\mathbf{u} = \bar{\mathbf{U}} + \mathbf{u}' \tag{2.24}$$

Where:

- $\bar{\mathbf{U}} = \langle \mathbf{u} \rangle$ = time/ensemble average (mean flow)
- $\mathbf{u}' = \mathbf{u} - \bar{\mathbf{U}}$ = fluctuation (turbulence)

2.9.2 Reynolds-Averaged Navier-Stokes (RANS)

Averaging the equations introduces **Reynolds stress**:

$$\frac{\partial \bar{\mathbf{U}}}{\partial t} + (\bar{\mathbf{U}} \cdot \boldsymbol{\nabla})\bar{\mathbf{U}} = -\frac{\boldsymbol{\nabla}\bar{P}}{\rho} + \nu \nabla^2 \bar{\mathbf{U}} - \boldsymbol{\nabla} \cdot \langle \mathbf{u}'\mathbf{u}' \rangle \tag{2.25}$$

The term $\langle \mathbf{u}'\mathbf{u}' \rangle$ (Reynolds stress tensor) is **unknown**.

2.9.3 The Closure Problem

We have 4 equations (continuity + 3 momentum) but now have 10 unknowns:

- 3 mean velocities ($\bar{U}, \bar{V}, \bar{W}$)
- 1 mean pressure ($\bar{P}$)
- 6 Reynolds stresses ($\langle u'^2 \rangle, \langle v'^2 \rangle, \langle w'^2 \rangle, \langle u'v' \rangle, \langle u'w' \rangle, \langle v'w' \rangle$)

The closure problem: We need additional equations/models for Reynolds stresses.
Traditional approaches:

- Eddy viscosity models (k-ε, k-ω)
- Reynolds stress models
- Algebraic stress models

ML approaches:

- Learn Reynolds stress from DNS
- Neural network turbulence models
- Data-driven closure terms

This repository's approach: Learn the sub-filter scale contributions directly from resolved scales.

2.10 ML Approaches to Navier-Stokes

2.10.1 Physics-Informed Neural Networks (PINNs)

Loss function includes physics:

```
loss = MSE_data + lambda * (NSE_residual**2 + continuity_residual**2)
```

Where:

$$\text{NSE\_residual} = \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} + \boldsymbol{\nabla} p - \nu \nabla^2 \mathbf{u} \tag{2.26}$$

$$\text{continuity\_residual} = \boldsymbol{\nabla} \cdot \mathbf{u} \tag{2.27}$$

2.10.2 Neural Operators

Learn solution operator directly:

$$\mathcal{G}_\theta : \mathbf{u}(t=0) \to \mathbf{u}(t=T) \tag{2.28}$$

Examples: Fourier Neural Operator (FNO), DeepONet, Transformers for PDEs.

2.10.3 Hybrid Methods

Combine numerical solvers with ML corrections for 40–80x speedup.

2.11 Connection to This Repository

2.11.1 What We Do

1. **Generate synthetic turbulence** satisfying N-S statistics (not time evolution)
2. **Filter to multiple scales** (coarse-graining)
3. **Learn scale reconstruction** ($\mathbf{u}_{\text{fine}}$ from $\mathbf{u}_{\text{coarse}}$)

2.11.2 Key Code Mappings

Physics	Code
$\boldsymbol{\nabla} \cdot \mathbf{u} = 0$	Helmholtz projection in `synthetic_turbulence.py`
$E(k) \sim k^{-5/3}$	`model_spectrum()` function
Viscous dissipation	η scale in `TurbulenceParams`
Sub-filter scales	`SpectralFilter.compute_subfilter_stress()`

2.11.3 The Divergence-Free Projection

From `src/synthetic_turbulence.py`:

```
# Project to divergence-free using Helmholtz decomposition
# u_solenoidal = u - k(k.u)/|k|^2
k_sq = kx**2 + ky**2 + kz**2
k_sq_safe = np.where(k_sq > 0, k_sq, 1.0)
k_dot_u = kx * u_hat + ky * v_hat + kz * w_hat
proj_factor = k_dot_u / k_sq_safe

u_hat = np.where(k_sq > 0, u_hat - kx * proj_factor, 0)
v_hat = np.where(k_sq > 0, v_hat - ky * proj_factor, 0)
w_hat = np.where(k_sq > 0, w_hat - kz * proj_factor, 0)
```

This ensures $\mathbf{k} \cdot \hat{\mathbf{u}} = 0$, which means $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$ in physical space.

2.12 Summary

2.12.1 Key Equations to Remember

Equation	Physical Meaning
$\boldsymbol{\nabla} \cdot \mathbf{u} = 0$	Mass conservation (incompressible)
$\partial \mathbf{u}/\partial t + (\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} = -\boldsymbol{\nabla} p + \nu \nabla^2 \mathbf{u}$	Momentum conservation
$\mathrm{Re} = UL/\nu$	Ratio of inertia to viscosity
$\varepsilon = \nu \langle \lvert \boldsymbol{\nabla} \mathbf{u} \rvert^2 \rangle$	Dissipation rate

2.12.2 For the ML Engineer

1. **N-S is a hard PDE**: Nonlinear, coupled, multi-scale
2. **Pressure enforces incompressibility**: Not a dynamic variable
3. **Reynolds number controls complexity**: High Re = turbulent = hard
4. **Vortex stretching creates small scales**: Why 3D turbulence is especially complex
5. **Energy cascades to small scales**: Must resolve or model

2.13 Exercises

1. **Non-dimensionalization**: Starting from dimensional N-S, derive the non-dimensional form and show Re is the only parameter.
2. **Energy Budget**: Show that multiplying momentum by $\mathbf{u}$ and integrating gives the kinetic energy equation.
3. **Vorticity**: Take the curl of the momentum equation to derive the vorticity equation.
4. **Spectral viscosity**: For a single Fourier mode $\hat{u}(k)e^{ikx}$, show that viscous term becomes $-\nu k^2 \hat{u}$.

Chapter 3

Introduction to Turbulence

> *"When I meet God, I am going to ask him two questions: Why relativity? And why turbulence? I really believe he will have an answer for the first."*
> — Werner Heisenberg (attributed)

3.1 What is Turbulence?

3.1.1 A Working Definition

Turbulence is a state of fluid motion characterized by:

1. **Irregularity**: Chaotic, apparently random fluctuations
2. **Diffusivity**: Enhanced mixing and transport
3. **Three-dimensionality**: Vortex stretching requires 3D
4. **Dissipation**: Continuous energy drain at small scales
5. **Multi-scale**: Wide range of active length and time scales
6. **Continuum**: Still governed by Navier-Stokes (not quantum)

3.1.2 Laminar vs Turbulent Flow

Property	**Laminar**	**Turbulent**
Velocity	Steady, predictable	Chaotic, fluctuating
Mixing	Slow (diffusion only)	Fast (advection + diffusion)
Drag	Low	High
Predictability	Deterministic	Statistical only
Reynolds number	Low (Re $<\sim$ 2000)	High (Re $>\sim$ 4000)

3.2 The Reynolds Experiment (1883)

3.2.1 The Classic Demonstration

Osborne Reynolds injected dye into pipe flow and varied the flow rate:

- **Low Re** (< 2000): Dye streak stays as thin line
- **High Re** (> 4000): Dye rapidly mixes throughout pipe

3.2.2 Transition to Turbulence

The **critical Reynolds number** for pipe flow:

$$\mathrm{Re}_{\mathrm{crit}} \approx 2300 \tag{3.1}$$

But transition is subtle:

- Below 2300: Always laminar
- 2300–4000: Depends on disturbances
- Above 4000: Always turbulent

Key insight: Turbulence is an **instability**—small perturbations grow exponentially.

3.3 Statistical Description

3.3.1 Why Statistics?

Turbulent flows are:

- **Deterministic**: Governed by N-S equations
- **Sensitive to initial conditions**: Chaos (Lyapunov exponents > 0)
- **Unpredictable**: Can't forecast individual realizations

Solution: Describe **statistical properties** instead of instantaneous fields.

3.3.2 Reynolds Decomposition

Split any quantity into mean and fluctuation:

$$u(\mathbf{x}, t) = \bar{U}(\mathbf{x}) + u'(\mathbf{x}, t) \tag{3.2}$$

Where:

- $\bar{U} = \langle u \rangle$ = time/ensemble average
- u' = fluctuation with $\langle u' \rangle = 0$

For homogeneous turbulence: $\bar{U} = 0$ (no mean flow), only fluctuations exist.

3.3.3 Key Statistical Quantities

Turbulent Kinetic Energy (TKE)

$$k = \frac{1}{2}\langle \mathbf{u}' \cdot \mathbf{u}' \rangle = \frac{1}{2}(\langle u'^2 \rangle + \langle v'^2 \rangle + \langle w'^2 \rangle) \tag{3.3}$$

Units: $\mathrm{m}^2/\mathrm{s}^2$ (energy per unit mass)

RMS Velocity

$$u_{\mathrm{rms}} = \sqrt{\langle u'^2 \rangle} \approx \sqrt{2k/3} \quad \text{for isotropic turbulence} \tag{3.4}$$

Turbulence Intensity

$$I = u_{\text{rms}}/U_{\text{mean}} \tag{3.5}$$

Typical values: 1–10% in wind tunnels, 10–30% in atmosphere.

Reynolds Stress Tensor

$$\tau_{ij} = -\rho\langle u_i' u_j' \rangle \tag{3.6}$$

This is a symmetric 3×3 tensor:

$$-\rho \times \begin{pmatrix} \langle u'^2 \rangle & \langle u'v' \rangle & \langle u'w' \rangle \\ \langle v'u' \rangle & \langle v'^2 \rangle & \langle v'w' \rangle \\ \langle w'u' \rangle & \langle w'v' \rangle & \langle w'^2 \rangle \end{pmatrix} \tag{3.7}$$

- Diagonal: Normal stresses (TKE components)
- Off-diagonal: Shear stresses (momentum transport)

3.4 Correlation Functions

3.4.1 Two-Point Correlation

Measures how velocity at one point relates to velocity at another:

$$R_{ij}(\mathbf{r}) = \langle u_i'(\mathbf{x}) u_j'(\mathbf{x}+\mathbf{r}) \rangle \tag{3.8}$$

For isotropic turbulence:

$$R_{ij}(\mathbf{r}) = u^2 \left[f(r) \left(\delta_{ij} - \frac{r_i r_j}{r^2} \right) + g(r) \frac{r_i r_j}{r^2} \right] \tag{3.9}$$

Where $f(r)$ and $g(r)$ are longitudinal and transverse correlations.

3.4.2 Integral Length Scale

$$L = \int_0^\infty f(r)\, dr \tag{3.10}$$

Physical meaning: Size of the largest energy-containing eddies.

3.4.3 Taylor Microscale

Defined from the curvature of correlation at origin:

$$\lambda^2 = -2/f''(0) \tag{3.11}$$

Physical meaning: Intermediate scale between energy-containing and dissipation.
Taylor Reynolds number:

$$\boxed{\text{Re}_\lambda = \frac{u_{\text{rms}} \times \lambda}{\nu}} \tag{3.12}$$

This is the Reynolds number used in isotropic turbulence research (and in our code).

3.5 The Energy Spectrum

3.5.1 From Correlation to Spectrum

The energy spectrum $E(k)$ is the Fourier transform of the correlation function:

$$E(k) = \frac{1}{2\pi} \int R(r) e^{-ikr} \, dr \tag{3.13}$$

Physical meaning: $E(k)\,dk$ = energy in wavenumber range $[k, k + dk]$

3.5.2 Energy Conservation

$$\text{Total KE} = \int_0^\infty E(k)\,dk = \frac{1}{2}\langle u'^2 \rangle \tag{3.14}$$

3.5.3 Typical Energy Spectrum

Three ranges:

1. **Energy-containing range** ($k \sim 1/L$):
   - Largest eddies
   - Most of the kinetic energy
   - Flow-dependent (geometry, forcing)
2. **Inertial subrange** ($1/L < k < 1/\eta$):
   - Universal behavior
   - $E(k) \propto k^{-5/3}$ [Kolmogorov]
   - Energy cascades, no dissipation
3. **Dissipation range** ($k \sim 1/\eta$):
   - Smallest eddies
   - Viscosity dominant
   - Energy converted to heat

3.6 Isotropy and Homogeneity

3.6.1 Definitions

Homogeneous turbulence: Statistics don't depend on position

$$\langle u'(\mathbf{x}) u'(\mathbf{x} + \mathbf{r}) \rangle = R(\mathbf{r}) \quad [\text{not } R(\mathbf{x}, \mathbf{r})] \tag{3.15}$$

Isotropic turbulence: Statistics don't depend on direction

$$\langle u'^2 \rangle = \langle v'^2 \rangle = \langle w'^2 \rangle \tag{3.16}$$

3.6.2 Why These Idealizations?

1. **Mathematical simplicity**: Enables analytical theory
2. **Computational efficiency**: Periodic boundaries, spectral methods
3. **Universal features**: Far from boundaries, turbulence approaches isotropy
4. **Foundation for models**: Understanding ideal case informs realistic models

3.6.3 Local Isotropy Hypothesis (Kolmogorov)

> At sufficiently high Reynolds number, small-scale turbulence becomes locally isotropic, regardless of large-scale anisotropy.

This is the basis for universal small-scale behavior.

3.7 Eddies and the Cascade Picture

3.7.1 What is an "Eddy"?

An **eddy** is a conceptual model of a coherent vortical structure:

- Not precisely defined
- Useful for physical intuition
- Hierarchy of sizes (cascade)

Richardson's Poem (1922):

> Big whirls have little whirls that feed on their velocity,
> And little whirls have lesser whirls and so on to viscosity.

3.7.2 The Energy Cascade

Energy flows from large to small scales:

1. **Large eddies**: Size $\sim L$ (integral scale), receive energy from forcing
2. **Inertial transfer**: Energy passes through intermediate scales without dissipation
3. **Small eddies**: Size $\sim \eta$ (Kolmogorov scale), dissipate energy to heat

3.7.3 Cascade Mechanism: Vortex Stretching

In 3D, vortex tubes can stretch. Conservation of angular momentum:

$$\omega \times r^2 = \text{constant} \quad \Rightarrow \quad \text{If } r \text{ decreases, } \omega \text{ increases} \tag{3.17}$$

This creates small-scale vorticity from large-scale motion.

3.8 Intermittency

3.8.1 What is Intermittency?

Small-scale turbulence is **not** uniformly distributed—it comes in bursts:

- Most of volume: relatively quiescent
- Rare regions: intense dissipation

3.8.2 Consequences

1. **Non-Gaussian statistics**: Velocity PDFs have stretched tails
2. **Structure functions**: Deviate from Kolmogorov predictions
3. **Anomalous scaling**: Exponents differ from dimensional analysis

3.8.3 Quantifying Intermittency

Flatness (kurtosis):

$$F = \frac{\langle u'^4 \rangle}{\langle u'^2 \rangle^2} \tag{3.18}$$

- Gaussian: $F = 3$
- Turbulent small scales: $F \gg 3$ (stretched tails)

Skewness:

$$S = \frac{\langle u'^3 \rangle}{\langle u'^2 \rangle^{3/2}} \tag{3.19}$$

For velocity derivatives, $S \approx -0.4$ (asymmetric).

3.9 Types of Turbulent Flows

3.9.1 Free Shear Flows

- Jets, wakes, mixing layers
- No solid boundaries
- Develop self-similar profiles

3.9.2 Wall-Bounded Flows

- Pipes, channels, boundary layers
- Strong near-wall gradients
- Law of the wall: $u^+ = (1/\kappa)\ln(y^+) + B$

3.9.3 Homogeneous Isotropic Turbulence (HIT)

- Idealized case
- No mean gradients
- **What we simulate in this repository!**

3.9.4 Stratified Turbulence

- Density variations (atmosphere, ocean)
- Buoyancy effects
- Internal waves

3.9.5 Rotating Turbulence

- Coriolis effects
- Two-dimensionalization
- Geophysical relevance

3.10 Turbulence Modeling Hierarchy

3.10.1 Direct Numerical Simulation (DNS)

Resolve all scales from L to η.

Grid requirement:

$$N \sim (L/\eta)^3 \sim \mathrm{Re}^{9/4} \tag{3.20}$$

Re_λ	N per dimension	Total grid points
50	~ 64	2.6×10^5
100	~ 128	2.1×10^6
200	~ 256	1.7×10^7
400	~ 512	1.3×10^8
1000	~ 2048	8.6×10^9

This repository generates synthetic DNS-like data at $N = 64$–256.

3.10.2 Large Eddy Simulation (LES)

Resolve large scales, model small scales.

Filter Navier-Stokes:

$$\frac{\partial \bar{\mathbf{u}}}{\partial t} + (\bar{\mathbf{u}} \cdot \boldsymbol{\nabla})\bar{\mathbf{u}} = -\boldsymbol{\nabla}\bar{p} + \nu\nabla^2\bar{\mathbf{u}} - \boldsymbol{\nabla} \cdot \tau_{\mathrm{SGS}} \tag{3.21}$$

Where τ_{SGS} = **subgrid-scale stress** (needs modeling).

This is what our neural network learns: The sub-filter scale contribution!

3.10.3 Reynolds-Averaged Navier-Stokes (RANS)

Solve for mean only, model all fluctuations.

Fast but loses all turbulence detail.

3.11 Why ML for Turbulence?

3.11.1 The Perfect ML Problem

Turbulence has characteristics that make it ideal for ML:

1. **High-dimensional**: $O(N^3)$ degrees of freedom
2. **Complex patterns**: Multi-scale, nonlinear
3. **Statistical regularity**: Despite chaos, statistics are reproducible
4. **Data availability**: DNS provides ground truth
5. **Computational need**: Current methods too slow for many applications

3.11.2 What ML Can Do

Task	**Traditional**	**ML Approach**
Subgrid modeling	Eddy viscosity	Neural network closure
Super-resolution	Interpolation	Learned upsampling
Acceleration	Reduced-order models	Neural operators
Turbulence generation	DNS	Generative models

3.11.3 Our Approach

This repository focuses on **scale reconstruction**:

$$\mathbf{u}_{\text{coarse}} \xrightarrow{\text{Neural Network}} \mathbf{u}_{\text{fine}} \tag{3.22}$$

We learn to add back the small-scale features removed by filtering.

3.12 Summary for ML Engineers

3.12.1 Key Concepts

Concept	What it Means	Why it Matters for ML
Reynolds number	Flow complexity parameter	Determines data complexity
Energy spectrum	Energy distribution vs scale	Target for validation
Isotropy	Directional uniformity	Enables data augmentation
Cascade	Energy flow to small scales	What we're reconstructing
Intermittency	Bursty small-scale activity	Affects loss function design

3.12.2 Mapping to Code

Physics	Code Variable
Integral scale L	`params.L_integral`
Kolmogorov scale η	`params.eta`
Taylor Reynolds Re_λ	`params.Re_lambda`
Energy spectrum $E(k)$	`compute_energy_spectrum_3d()`
RMS velocity u'	`params.u_rms`

3.13 Further Reading

3.13.1 Textbooks

1. **Pope**—"Turbulent Flows" (Chapters 5–6)
2. **Tennekes & Lumley**—"A First Course in Turbulence"
3. **Frisch**—"Turbulence: The Legacy of A.N. Kolmogorov"

3.14 Exercises

1. **Reynolds Decomposition**: For $u(t) = 5 + 2\sin(\omega t)$, find $\bar{U}$ and $u'(t)$.
2. **Energy Spectrum**: If $E(k) = Ck^{-5/3}$ for $k \in [1, 100]$, compute $\int E(k)\, dk$.
3. **Scale Ratio**: For $\text{Re}_\lambda = 200$, estimate L/η using the relation $L/\eta \sim \text{Re}_\lambda^{3/2}$.
4. **Isotropy Check**: Given $\langle u'^2 \rangle = 1.2$, $\langle v'^2 \rangle = 0.9$, $\langle w'^2 \rangle = 0.9$, is this isotropic?

Chapter 4

Kolmogorov Theory and Energy Cascade

> *"Kolmogorov's 1941 theory is perhaps the most important result in turbulence research—it provides the only quantitative predictions that are approximately universal."*

4.1 Historical Context

4.1.1 The State of Turbulence Before 1941

Before Kolmogorov:

- Richardson's cascade picture (1922)—qualitative
- Taylor's statistical approach (1935)—mathematical framework
- No quantitative predictions for turbulence structure

4.1.2 Kolmogorov's Breakthrough

In 1941, Andrey Nikolaevich Kolmogorov (Soviet mathematician, age 38) published two papers that revolutionized turbulence theory:

1. "The local structure of turbulence in incompressible viscous fluid for very large Reynolds numbers"—introduced the hypotheses
2. "On degeneration (decay) of isotropic turbulence"—derived the spectral predictions

Key insight: At high Re, small-scale turbulence has **universal** properties independent of how energy is injected.

4.2 The Three Kolmogorov Hypotheses

4.2.1 Hypothesis 1: Local Isotropy

> At sufficiently high Reynolds numbers, the small-scale turbulent motions are statistically isotropic.

What this means:

- Large scales may be anisotropic (depend on geometry, forcing)
- Small scales "forget" the large-scale anisotropy

- Universal small-scale behavior

Mathematical statement: For $r \ll L$:

$$\langle u_i'(\mathbf{x}) u_j'(\mathbf{x}+\mathbf{r}) \rangle \text{ depends only on } |\mathbf{r}|, \text{ not direction} \tag{4.1}$$

4.2.2 Hypothesis 2: Similarity (First Similarity Hypothesis)

In the equilibrium range (inertial + dissipation), statistics depend only on:

- Dissipation rate ε
- Kinematic viscosity ν

Key consequence: We can derive **all** small-scale quantities from just ε and ν using dimensional analysis!

4.2.3 Hypothesis 3: Inertial Range (Second Similarity Hypothesis)

In the inertial range (scales much larger than η, much smaller than L), statistics depend only on ε (not ν).

Physical meaning:

- Viscosity only matters at smallest scales
- In between, pure inertial dynamics
- Universal behavior determined by energy flux alone

4.3 Kolmogorov Scales

4.3.1 Dimensional Analysis

From Hypothesis 2, the only parameters are ε and ν. What unique scales can we form?

Length scale (Kolmogorov microscale):

$$\boxed{\eta = \left(\frac{\nu^3}{\varepsilon}\right)^{1/4}} \tag{4.2}$$

Time scale:

$$\tau_\eta = \left(\frac{\nu}{\varepsilon}\right)^{1/2} \tag{4.3}$$

Velocity scale:

$$u_\eta = (\nu\varepsilon)^{1/4} \tag{4.4}$$

4.3.2 Physical Meaning

At the Kolmogorov scale:

- Local Reynolds number $\mathrm{Re}_\eta = u_\eta \times \eta/\nu = 1$
- Inertia balances viscosity
- Energy dissipates into heat

4.3.3 Numerical Values

For atmospheric turbulence ($\varepsilon \sim 10^{-4}$ m^2/s^3, $\nu \sim 1.5 \times 10^{-5}$ m^2/s):

$$\eta \approx 0.6 \text{ mm} \tag{4.5}$$

$$\tau_\eta \approx 0.4 \text{ s} \tag{4.6}$$

$$u_\eta \approx 0.02 \text{ m/s} \tag{4.7}$$

For laboratory turbulence ($\varepsilon \sim 1$ m^2/s^3):

$$\eta \approx 30 \ \mu\text{m} \tag{4.8}$$

$$\tau_\eta \approx 4 \text{ ms} \tag{4.9}$$

$$u_\eta \approx 0.03 \text{ m/s} \tag{4.10}$$

4.4 The Scale Hierarchy

4.4.1 Three Characteristic Scales

Scale	Symbol	Definition	Typical Ratio
Integral	L	u^3/ε	$L/\eta \sim \text{Re}^{3/4}$
Taylor	λ	$\sqrt{15\nu u^2/\varepsilon}$	$\lambda/\eta \sim \text{Re}^{1/4}$
Kolmogorov	η	$(\nu^3/\varepsilon)^{1/4}$	$= 1$ (reference)

4.4.2 Scale Ratios

The **scale separation** determines the width of the inertial range:

$$L/\eta \sim \text{Re}^{3/4} \quad \text{or equivalently} \quad \sim \text{Re}_\lambda^{3/2} \tag{4.11}$$

Re_λ	L/η	Decades of inertial range
50	350	1.5
100	1000	2.0
200	2800	2.5
500	11000	3.0
1000	32000	3.5

For DNS: Grid must resolve η, so $N \sim L/\eta$. This is why DNS is so expensive.

4.5 The Energy Spectrum: $k^{-5/3}$ Law

4.5.1 Derivation

In the inertial range, by Hypothesis 3:

$$E(k) = \text{function}(\varepsilon, k) \tag{4.12}$$

Dimensional analysis:

- $[E(k)] = \text{m}^3/\text{s}^2$ (energy per unit wavenumber)
- $[\varepsilon] = \text{m}^2/\text{s}^3$
- $[k] = 1/\text{m}$

The only possibility:

$$\boxed{E(k) = C_K \varepsilon^{2/3} k^{-5/3}} \tag{4.13}$$

4.5.2 The Kolmogorov Constant

$$C_K \approx 1.5 \quad \text{(empirically determined)} \tag{4.14}$$

This is **universal**—same for all turbulent flows at high Re.

4.5.3 Experimental Verification

The $k^{-5/3}$ law has been verified in:

- Atmospheric boundary layer
- Ocean mixing
- Wind tunnels
- DNS simulations
- Astrophysical plasmas

It is one of the most precisely verified results in fluid mechanics.

4.5.4 Full Pope Spectrum Model

From Pope, "Turbulent Flows":

$$E(k) = C_K \varepsilon^{2/3} k^{-5/3} f_L(kL) f_\eta(k\eta) \tag{4.15}$$

Where:

- $f_L(kL) \to 1$ as $kL \to \infty$ (correction for large scales)
- $f_\eta(k\eta) \to 1$ as $k\eta \to 0$ (correction for dissipation)

In our code (`src/synthetic_turbulence.py`):

```
def model_spectrum_pope(k, params, C_k=1.5, ...):
    # Large-scale correction
    f_L = (kL / np.sqrt(kL**2 + c_L))**(5/3 + p0)
    # Dissipation correction
    f_eta = np.exp(-beta * (((k_eta)**4 + c_eta**4)**0.25 - c_eta))
    # Full spectrum
    E_k = C_k * epsilon**(2/3) * k**(-5/3) * f_L * f_eta
```

4.6 Structure Functions

4.6.1 Definition

The **n-th order longitudinal structure function**:

$$S_n(r) = \langle [u(\mathbf{x} + \mathbf{r}) - u(\mathbf{x})]^n \rangle \tag{4.16}$$

Where the velocity difference is along the separation direction.

4.6.2 Kolmogorov's Prediction

In the inertial range, by dimensional analysis:

$$S_n(r) \sim (\varepsilon r)^{n/3} \tag{4.17}$$

Specifically:

$$S_2(r) = C_2(\varepsilon r)^{2/3} \quad \text{[Second-order]} \tag{4.18}$$

$$S_3(r) = -\frac{4}{5}\varepsilon r \quad \text{[Third-order—EXACT!]} \tag{4.19}$$

4.6.3 The 4/5 Law

Kolmogorov proved **exactly** (no empirical constants):

$$\boxed{S_3(r) = -\frac{4}{5}\varepsilon r} \tag{4.20}$$

This is the only exact result in turbulence theory!
Physical meaning: Negative sign indicates energy flux from large to small scales.

4.6.4 Structure Function Scaling

Order n	**K41 prediction**	**Observed exponent**
1	1/3	~ 0.37
2	2/3	~ 0.70
3	1	1.00 (exact)
4	4/3	~ 1.28
5	5/3	~ 1.53
6	2	~ 1.78

Deviations from K41 are due to intermittency (Chapter 3).

4.7 The Energy Cascade Mechanism

4.7.1 Physical Picture

Energy flows from large to small scales:

1. **Large eddies**: Size $\sim L$ (integral scale), receive energy from forcing
2. **Inertial transfer**: Energy passes through intermediate scales without dissipation
3. **Small eddies**: Size $\sim \eta$ (Kolmogorov scale), dissipate energy to heat

4.7.2 Energy Balance

In statistically steady state:

$$P = \varepsilon \tag{4.21}$$

(Power input = Dissipation rate)
The energy flux through any scale in the inertial range equals ε.

4.7.3 Eddy Turnover Time

For eddies of size r:

$$\tau(r) \sim \frac{r}{u(r)} \sim \frac{r}{(\varepsilon r)^{1/3}} \sim \frac{r^{2/3}}{\varepsilon^{1/3}} \tag{4.22}$$

Key result: Smaller eddies have shorter lifetimes.

At scale η:

$$\tau_\eta = \frac{\eta^{2/3}}{\varepsilon^{1/3}} = \left(\frac{\nu}{\varepsilon}\right)^{1/2} \tag{4.23}$$

4.8 Spectral Energy Transfer

4.8.1 The Spectral Energy Equation

In Fourier space, energy at wavenumber k evolves as:

$$\frac{\partial E(k)}{\partial t} = T(k) - 2\nu k^2 E(k) + P(k) \tag{4.24}$$

Where:

- $T(k)$ = **spectral transfer function** (nonlinear interactions)
- $2\nu k^2 E(k)$ = **viscous dissipation**
- $P(k)$ = **energy production** (forcing)

4.8.2 Properties of $T(k)$

Conservation:

$$\int_0^\infty T(k)\, dk = 0 \tag{4.25}$$

Energy is redistributed, not created.

Forward cascade (3D):

- $T(k) < 0$ for small k (energy removed)
- $T(k) > 0$ for large k (energy added)

4.8.3 Energy Flux

The energy flux through wavenumber k:

$$\Pi(k) = -\int_0^k T(k')\, dk' \tag{4.26}$$

In the inertial range:

$$\Pi(k) = \varepsilon = \text{constant} \tag{4.27}$$

4.9 Refined Similarity Hypothesis (K62)

4.9.1 The Problem with K41

K41 assumes dissipation ε is constant in space. But observations show:

- ε fluctuates strongly
- Intense dissipation in thin sheets/tubes
- Non-Gaussian statistics

4.9.2 Kolmogorov's 1962 Refinement

Replace constant ε with local average ε_r:

$$S_n(r) \sim \langle \varepsilon_r^{n/3} \rangle r^{n/3} \tag{4.28}$$

4.9.3 Log-Normal Model

Kolmogorov proposed ε_r is log-normally distributed:

$$\ln(\varepsilon_r/\varepsilon) \sim \mathcal{N}(0, \sigma^2 \ln(L/r)) \tag{4.29}$$

This leads to anomalous scaling:

$$S_n(r) \sim r^{\zeta_n} \tag{4.30}$$

Where $\zeta_n = n/3 - \mu n(n-3)/18$ deviates from K41.
Intermittency parameter $\mu \approx 0.25$ (empirical).
In our code (`src/synthetic_turbulence.py`):

```
def add_intermittency(velocity, mu=0.25, seed=None):
    """Add intermittency effects using log-normal model."""
    log_epsilon = np.random.normal(0, np.sqrt(mu * np.log(2)), shape)
    epsilon_local = np.exp(log_epsilon)
    ...
```

4.10 Inverse and Dual Cascades

4.10.1 2D Turbulence: Inverse Cascade

In 2D, there's **no vortex stretching**. Two conserved quantities:

- Energy: $E = \frac{1}{2}\langle u^2 \rangle$
- Enstrophy: $Z = \frac{1}{2}\langle \omega^2 \rangle$

This leads to **dual cascade**:

- Energy cascades to **larger** scales (inverse cascade): $E(k) \sim k^{-5/3}$
- Enstrophy cascades to smaller scales (forward cascade): $E(k) \sim k^{-3}$

4.10.2 3D Turbulence: Forward Only

Only forward cascade of energy (what we simulate).

4.11 Connection to Our Code

4.11.1 TurbulenceParams Class

From `src/synthetic_turbulence.py`:

```
@dataclass
class TurbulenceParams:
    N: int = 128                    # Grid resolution
    L: float = 2 * np.pi            # Domain size
    Re_lambda: float = 100.0        # Taylor Reynolds number
    u_rms: float = 1.0              # RMS velocity
```

```
    k_max_eta_target: float = 1.5   # Resolution requirement

    def __post_init__(self):
        # Kolmogorov scale from grid resolution
        self.eta = self.k_max_eta_target / k_max
        # Integral scale to fit in domain
        self.L_integral = self.L / (2 * np.pi)
        # Dissipation rate from scaling relation
        self.epsilon = self.u_rms**3 / self.L_integral
        # Viscosity from eta = (nu^3/eps)^(1/4)
        self.nu = (self.eta**4 * self.epsilon)**(1/3)
        # Taylor microscale
        self.lambda_taylor = np.sqrt(15*self.nu*self.u_rms**2/self.
    epsilon)
```

4.11.2 Energy Spectrum Validation

From `src/analysis.py`:

```
def compute_energy_spectrum_3d(velocity, L):
    """Compute E(k) and verify k^(-5/3) scaling."""
    # FFT of velocity components
    u_hat = np.fft.fftn(velocity[..., 0])
    ...
    # Shell averaging
    for i in range(num_bins):
        mask = (k_mag >= k_bins[i]) & (k_mag < k_bins[i + 1])
        E_k[i] = np.sum(energy_hat[mask]) / N**6 / dk
```

4.12 Summary: The K41 Framework

4.12.1 Key Results

Quantity	K41 Prediction	Formula
Kolmogorov length	—	$\eta = (\nu^3/\varepsilon)^{1/4}$
Kolmogorov time	—	$\tau_\eta = (\nu/\varepsilon)^{1/2}$
Energy spectrum	Inertial range	$E(k) = C_K \varepsilon^{2/3} k^{-5/3}$
Second structure function	Inertial range	$S_2(r) = C_2(\varepsilon r)^{2/3}$
Third structure function	Exact	$S_3(r) = -\frac{4}{5}\varepsilon r$
Scale ratio	Re dependence	$L/\eta \sim \mathrm{Re}^{3/4}$

4.12.2 Assumptions

1. High Reynolds number ($\mathrm{Re}_\lambda >\sim 100$)
2. Homogeneous, isotropic turbulence
3. Statistically stationary
4. Local isotropy at small scales

4.12.3 Limitations

1. Intermittency deviations at high orders
2. Finite Re effects (limited inertial range)
3. Near-wall or strongly anisotropic regions
4. Transitional turbulence

4.13 Exercises

1. **Kolmogorov Scales**: For $\varepsilon = 0.01$ m^2/s^3 and $\nu = 10^{-6}$ m^2/s (water), calculate η, τ_η, u_η.
2. **Scale Ratio**: What grid resolution N is needed for DNS at $\mathrm{Re}_\lambda = 500$?
3. **Energy Spectrum**: Given $E(k) = 1.5 \times \varepsilon^{2/3} \times k^{-5/3}$, with $\varepsilon = 0.1$, compute E at $k = 10$.
4. **Structure Function**: The 4/5 law states $S_3(r) = -\frac{4}{5}\varepsilon r$. If $S_3 = -0.08$ at $r = 0.1$ m, what is ε?

Chapter 5

Spectral Methods and Fourier Analysis

"Spectral methods are often the method of choice for problems with smooth solutions in simple geometries because of their superior accuracy."

5.1 Why Spectral Methods?

5.1.1 Advantages for Turbulence Simulation

Feature	Finite Difference	Spectral Methods
Accuracy	$O(h^2)$ to $O(h^4)$	**Exponential** (spectral)
Derivatives	Approximate	**Exact** in Fourier space
Periodic BC	Requires ghost cells	**Natural**
Memory	Local stencil	Global (FFT)
Parallelization	Excellent	Good (with care)

5.1.2 Key Insight

Derivatives in physical space become **multiplication in Fourier space**:

$$\frac{\partial u}{\partial x} \to ik\hat{u}(k) \tag{5.1}$$

$$\frac{\partial^2 u}{\partial x^2} \to -k^2\hat{u}(k) \tag{5.2}$$

No approximation needed for derivatives!

5.1.3 When to Use Spectral Methods

Ideal for:

- Periodic boundaries
- Smooth solutions
- Homogeneous turbulence
- High accuracy requirements

Avoid for:

- Complex geometries
- Discontinuities (shocks)
- Non-periodic boundaries (use Chebyshev)

5.2 Fourier Series Fundamentals

5.2.1 Continuous Fourier Transform

For a function $f(x)$ on infinite domain:

$$\hat{f}(k) = \int_{-\infty}^{\infty} f(x)e^{-ikx}\,dx \quad \text{[Forward]} \tag{5.3}$$

$$f(x) = \frac{1}{2\pi}\int_{-\infty}^{\infty} \hat{f}(k)e^{ikx}\,dk \quad \text{[Inverse]} \tag{5.4}$$

5.2.2 Discrete Fourier Transform (DFT)

For N samples on domain $[0, L)$:

$$\hat{f}_n = \sum_{j=0}^{N-1} f_j e^{-i(2\pi nj/N)} \quad \text{[Forward]} \tag{5.5}$$

$$f_j = \frac{1}{N}\sum_{n=0}^{N-1} \hat{f}_n e^{i(2\pi nj/N)} \quad \text{[Inverse]} \tag{5.6}$$

5.2.3 Wavenumbers

Grid points: $x_j = jL/N$ for $j = 0, 1, \ldots, N-1$

Wavenumbers:

$$k_n = \frac{2\pi n}{L} \quad \text{for } n = 0, 1, \ldots, N/2, -N/2+1, \ldots, -1 \tag{5.7}$$

In radians per unit length:

```
k = np.fft.fftfreq(N, d=L/(2*np.pi*N))  # Our code convention
```

Nyquist wavenumber (highest resolvable):

$$k_{\max} = \frac{\pi N}{L} = \frac{\pi}{dx} \tag{5.8}$$

5.3 Fast Fourier Transform (FFT)

5.3.1 The Computational Breakthrough

- **Naive DFT**: $O(N^2)$ operations
- **FFT (Cooley-Tukey, 1965)**: $O(N \log N)$ operations

For $N = 1024$: Speedup of ~ 100x

5.3.2 NumPy FFT Functions

```
import numpy as np

# 1D transforms
f_hat = np.fft.fft(f)          # Forward
f = np.fft.ifft(f_hat)         # Inverse
```

```
# 2D transforms
f_hat = np.fft.fft2(f)
f = np.fft.ifft2(f_hat)

# 3D transforms (what we use)
f_hat = np.fft.fftn(f)
f = np.fft.ifftn(f_hat)

# Wavenumber arrays
k = np.fft.fftfreq(N, d=dx)    # Returns cycles per unit length
```

5.3.3 PyTorch FFT (GPU Accelerated)

```
import torch

# GPU-accelerated FFT
f_hat = torch.fft.fftn(f)
f = torch.fft.ifftn(f_hat)
```

In our code (src/filtering.py):

```
if is_torch:
    v_hat = torch.fft.fftn(velocity[..., i])
    filtered[..., i] = torch.real(torch.fft.ifftn(v_hat * G_hat))
else:
    v_hat = np.fft.fftn(velocity[..., i])
    filtered[..., i] = np.real(np.fft.ifftn(v_hat * G_hat))
```

5.4 Properties of the Fourier Transform

5.4.1 Linearity

$$\mathcal{F}[af + bg] = a\mathcal{F}[f] + b\mathcal{F}[g] \tag{5.9}$$

5.4.2 Differentiation

$$\mathcal{F}\left[\frac{\partial f}{\partial x}\right] = ik\mathcal{F}[f] \tag{5.10}$$

$$\mathcal{F}\left[\frac{\partial^n f}{\partial x^n}\right] = (ik)^n\mathcal{F}[f] \tag{5.11}$$

This is **why spectral methods are so powerful**—derivatives are exact!

5.4.3 Convolution Theorem

$$\mathcal{F}[f * g] = \mathcal{F}[f] \cdot \mathcal{F}[g] \tag{5.12}$$

where

$$f * g = \frac{1}{2\pi}\int f(y)g(x-y)\,dy \tag{5.13}$$

Convolution in physical space = multiplication in Fourier space.
Application: Filtering is multiplication by transfer function.

5.4.4 Parseval's Theorem (Energy Conservation)

$$\int |f(x)|^2 \, dx = \frac{1}{2\pi} \int |\hat{f}(k)|^2 \, dk \tag{5.14}$$

Energy is the same in both domains.

In our code (src/analysis.py):

```
# Energy in Fourier space
energy_hat = 0.5 * (np.abs(u_hat)**2 + np.abs(v_hat)**2 + np.abs(w_hat)
    **2)
# Total energy = sum(energy_hat) / N^6 (normalization from Parseval)
E_k[i] = np.sum(energy_hat[mask]) / N**6 / dk
```

5.5 3D Spectral Representation

5.5.1 Velocity Field in Fourier Space

For velocity $\mathbf{u}(\mathbf{x})$ on cubic domain $[0, L]^3$:

$$\mathbf{u}(\mathbf{x}) = \sum_{\mathbf{k}} \hat{\mathbf{u}}(\mathbf{k}) e^{i\mathbf{k}\cdot\mathbf{x}} \tag{5.15}$$

Where $\mathbf{k} = (k_x, k_y, k_z)$ is the 3D wavevector.

5.5.2 Wavenumber Magnitude

$$|\mathbf{k}| = \sqrt{k_x^2 + k_y^2 + k_z^2} \tag{5.16}$$

Used for shell averaging to compute $E(k)$.

In our code (src/synthetic_turbulence.py):

```
def generate_wavenumbers(N, L, dims=3):
    k1d = np.fft.fftfreq(N, d=L/(2*np.pi*N))
    kx, ky, kz = np.meshgrid(k1d, k1d, k1d, indexing='ij')
    k_mag = np.sqrt(kx**2 + ky**2 + kz**2)
    k_mag[0, 0, 0] = 1.0  # Avoid division by zero
    return kx, ky, kz, k_mag
```

5.5.3 Hermitian Symmetry

For real-valued $\mathbf{u}(\mathbf{x})$, the Fourier transform satisfies:

$$\hat{\mathbf{u}}(-\mathbf{k}) = \hat{\mathbf{u}}^*(\mathbf{k}) \quad \text{[complex conjugate]} \tag{5.17}$$

This means only half the modes are independent.

5.6 Spectral Derivatives

5.6.1 First Derivative

$$\frac{\partial u}{\partial x} \rightarrow ik_x \hat{u}(\mathbf{k}) \tag{5.18}$$

Implementation:

```
# Derivative in x direction
du_dx = np.real(np.fft.ifftn(1j * kx * np.fft.fftn(u)))
```

5.6.2 Second Derivative (Laplacian)

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \to -|\mathbf{k}|^2 \hat{u}(\mathbf{k}) \tag{5.19}$$

Implementation:

```
# Laplacian
k_sq = kx**2 + ky**2 + kz**2
laplacian_u = np.real(np.fft.ifftn(-k_sq * np.fft.fftn(u)))
```

5.6.3 Gradient, Divergence, Curl

Operation	Physical	Spectral
Gradient	$\boldsymbol{\nabla} f$	$i\mathbf{k}\hat{f}$
Divergence	$\boldsymbol{\nabla} \cdot \mathbf{u}$	$i\mathbf{k} \cdot \hat{\mathbf{u}}$
Curl	$\boldsymbol{\nabla} \times \mathbf{u}$	$i\mathbf{k} \times \hat{\mathbf{u}}$
Laplacian	$\nabla^2 f$	$-k^2 \hat{f}$

5.7 The Divergence-Free Projection

5.7.1 Why It Matters

Incompressibility requires $\boldsymbol{\nabla} \cdot \mathbf{u} = 0$, which in Fourier space is:

$$\mathbf{k} \cdot \hat{\mathbf{u}} = 0 \tag{5.20}$$

The velocity Fourier modes must be **perpendicular** to their wavevector.

5.7.2 Helmholtz Decomposition

Any vector field can be decomposed:

$$\mathbf{u} = \mathbf{u}_{\text{solenoidal}} + \mathbf{u}_{\text{irrotational}} \tag{5.21}$$

Where:

- $\mathbf{u}_{\text{solenoidal}}$: $\boldsymbol{\nabla} \cdot \mathbf{u}_{\text{sol}} = 0$ (divergence-free)
- $\mathbf{u}_{\text{irrotational}}$: $\boldsymbol{\nabla} \times \mathbf{u}_{\text{irr}} = 0$ (curl-free)

5.7.3 Projection Operator

In Fourier space, project onto divergence-free component:

$$\hat{\mathbf{u}}_{\text{sol}} = \left(\mathbf{I} - \frac{\mathbf{k}\mathbf{k}^T}{|\mathbf{k}|^2} \right) \hat{\mathbf{u}}_{\text{raw}} = \hat{\mathbf{u}}_{\text{raw}} - \mathbf{k} \frac{\mathbf{k} \cdot \hat{\mathbf{u}}_{\text{raw}}}{|\mathbf{k}|^2} \tag{5.22}$$

In our code (`src/synthetic_turbulence.py`):

```
# Project to divergence-free using Helmholtz decomposition
k_sq = kx**2 + ky**2 + kz**2
k_sq_safe = np.where(k_sq > 0, k_sq, 1.0)
k_dot_u = kx * u_hat + ky * v_hat + kz * w_hat
proj_factor = k_dot_u / k_sq_safe

u_hat = np.where(k_sq > 0, u_hat - kx * proj_factor, 0)
v_hat = np.where(k_sq > 0, v_hat - ky * proj_factor, 0)
w_hat = np.where(k_sq > 0, w_hat - kz * proj_factor, 0)
```

5.7.4 Energy Loss from Projection

Projection removes 1/3 of the degrees of freedom (1 out of 3 components):

```
# Compensate for energy loss: multiply by sqrt(3/2)
u_hat = np.where(k_sq > 0, (u_hat - kx * proj_factor) * np.sqrt(3/2), 0)
```

5.8 Spectral Filtering

5.8.1 The Filtering Operation

Filtering removes small-scale content:

$$\bar{u}(\mathbf{x}) = \int G(\mathbf{x} - \mathbf{x}')u(\mathbf{x}')\, d\mathbf{x}' \quad \text{[Convolution]} \tag{5.23}$$

In Fourier space:

$$\hat{u}_{\text{filtered}}(\mathbf{k}) = \hat{G}(\mathbf{k})\hat{u}(\mathbf{k}) \quad \text{[Multiplication]} \tag{5.24}$$

Where $\hat{G}(\mathbf{k})$ is the **filter transfer function**.

5.8.2 Common Filter Types

Sharp Spectral (Box) Filter

$$\hat{G}(k) = \begin{cases} 1 & \text{if } |k| \leq k_c \\ 0 & \text{otherwise} \end{cases} \tag{5.25}$$

In code (`src/filtering.py`):

```
def sharp_spectral_filter_kernel(k_mag, k_cutoff):
    return (k_mag <= k_cutoff).astype(np.float64)
```

Properties:

- Ideal frequency cutoff
- Gibbs phenomenon in physical space
- Used for LES with known cutoff

Gaussian Filter

$$\hat{G}(k) = \exp\left(-\frac{k^2 R^2}{24}\right) \tag{5.26}$$

In code:

```
def gaussian_filter_kernel_spectral(k_mag, R):
    return np.exp(-k_mag**2 * R**2 / 24)
```

Properties:

- Smooth in both domains
- No Gibbs phenomenon
- Most physical

Box (Top-Hat) Filter

$$\hat{G}(k) = \frac{3(\sin(kR) - kR\cos(kR))}{(kR)^3} \quad \text{[3D spherical]} \tag{5.27}$$

In code:

```
def box_filter_kernel_spectral(k_mag, R, dims=3):
    kR = k_mag * R + 1e-10
    return 3 * (np.sin(kR) - kR * np.cos(kR)) / (kR**3)
```

5.9 Shell Averaging for $E(k)$

5.9.1 The Energy Spectrum

The 3D energy spectrum $E(k)$ is defined by shell-averaging:

$$E(k)\,dk = (\text{energy in shell from } k \text{ to } k + dk) \tag{5.28}$$

5.9.2 Algorithm

1. Compute 3D energy density: $e(\mathbf{k}) = \frac{1}{2}|\hat{\mathbf{u}}|^2$
2. Bin wavenumber magnitudes into shells
3. Sum energy in each shell
4. Divide by shell width dk

In code (src/analysis.py):

```
def compute_energy_spectrum_3d(velocity, L):
    # FFT
    u_hat = np.fft.fftn(velocity[..., 0])
    v_hat = np.fft.fftn(velocity[..., 1])
    w_hat = np.fft.fftn(velocity[..., 2])

    # Spectral energy density
    energy_hat = 0.5 * (np.abs(u_hat)**2 + np.abs(v_hat)**2
                        + np.abs(w_hat)**2)

    # Wavenumber grid
    k1d = np.fft.fftfreq(N, d=dx) * 2 * np.pi
    kx, ky, kz = np.meshgrid(k1d, k1d, k1d, indexing='ij')
    k_mag = np.sqrt(kx**2 + ky**2 + kz**2)

    # Shell averaging
    for i in range(num_bins):
        mask = (k_mag >= k_bins[i]) & (k_mag < k_bins[i + 1])
        E_k[i] = np.sum(energy_hat[mask]) / N**6 / dk

    return k_centers, E_k
```

5.10 Pseudo-Spectral Methods

5.10.1 The Nonlinear Term Problem

The Navier-Stokes convective term $(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}$ involves multiplication:

- In Fourier space: **convolution** (expensive: $O(N^6)$)
- In physical space: **multiplication** (cheap: $O(N^3)$)

5.10.2 Pseudo-Spectral Algorithm

Evaluate nonlinear terms in physical space:

1. Transform $\mathbf{u}$ from spectral $\rightarrow$ physical
2. Compute $(\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u}$ in physical space
3. Transform result from physical $\rightarrow$ spectral

Total cost: $O(N^3 \log N)$ instead of $O(N^6)$

5.10.3 Aliasing Error

Multiplication of two modes at k_1 and k_2 creates energy at $k_1 + k_2$.
If $k_1 + k_2 > k_{\max}$, this energy **aliases** back into resolved range.

5.10.4 Dealiasing (2/3 Rule)

Zero-pad the spectrum to $3N/2$ before transforming:

```
# Dealias using 2/3 rule
k_dealias = 2/3 * k_max
u_hat[k_mag > k_dealias] = 0
```

5.11 Connection to Machine Learning

5.11.1 Fourier Features in Neural Networks

Random Fourier Features:

```
# Map low-dimensional input to high-dimensional Fourier features
features = [sin(2*pi * sigma_i * B @ x), cos(2*pi * sigma_i * B @ x)]
```

This helps networks learn high-frequency functions.

5.11.2 Fourier Neural Operator (FNO)

The FNO applies learned filters in Fourier space:

$$u \xrightarrow{\text{FFT}} (\text{learned spectral weights}) \xrightarrow{\text{iFFT}} v \tag{5.29}$$

Key advantage: Global receptive field in single layer.

5.11.3 Our Approach: Hybrid

We use spectral methods for:

- Turbulence generation (exact physics)
- Filtering (exact coarse-graining)
- Analysis (energy spectrum)

Neural network learns the residual/correction.

5.12 Practical Considerations

5.12.1 FFT Normalization

Different libraries use different normalizations:

Library	Forward	Inverse	Our convention
NumPy	$\Sigma f \cdot e^{-ikx}$	$(1/N)\Sigma \hat{f} \cdot e^{ikx}$	✓
FFTW	Same	Same	—
Torch	Same	Same	✓

Parseval relation with NumPy:

$$\int |f|^2 \, dx \approx \frac{1}{N} \sum |f_j|^2 = \frac{1}{N^3} \sum |\hat{f}_n|^2 \tag{5.30}$$

5.12.2 Memory for 3D FFT

For N^3 complex array:

$$\text{Memory} = N^3 \times 16 \text{ bytes (complex128)} \tag{5.31}$$

N	Memory
64	4 MB
128	32 MB
256	256 MB
512	2 GB
1024	16 GB

5.13 Summary

5.13.1 Key Formulas

Operation	Physical Space	Fourier Space
Derivative	$\partial f/\partial x$	$ik\hat{f}$
Laplacian	$\nabla^2 f$	$-k^2\hat{f}$
Filtering	$\int G \cdot f$	$\hat{G} \cdot \hat{f}$
Divergence-free	$\boldsymbol{\nabla} \cdot \mathbf{u} = 0$	$\mathbf{k} \cdot \hat{\mathbf{u}} = 0$
Energy spectrum	$\frac{1}{2}\langle u^2 \rangle$	Shell-average $\frac{1}{2}\lvert\hat{\mathbf{u}}\rvert^2$

5.13.2 Code Quick Reference

```python
import numpy as np

# Setup
N = 128; L = 2*np.pi; dx = L/N
k1d = np.fft.fftfreq(N, d=dx) * 2*np.pi
kx, ky, kz = np.meshgrid(k1d, k1d, k1d, indexing='ij')
k_mag = np.sqrt(kx**2 + ky**2 + kz**2)

# Forward/inverse
u_hat = np.fft.fftn(u)
u = np.real(np.fft.ifftn(u_hat))

# Derivative
du_dx = np.real(np.fft.ifftn(1j * kx * u_hat))

# Filtering
G_hat = np.exp(-k_mag**2 * R**2 / 24)  # Gaussian
u_filtered = np.real(np.fft.ifftn(G_hat * u_hat))
```

5.14 Exercises

1. **Derivative**: Compute the spectral derivative of $f(x) = \sin(4x)$ on $[0, 2\pi]$ with $N = 32$.
2. **Filtering**: Apply a Gaussian filter with $R = 0.5$ to random noise. Plot before/after spectra.
3. **Divergence-free**: Generate random 3D vector field, project to divergence-free, verify $\boldsymbol{\nabla} \cdot \mathbf{u} \approx 0$.
4. **Energy Spectrum**: Compute $E(k)$ for synthetic turbulence and fit power law in inertial range.

Part II

Code Implementation

Chapter 6

Synthetic Turbulence Generation

This chapter provides a detailed explanation of how we generate realistic turbulent velocity fields for training data.

6.1 Overview

6.1.1 Purpose of This Module

Generate **synthetic DNS-like turbulent velocity fields** with:

- Correct statistical properties (energy spectrum, correlations)
- Divergence-free (incompressible)
- Controllable parameters (Re_λ, u_{rms}, resolution)
- Reproducibility (seed control)

6.1.2 Why Synthetic Data?

Real DNS	**Synthetic**
Expensive ($millions in compute)	Cheap (seconds)
Limited availability	Unlimited samples
Complex storage	Generated on demand
Fixed parameters	Tunable parameters

Trade-off: Synthetic captures statistics, not exact N-S dynamics.

6.2 Module Structure

```
src/synthetic_turbulence.py
+-- TurbulenceParams (dataclass)
+-- compute_kolmogorov_scales()
+-- generate_wavenumbers()
+-- model_spectrum_pope()
+-- model_spectrum_simple()
+-- generate_isotropic_turbulence_spectral()
+-- _ensure_hermitian_2d/3d()
+-- diagnose_spectrum()
+-- add_intermittency()
+-- SyntheticTurbulenceDataset (PyTorch)
```

6.3 TurbulenceParams: Configuration Class

6.3.1 The Dataclass

```python
@dataclass
class TurbulenceParams:
    """Parameters for synthetic turbulence generation."""
    N: int = 128                    # Grid resolution (N x N x N)
    L: float = 2 * np.pi            # Domain size [m]
    Re_lambda: float = 100.0        # Taylor Reynolds number
    u_rms: float = 1.0              # Target RMS velocity [m/s]
    dims: int = 3                   # 2D or 3D
    seed: Optional[int] = None      # For reproducibility
    k_max_eta_target: float = 1.5   # Resolution parameter

    # Derived (computed in __post_init__)
    nu: float = None                # Kinematic viscosity
    epsilon: float = None           # Dissipation rate
    lambda_taylor: float = None     # Taylor microscale
    eta: float = None               # Kolmogorov scale
    L_integral: float = None        # Integral scale
```

6.3.2 Key Relationships

From Chapter 4 (Kolmogorov theory):

$$\eta = (\nu^3/\varepsilon)^{1/4} \quad \text{Kolmogorov scale} \tag{6.1}$$

$$\lambda = \sqrt{15\nu u'^2/\varepsilon} \quad \text{Taylor microscale} \tag{6.2}$$

$$\mathrm{Re}_\lambda = u'\lambda/\nu \quad \text{Taylor Reynolds number} \tag{6.3}$$

$$L = u'^3/\varepsilon \quad \text{Integral scale} \tag{6.4}$$

6.3.3 Automatic Derivation in `__post_init__`

```python
def __post_init__(self):
    # Grid parameters
    dx = self.L / self.N
    k_max = np.pi / dx  # Nyquist wavenumber

    # Set eta to resolve dissipation: k_max x eta = target
    self.eta = self.k_max_eta_target / k_max

    # Integral scale fits in domain
    self.L_integral = self.L / (2 * np.pi)

    # Dissipation from u'^3/L (dimensional analysis)
    self.epsilon = self.u_rms**3 / self.L_integral

    # Viscosity from eta definition
    self.nu = (self.eta**4 * self.epsilon)**(1/3)

    # Taylor microscale from dissipation relation
    self.lambda_taylor = np.sqrt(15 * self.nu * self.u_rms**2
                                 / self.epsilon)
```

6.3.4 Usage Example

```python
from src.synthetic_turbulence import TurbulenceParams

# Create parameters
params = TurbulenceParams(
    N=128,           # 128^3 grid
    Re_lambda=200,   # Moderate Reynolds number
    u_rms=1.0,       # Unit RMS velocity
    seed=42          # Reproducible
)

print(f"Kolmogorov scale eta = {params.eta:.4e}")
print(f"Integral scale L = {params.L_integral:.4f}")
```

6.4 Wavenumber Generation

6.4.1 The Function

```python
def generate_wavenumbers(N: int, L: float, dims: int = 3):
    """Generate wavenumber arrays for spectral methods."""
```

6.4.2 Wavenumber Convention

Wavenumbers are in **radians per unit length**:

```python
k1d = np.fft.fftfreq(N, d=L/(2*np.pi*N))
```

This gives $k = [0, 1, 2, \ldots, N/2-1, -N/2, \ldots, -1] \times (2\pi/L)$

6.4.3 3D Mesh Generation

```python
kx, ky, kz = np.meshgrid(k1d, k1d, k1d, indexing='ij')
k_mag = np.sqrt(kx**2 + ky**2 + kz**2)
k_mag[0, 0, 0] = 1.0  # Avoid division by zero
return kx, ky, kz, k_mag
```

6.5 Energy Spectrum Models

6.5.1 Pope's Model Spectrum

From Pope's textbook (Eq. 6.246):

$$E(k) = C_K \varepsilon^{2/3} k^{-5/3} f_L(kL) f_\eta(k\eta) \tag{6.5}$$

Large-scale correction f_L:

$$f_L(kL) = \left(\frac{kL}{\sqrt{k^2 L^2 + c_L}} \right)^{5/3+p_0} \tag{6.6}$$

Dissipation correction f_η:

$$f_\eta(k\eta) = \exp\left(-\beta \left[(k^4\eta^4 + c_\eta^4)^{1/4} - c_\eta \right] \right) \tag{6.7}$$

6.5.2 Implementation

```
def model_spectrum_pope(k, params, C_k=1.5, p0=2.0, beta=5.2, c_eta
    =0.40):
    eta = params.eta
    L_eff = params.L_integral
    epsilon = params.epsilon

    k_safe = np.maximum(k, 1e-10)
    kL = k_safe * L_eff
    k_eta = k_safe * eta

    # Large-scale shape function
    c_L = 6.78
    f_L = (kL / np.sqrt(kL**2 + c_L))**(5/3 + p0)

    # Dissipation-range shape function
    f_eta = np.exp(-beta * (((k_eta)**4 + c_eta**4)**0.25 - c_eta))

    # Full spectrum
    E_k = C_k * epsilon**(2/3) * k_safe**(-5/3) * f_L * f_eta
    E_k = np.where(k > 0, E_k, 0)  # Zero k=0 mode

    return E_k
```

6.6 Isotropic Turbulence Generation

6.6.1 The Main Function

```
def generate_isotropic_turbulence_spectral(params: TurbulenceParams) ->
    np.ndarray:
    """
    Generate synthetic isotropic turbulence using spectral method.

    Returns velocity field of shape (N, N, N, 3)
    """
```

6.6.2 Algorithm Steps

Step 1: Set Random Seed

```
if params.seed is not None:
    np.random.seed(params.seed)
```

Step 2: Generate Wavenumbers

```
kx, ky, kz, k_mag = generate_wavenumbers(N, L, dims=3)
```

Step 3: Target Energy Spectrum

```
E_k_target = model_spectrum(k_mag, params)
```

Step 4: Mode Amplitudes

The energy spectrum $E(k)$ is energy per unit wavenumber. For individual modes:

```
# Mode spacing
dk = 2 * np.pi / L

# Energy per mode (accounting for shell volume ~ 4*pi*k^2*dk^2)
mode_energy_total = E_k_target * dk**3 / (4 * np.pi * k_mag**2 + 1e-10)

# Split among 3 velocity components
mode_energy_per_component = mode_energy_total / 3.0

# Amplitude for complex Gaussian
amplitude = np.sqrt(np.maximum(mode_energy_per_component, 0))
amplitude[0, 0, 0] = 0  # Zero mean velocity
```

Step 5: Random Complex Gaussian Modes

```
u_hat = amplitude * (np.random.randn(N,N,N) + 1j*np.random.randn(N,N,N))
    / np.sqrt(2)
v_hat = amplitude * (np.random.randn(N,N,N) + 1j*np.random.randn(N,N,N))
    / np.sqrt(2)
w_hat = amplitude * (np.random.randn(N,N,N) + 1j*np.random.randn(N,N,N))
    / np.sqrt(2)
```

Why divide by $\sqrt{2}$? For $z = (X + iY)/\sqrt{2}$ with $X, Y \sim N(0, \sigma^2)$, we get $\langle |z|^2 \rangle = \sigma^2$.

Step 6: Divergence-Free Projection

Remove the compressible part:

```
k_sq = kx**2 + ky**2 + kz**2
k_sq_safe = np.where(k_sq > 0, k_sq, 1.0)

# Dot product k.u_hat
k_dot_u = kx * u_hat + ky * v_hat + kz * w_hat
proj_factor = k_dot_u / k_sq_safe

# Project: u_hat_sol = u_hat - k(k.u_hat)/|k|^2
# Compensate for 1/3 energy loss with sqrt(3/2)
u_hat = np.where(k_sq > 0, (u_hat - kx * proj_factor) * np.sqrt(3/2), 0)
v_hat = np.where(k_sq > 0, (v_hat - ky * proj_factor) * np.sqrt(3/2), 0)
w_hat = np.where(k_sq > 0, (w_hat - kz * proj_factor) * np.sqrt(3/2), 0)
```

Step 7: Ensure Hermitian Symmetry

For real output:

```
u_hat = _ensure_hermitian_3d(u_hat)
v_hat = _ensure_hermitian_3d(v_hat)
w_hat = _ensure_hermitian_3d(w_hat)
```

Step 8: Transform to Physical Space

```
u = np.real(np.fft.ifftn(u_hat)) * N**3
v = np.real(np.fft.ifftn(v_hat)) * N**3
w = np.real(np.fft.ifftn(w_hat)) * N**3

velocity = np.stack([u, v, w], axis=-1)
```

Step 9: Normalize to Target RMS

```
u_squared = np.sum(velocity**2, axis=-1)
u_rms_actual = np.sqrt(np.mean(u_squared))
velocity *= params.u_rms / (u_rms_actual + 1e-10)
```

6.7 Hermitian Symmetry

6.7.1 Why Required?

For the inverse FFT to give real values, the spectrum must satisfy:

$$\hat{\mathbf{u}}(-\mathbf{k}) = \hat{\mathbf{u}}^*(\mathbf{k}) \tag{6.8}$$

6.7.2 Implementation

```
def _ensure_hermitian_3d(f_hat):
    """Ensure 3D array has Hermitian symmetry."""
    N = f_hat.shape[0]
    # Special points that must be real
    for idx in [(0,0,0), (N//2,0,0), (0,N//2,0), (0,0,N//2),
                (N//2,N//2,0), (N//2,0,N//2), (0,N//2,N//2),
                (N//2,N//2,N//2)]:
        f_hat[idx] = np.real(f_hat[idx])
    return f_hat
```

6.8 Diagnostic Tools

6.8.1 Spectrum Diagnosis

```
def diagnose_spectrum(velocity, params, plot=True):
    """Compute E(k) and verify -5/3 scaling."""
```

Output includes:

- `inertial_exponent`: Fitted slope (should be ≈ -1.667)
- `r_squared`: Quality of fit
- `k_inertial_min`, `k_inertial_max`: Inertial range bounds
- `E_k`, `k_centers`: Spectrum data

6.9 Common Usage Patterns

6.9.1 Generate Single Field

```python
from src.synthetic_turbulence import TurbulenceParams,
    generate_isotropic_turbulence_spectral

params = TurbulenceParams(N=64, Re_lambda=100, seed=42)
velocity = generate_isotropic_turbulence_spectral(params)
print(f"Shape: {velocity.shape}")  # (64, 64, 64, 3)
```

6.9.2 Verify Divergence-Free

```python
dx = params.L / params.N
div = (np.gradient(velocity[...,0], dx, axis=0) +
       np.gradient(velocity[...,1], dx, axis=1) +
       np.gradient(velocity[...,2], dx, axis=2))
print(f"Max divergence: {np.max(np.abs(div)):.2e}")  # Should be ~0
```

6.10 Limitations and Caveats

6.10.1 What This Does NOT Capture

1. **Time dynamics**: Only instantaneous snapshots
2. **Exact N-S solutions**: Statistics match, not exact physics
3. **Non-Gaussian features**: Limited intermittency
4. **Coherent structures**: No vortex tubes, sheets

6.10.2 When This Is Sufficient

✓ Training data for super-resolution

✓ Testing filtering/analysis code

✓ Statistical studies

✓ Initial conditions

6.11 Summary

6.11.1 Key Functions

Function	Purpose
`TurbulenceParams`	Configure generation parameters
`generate_isotropic_turbulence_spectral`	Main generation routine
`diagnose_spectrum`	Verify $k^{-5/3}$ scaling
`print_turbulence_parameters`	Display derived quantities

6.11.2 Key Concepts Implemented

1. **Spectral synthesis**: Build velocity in Fourier space
2. **Kolmogorov scaling**: Energy spectrum $E(k) \sim k^{-5/3}$
3. **Divergence-free projection**: Helmholtz decomposition
4. **Hermitian symmetry**: For real output

Chapter 7

Spectral Filtering

This chapter explains how we coarse-grain DNS fields using spectral filters—the key operation that creates our training data pairs.

7.1 Overview

7.1.1 The Fundamental Operation

Filtering removes small-scale information from a turbulent field:

$$\text{DNS velocity field } \mathbf{V}_{\text{fine}}(\mathbf{x}) \xrightarrow{G_R} \mathbf{V}_{\text{coarse}}(\mathbf{x}) \tag{7.1}$$

Mathematically:

$$\bar{V}_R(\mathbf{x}) = \int G_R(\mathbf{x} - \mathbf{y}) V(\mathbf{y})\, d\mathbf{y} \quad \text{[Convolution]} \tag{7.2}$$

7.1.2 Why Filtering Matters

In Large Eddy Simulation (LES):

- Resolve large scales explicitly
- **Model** small scales (sub-filter)

Our neural network learns:

$$\mathbf{V}_{\text{coarse}} \xrightarrow{\text{NN}} \delta\mathbf{V} = \mathbf{V}_{\text{fine}} - \mathbf{V}_{\text{coarse}} \tag{7.3}$$

The network predicts what was removed by filtering!

7.2 Module Structure

```
src/filtering.py
+-- FilterType (Enum)
+-- FilterConfig (dataclass)
+-- Filter kernels:
|   +-- gaussian_filter_kernel_spectral()
|   +-- box_filter_kernel_spectral()
|   +-- sharp_spectral_filter_kernel()
+-- get_filter_kernel_spectral()
+-- SpectralFilter (class)
+-- MultiScaleDecomposition (class)
+-- prepare_training_data()
```

7.3 Filter Types

7.3.1 Enum Definition

```
class FilterType(Enum):
    GAUSSIAN = "gaussian"
    BOX = "box"
    SHARP_SPECTRAL = "sharp_spectral"
```

7.3.2 Which to Use?

Filter	Spectral	Physical	Use Case
Gaussian	Smooth decay	Gaussian blob	Default, most physical
Box	Oscillatory	Sharp box average	Physical space averaging
Sharp	Step function	Gibbs ringing	Exact cutoff analysis

7.4 Gaussian Filter

7.4.1 Theory

Physical space: Gaussian convolution kernel

$$G_R(r) = \left(\frac{6}{\pi R^2}\right)^{3/2} \exp\left(-\frac{6r^2}{R^2}\right) \tag{7.4}$$

Spectral space: Also Gaussian!

$$\hat{G}(k) = \exp\left(-\frac{k^2 R^2}{24}\right) \tag{7.5}$$

The factor 24 ensures the filter width R corresponds to the second moment.

7.4.2 Implementation

```
def gaussian_filter_kernel_spectral(k_mag: np.ndarray, R: float) -> np.
    ndarray:
    r"""
    Gaussian filter transfer function in spectral space.

    G_hat(k) = exp(-k^2 R^2 / 24)
    """
    return np.exp(-k_mag**2 * R**2 / 24)
```

7.4.3 Properties

- **Positive**: $\hat{G}(k) > 0$ for all k
- **Monotonic**: Smooth decay, no ringing
- **Self-similar**: Gaussian in both domains
- **Recommended for most uses**

7.5 Box (Top-Hat) Filter

7.5.1 Theory

Physical space: Average over sphere of radius R

$$G_R(r) = \frac{3}{4\pi R^3} \text{ if } |r| < R, \text{ else } 0 \tag{7.6}$$

Spectral space (3D spherical):

$$\hat{G}(k) = \frac{3[\sin(kR) - kR\cos(kR)]}{(kR)^3} \tag{7.7}$$

7.5.2 Implementation

```
def box_filter_kernel_spectral(k_mag: np.ndarray, R: float, dims: int =
    3) -> np.ndarray:
    r"""
    Box (top-hat) filter transfer function.

    3D spherical: G_hat(k) = 3(sin(kR) - kR*cos(kR))/(kR)^3
    """
    kR = k_mag * R + 1e-10  # Avoid division by zero

    if dims == 1:
        return np.sinc(kR / (2 * np.pi))
    elif dims == 2:
        from scipy.special import j1
        return 2 * j1(kR) / kR
    else:  # 3D spherical
        return 3 * (np.sin(kR) - kR * np.cos(kR)) / (kR**3)
```

7.6 Sharp Spectral Filter

7.6.1 Theory

Ideal low-pass filter:

$$\hat{G}(k) = \begin{cases} 1 & \text{if } |k| \leq k_c \\ 0 & \text{otherwise} \end{cases} \tag{7.8}$$

Where $k_c = \pi/R$ is the cutoff wavenumber.

7.6.2 Implementation

```
def sharp_spectral_filter_kernel(k_mag: np.ndarray, k_cutoff: float) ->
    np.ndarray:
    r"""
    Sharp spectral cutoff filter.

    G_hat(k) = 1 if |k| <= k_cutoff, else 0
    """
    return (k_mag <= k_cutoff).astype(np.float64)
```

7.6.3 Properties

- **Exact frequency cutoff**: No energy above k_c
- **Gibbs phenomenon**: Ringing in physical space
- **Not physically realizable** in experiments
- **Useful for analysis** of scale separation

7.7 SpectralFilter Class

7.7.1 Initialization

```
class SpectralFilter:
    def __init__(self, N: int, L: float = 2*np.pi, dims: int = 3,
                 device: str = 'cpu'):
        self.N = N
        self.L = L
        self.dims = dims
        self.device = device
        self.dx = L / N
        self._setup_wavenumbers()
```

7.7.2 Main Filter Method

```
def filter_field(self, velocity, R, filter_type=FilterType.GAUSSIAN):
    """
    Apply filter to velocity field.

    Args:
        velocity: Shape (..., N, N, N, 3)
        R: Filter size (in same units as domain)
        filter_type: Type of filter kernel

    Returns:
        Filtered velocity field (same shape)
    """
    is_torch = isinstance(velocity, torch.Tensor)

    if is_torch:
        return self._filter_torch(velocity, R, filter_type)
    else:
        return self._filter_numpy(velocity, R, filter_type)
```

7.7.3 NumPy Implementation

```
def _filter_numpy(self, velocity, R, filter_type):
    # Get filter kernel
    G_hat = get_filter_kernel_spectral(self.k_mag, R, filter_type, self.
   dims)

    num_components = velocity.shape[-1]
    filtered = np.zeros_like(velocity)
```

```
    for i in range(num_components):
        # FFT -> multiply by filter -> inverse FFT
        v_hat = np.fft.fftn(velocity[..., i])
        filtered[..., i] = np.real(np.fft.ifftn(v_hat * G_hat))

    return filtered
```

7.8 Multi-Scale Hierarchy

7.8.1 Creating Scale Hierarchy

```
def create_multiscale_hierarchy(self, velocity, filter_sizes,
    filter_type):
    """
    Create hierarchy of filtered fields at multiple scales.

    Args:
        velocity: DNS field
        filter_sizes: [R1, R2, ...] from largest to smallest
        filter_type: Type of filter

    Returns:
        [V_R1, V_R2, ...] plus original
    """
    hierarchy = []
    for R in filter_sizes:
        V_R = self.filter_field(velocity, R, filter_type)
        hierarchy.append(V_R)
    return hierarchy
```

7.8.2 Example Usage

```
filter_sizes = [0.5, 0.25, 0.125]  # Coarse to fine

hierarchy = spectral_filter.create_multiscale_hierarchy(
    velocity, filter_sizes, FilterType.GAUSSIAN
)

# hierarchy[0]: Most filtered (coarsest)
# hierarchy[1]: Medium
# hierarchy[2]: Least filtered (finest)
```

7.9 Sub-Filter Scale Quantities

7.9.1 Sub-Filter Field

The "missing" information:

```
def compute_subfilter_field(self, V_coarse, V_fine):
    """V_fine - V_coarse: What filtering removed."""
    return V_fine - V_coarse
```

7.9.2 Sub-Filter Stress Tensor

The Leonard decomposition:

$$\tau_{ij} = \overline{u_i u_j}_R - \bar{u}_i^R \bar{u}_j^R \tag{7.9}$$

This is what LES subgrid models try to represent.

```
def compute_subfilter_stress(self, velocity, R, filter_type):
    """
    Compute sub-filter scale stress tensor.

    tau_ij = <u_i u_j>_R - <u_i>_R <u_j>_R
    """
    V_filtered = self.filter_field(velocity, R, filter_type)

    tau = np.zeros((*velocity.shape[:-1], 3, 3))

    for i in range(3):
        for j in range(i, 3):
            # Filter the product
            uiuj = velocity[..., i] * velocity[..., j]
            uiuj_filtered = self.filter_field(uiuj[..., None], R,
                                              filter_type)[..., 0]
            # Product of filtered
            UiUj = V_filtered[..., i] * V_filtered[..., j]
            # Sub-filter stress
            tau[..., i, j] = uiuj_filtered - UiUj
            if i != j:
                tau[..., j, i] = tau[..., i, j]  # Symmetric

    return tau
```

7.10 Training Data Preparation

7.10.1 The Key Function

```
def prepare_training_data(velocity, R1, R2, spectral_filter, filter_type
    , device):
    """
    Prepare data for training the scale reconstruction network.

    Args:
        velocity: DNS velocity field
        R1: Coarse filter size (larger)
        R2: Fine filter size (smaller, R2 < R1)

    Returns:
        dict with:
        - V1: Coarse filtered
        - V2: Fine filtered
        - delta_V: V2 - V1 (target for reconstruction)
        - coordinates: Position grid
    """
    V1 = spectral_filter.filter_field(velocity, R1, filter_type)
    V2 = spectral_filter.filter_field(velocity, R2, filter_type)
    delta_V = V2 - V1  # What we want to predict!
```

```
    return {
        'V1': V1,           # Input to network
        'V2': V2,           # Ground truth fine
        'delta_V': delta_V, # Target for network
        'R1': R1,
        'R2': R2
    }
```

7.10.2 Usage Pattern

```
from src.filtering import SpectralFilter, FilterType,
    prepare_training_data
from src.synthetic_turbulence import TurbulenceParams,
    generate_isotropic_turbulence_spectral

# Generate DNS
params = TurbulenceParams(N=128, Re_lambda=200, seed=42)
velocity = generate_isotropic_turbulence_spectral(params)

# Set up filter
spectral_filter = SpectralFilter(params.N, params.L, params.dims)

# Prepare training data
data = prepare_training_data(
    velocity,
    R1=0.5,   # Coarse
    R2=0.25,  # Fine
    spectral_filter=spectral_filter,
    filter_type=FilterType.GAUSSIAN,
    device='cuda'
)

# Training pairs
input_coarse = data['V1']      # Network input
target_delta = data['delta_V']  # Network should predict this
```

7.11 Filter Size Selection

7.11.1 Physical Interpretation

Filter size R corresponds to:

- **Physical length scale** smoothed away
- **Wavenumber cutoff** $k_c \approx \pi/R$

7.11.2 Typical Values

R (fraction of L)	What it filters
$L/4$	All but largest scales
$L/8$	Medium scales
$L/16$	Small scales
$L/32$	Only finest scales

7.11.3 Choosing R_1 and R_2

For training:

```
R1 = 0.5    # Coarse: filters more
R2 = 0.25   # Fine: filters less
ratio = R1/R2  # = 2, typical ratio
```

The network learns to add back the scales between R_2 and R_1.

7.12 Summary

7.12.1 Key Concepts

Concept	Meaning
Filter transfer function $\hat{G}(k)$	How much each wavenumber is attenuated
Filter size R	Characteristic smoothing length
Sub-filter scales	Information removed by filtering
Scale decomposition	Hierarchy of filtered fields

7.12.2 Key Functions

Function	Purpose
`SpectralFilter.filter_field()`	Main filtering operation
`SpectralFilter.create_multiscale_hierarchy()`	Multiple filter sizes
`SpectralFilter.compute_subfilter_stress()`	LES stress tensor
`MultiScaleDecomposition.decompose()`	Wavelet-like decomposition
`prepare_training_data()`	Create network training pairs

Chapter 8

Neural Network Architectures

This chapter explains the neural network architectures designed for turbulence scale reconstruction.

8.1 Overview

8.1.1 The Task

Predict small-scale turbulent fluctuations from coarse-grained fields:

$$\mathbf{V}_{\text{coarse}} \xrightarrow{\text{Neural Network}} \delta\mathbf{V} \quad \text{(sub-filter contribution)} \tag{8.1}$$

Then reconstruct:

$$\mathbf{V}_{\text{fine}} \approx \mathbf{V}_{\text{coarse}} + \delta\mathbf{V} \tag{8.2}$$

8.1.2 Architecture Requirements

Requirement	Why	Solution
3D spatial data	Turbulence is 3D	3D convolutions
Multi-scale features	Energy cascade	U-Net / multi-scale
Physical constraints	$\nabla \cdot \mathbf{u} = 0$	Projection layers
Local + global context	Eddies span scales	Skip connections
Memory efficient	$256^3 = 16$M points	Progressive training

8.2 Module Structure

```
src/networks.py
+-- PositionalEncoding
+-- FourierFeatures
+-- ConvBlock3D
+-- ResBlock3D
+-- AttentionBlock3D
+-- ScaleReconstructionNet
+-- UNet3D
+-- FourierNeuralOperator3D
+-- MultiScaleNetwork
+-- Helper functions
```

8.3 Building Blocks

8.3.1 3D Convolutional Block

```python
class ConvBlock3D(nn.Module):
    """Basic 3D convolutional block with normalization and activation.
    """

    def __init__(self, in_channels, out_channels, kernel_size=3,
                 stride=1, padding=1, norm='batch', activation='relu'):
        super().__init__()

        # Convolution
        self.conv = nn.Conv3d(in_channels, out_channels,
                              kernel_size, stride, padding)

        # Normalization
        if norm == 'batch':
            self.norm = nn.BatchNorm3d(out_channels)
        elif norm == 'instance':
            self.norm = nn.InstanceNorm3d(out_channels)
        elif norm == 'group':
            self.norm = nn.GroupNorm(8, out_channels)
        else:
            self.norm = nn.Identity()

        # Activation
        if activation == 'relu':
            self.act = nn.ReLU(inplace=True)
        elif activation == 'gelu':
            self.act = nn.GELU()
        elif activation == 'silu':
            self.act = nn.SiLU()
        else:
            self.act = nn.Identity()

    def forward(self, x):
        return self.act(self.norm(self.conv(x)))
```

8.3.2 Residual Block

```python
class ResBlock3D(nn.Module):
    """3D Residual block with skip connection."""

    def __init__(self, channels, kernel_size=3):
        super().__init__()
        padding = kernel_size // 2

        self.block = nn.Sequential(
            nn.Conv3d(channels, channels, kernel_size, padding=padding),
            nn.BatchNorm3d(channels),
            nn.ReLU(inplace=True),
            nn.Conv3d(channels, channels, kernel_size, padding=padding),
            nn.BatchNorm3d(channels),
        )
        self.relu = nn.ReLU(inplace=True)
```

```
    def forward(self, x):
        identity = x
        out = self.block(x)
        out = out + identity  # Skip connection
        return self.relu(out)
```

8.3.3 Attention Block

```
class AttentionBlock3D(nn.Module):
    """Self-attention for capturing long-range dependencies."""

    def __init__(self, channels, num_heads=8):
        super().__init__()
        self.norm = nn.GroupNorm(8, channels)
        self.attention = nn.MultiheadAttention(channels, num_heads,
                                               batch_first=True)

    def forward(self, x):
        b, c, d, h, w = x.shape
        # Reshape to sequence: (B, C, D*H*W) -> (B, D*H*W, C)
        x_flat = x.view(b, c, -1).permute(0, 2, 1)
        x_norm = self.norm(x.view(b, c, -1)).permute(0, 2, 1)
        # Self-attention
        attn_out, _ = self.attention(x_norm, x_norm, x_norm)
        # Reshape back
        out = attn_out.permute(0, 2, 1).view(b, c, d, h, w)
        return x + out  # Residual
```

8.4 Positional Encoding

8.4.1 Fourier Features

Convolutions are translation-equivariant but turbulence can be position-dependent. Fourier features provide positional information:

```
class FourierFeatures(nn.Module):
    """Random Fourier features for positional encoding."""

    def __init__(self, input_dim=3, num_frequencies=64, sigma=10.0):
        super().__init__()
        self.num_frequencies = num_frequencies
        # Random frequencies (fixed, not learned)
        B = torch.randn(input_dim, num_frequencies) * sigma
        self.register_buffer('B', B)

    def forward(self, coords):
        """
        Args:
            coords: (..., input_dim) normalized coordinates in [-1, 1]
        Returns:
            features: (..., 2 * num_frequencies) Fourier features
        """
        proj = coords @ self.B
        return torch.cat([torch.sin(2*np.pi*proj),
                          torch.cos(2*np.pi*proj)], dim=-1)
```

8.5 U-Net 3D Architecture

8.5.1 Overview

U-Net is the workhorse architecture for dense prediction with encoder-decoder structure and skip connections:

```
Input (3 channels)
    | Encoder
    | Downsample (2x)
    | Encoder
    | Downsample (2x)
    ...
    | Bottleneck
    ...
    ^ Upsample (2x)
    ^ Decoder + Skip <-----|
    ^ Upsample (2x)
    ^ Decoder + Skip <-----|
    |
Output (3 channels)
```

8.5.2 Implementation

```python
class UNet3D(nn.Module):
    """3D U-Net for turbulence scale reconstruction."""

    def __init__(self, in_channels=3, out_channels=3,
                 features=[32, 64, 128, 256], use_attention=False):
        super().__init__()

        self.encoder = nn.ModuleList()
        self.decoder = nn.ModuleList()
        self.pool = nn.MaxPool3d(kernel_size=2, stride=2)

        # Encoder path
        for feature in features:
            self.encoder.append(self._double_conv(in_channels, feature))
            in_channels = feature

        # Bottleneck
        self.bottleneck = self._double_conv(features[-1], features[-1] *
    2)

        # Decoder path
        for feature in reversed(features):
            self.decoder.append(
                nn.ConvTranspose3d(feature * 2, feature, kernel_size=2,
   stride=2)
            )
            self.decoder.append(self._double_conv(feature * 2, feature))

        # Final output
        self.final_conv = nn.Conv3d(features[0], out_channels,
   kernel_size=1)

    def _double_conv(self, in_ch, out_ch):
```

```
        return nn.Sequential(
            ConvBlock3D(in_ch, out_ch),
            ConvBlock3D(out_ch, out_ch)
        )

    def forward(self, x):
        skip_connections = []

        # Encoder
        for encoder in self.encoder:
            x = encoder(x)
            skip_connections.append(x)
            x = self.pool(x)

        # Bottleneck
        x = self.bottleneck(x)

        # Decoder
        skip_connections = skip_connections[::-1]
        for idx in range(0, len(self.decoder), 2):
            x = self.decoder[idx](x)  # Upsample
            skip = skip_connections[idx // 2]
            x = torch.cat([skip, x], dim=1)  # Concatenate skip
            x = self.decoder[idx + 1](x)  # Conv

        return self.final_conv(x)
```

8.6 Scale Reconstruction Network

```
class ScaleReconstructionNet(nn.Module):
    """
    Neural network for reconstructing fine-scale turbulence from coarse.

    Input: V_coarse (B, 3, N, N, N)
    Output: delta_V = V_fine - V_coarse (B, 3, N, N, N)
    """

    def __init__(self, in_channels=3, out_channels=3,
                 base_features=32, num_levels=4,
                 use_residual=True, use_attention=False):
        super().__init__()
        self.use_residual = use_residual

        # Initial convolution
        self.input_conv = ConvBlock3D(in_channels, base_features)

        # Encoder with residual blocks
        self.encoder = nn.ModuleList()
        self.downsample = nn.ModuleList()
        in_ch = base_features

        for i in range(num_levels):
            out_ch = min(base_features * (2 ** i), 256)
            self.encoder.append(nn.Sequential(
                ResBlock3D(in_ch), ResBlock3D(in_ch)
            ))
```

```
            self.downsample.append(
                nn.Conv3d(in_ch, out_ch, 3, stride=2, padding=1)
            )
            in_ch = out_ch

        # Bottleneck
        self.bottleneck = nn.Sequential(
            ResBlock3D(in_ch),
            AttentionBlock3D(in_ch) if use_attention else nn.Identity(),
            ResBlock3D(in_ch)
        )

        # Output projection
        self.output_conv = nn.Sequential(
            ConvBlock3D(in_ch, base_features),
            nn.Conv3d(base_features, out_channels, 1)
        )
```

8.7 Fourier Neural Operator

8.7.1 Concept

Instead of convolutions, learn transformations in Fourier space:

$$\text{Input} \xrightarrow{\text{FFT}} \text{Spectral Conv (learned weights)} \xrightarrow{\text{iFFT}} \text{Output} \tag{8.3}$$

Advantage: Global receptive field in a single layer.

8.7.2 Spectral Convolution Layer

```
class SpectralConv3d(nn.Module):
    """Spectral convolution layer for FNO."""

    def __init__(self, in_channels, out_channels, modes):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.modes = modes  # Number of Fourier modes to keep

        # Complex-valued weights
        self.scale = 1 / (in_channels * out_channels)
        self.weights = nn.Parameter(
            self.scale * torch.randn(in_channels, out_channels,
                                     modes, modes, modes, dtype=torch.
   cfloat)
        )

    def forward(self, x):
        batchsize = x.shape[0]
        # FFT
        x_ft = torch.fft.rfftn(x, dim=(-3, -2, -1))
        # Multiply by learned weights (for low modes only)
        out_ft = torch.zeros(batchsize, self.out_channels,
                             x.size(-3), x.size(-2), x.size(-1)//2 + 1,
                             dtype=torch.cfloat, device=x.device)
```

```
        out_ft[:, :, :self.modes, :self.modes, :self.modes] = \
            torch.einsum("bixyz,ioxyz->boxyz",
                         x_ft[:, :, :self.modes, :self.modes, :self.modes
    ],
                         self.weights)
        # Inverse FFT
        x = torch.fft.irfftn(out_ft, s=(x.size(-3), x.size(-2), x.size
    (-1)))
        return x
```

8.8 Divergence-Free Output Layer

8.8.1 Physical Constraint

The predicted velocity must satisfy $\nabla \cdot \mathbf{u} = 0$ for incompressible flow.

8.8.2 Projection Approach

```
class DivergenceFreeProjection(nn.Module):
    """Project velocity field to divergence-free space."""

    def __init__(self, N, L):
        super().__init__()
        # Pre-compute wavenumbers
        k1d = torch.fft.fftfreq(N, d=L/(2*np.pi*N))
        kx, ky, kz = torch.meshgrid(k1d, k1d, k1d, indexing='ij')
        k_sq = kx**2 + ky**2 + kz**2
        k_sq[0, 0, 0] = 1.0  # Avoid division by zero

        self.register_buffer('kx', kx)
        self.register_buffer('ky', ky)
        self.register_buffer('kz', kz)
        self.register_buffer('k_sq', k_sq)

    def forward(self, velocity):
        # velocity: (B, 3, N, N, N)
        u_hat = torch.fft.fftn(velocity[:, 0], dim=(-3, -2, -1))
        v_hat = torch.fft.fftn(velocity[:, 1], dim=(-3, -2, -1))
        w_hat = torch.fft.fftn(velocity[:, 2], dim=(-3, -2, -1))

        # k . u_hat
        k_dot_u = self.kx * u_hat + self.ky * v_hat + self.kz * w_hat

        # Project: u_hat_sol = u_hat - k(k.u_hat)/|k|^2
        u_hat = u_hat - self.kx * k_dot_u / self.k_sq
        v_hat = v_hat - self.ky * k_dot_u / self.k_sq
        w_hat = w_hat - self.kz * k_dot_u / self.k_sq

        # Back to physical space
        u = torch.fft.ifftn(u_hat, dim=(-3, -2, -1)).real
        v = torch.fft.ifftn(v_hat, dim=(-3, -2, -1)).real
        w = torch.fft.ifftn(w_hat, dim=(-3, -2, -1)).real

        return torch.stack([u, v, w], dim=1)
```

8.9 Model Selection Guide

8.9.1 By Problem Size

Grid Size	Recommended Architecture	Parameters
64^3	UNet3D (4 levels)	∼2M
128^3	ScaleReconstructionNet	∼5M
256^3	FNO or Progressive UNet	∼10M

8.9.2 By Task

Task	Best Architecture	Why
Super-resolution	UNet3D	Multi-scale features
Scale prediction	ScaleReconstructionNet	Residual learning
Long-range patterns	FNO	Global receptive field
Physics-informed	+ DivFree projection	Hard constraint

8.10 Summary

8.10.1 Key Architectures

Network	Strength	Use Case
UNet3D	Multi-scale features	General purpose
ScaleReconstructionNet	Residual learning	Scale reconstruction
FNO	Global receptive field	Large domains

8.10.2 Key Layers

Layer	Purpose
ConvBlock3D	Basic feature extraction
ResBlock3D	Deep networks without degradation
AttentionBlock3D	Long-range dependencies
SpectralConv3d	Fourier-space learning
DivergenceFreeProjection	Physical constraint

Chapter 9

Physics-Informed Loss Functions

This chapter explains how we design loss functions that encode turbulence physics, ensuring the network learns physically meaningful representations.

9.1 Overview

9.1.1 Why Physics-Informed Losses?

Standard MSE loss:

$$L_{\text{MSE}} = \|\mathbf{V}_{\text{pred}} - \mathbf{V}_{\text{true}}\|^2 \tag{9.1}$$

Problems with standard losses:

- Treats all errors equally (but small scales matter more for energy!)
- Ignores physical constraints (divergence-free, energy spectrum)
- May learn non-physical artifacts

Solution: Add physics-based terms that penalize unphysical outputs.

9.1.2 Loss Function Structure

$$L_{\text{total}} = \lambda_1 L_{\text{data}} + \lambda_2 L_{\text{spectral}} + \lambda_3 L_{\text{divergence}} + \lambda_4 L_{\text{gradient}} + \cdots \tag{9.2}$$

Where λ_i are learnable or tuned weights.

9.2 Module Structure

```
src/losses.py
+-- MSELoss, L1Loss (wrappers)
+-- SpectralLoss
+-- DivergenceLoss
+-- GradientLoss
+-- StructureFunctionLoss
+-- EnergyConservationLoss
+-- VorticityLoss
+-- TurbulenceAwareLoss
+-- CombinedPhysicsLoss
+-- Loss weight schedulers
```

9.3 Data Fidelity Losses

9.3.1 Mean Squared Error

```python
class MSELoss(nn.Module):
    """Standard MSE with optional weighting."""

    def __init__(self, reduction='mean', weight=None):
        super().__init__()
        self.reduction = reduction
        self.weight = weight

    def forward(self, pred, target):
        diff = (pred - target) ** 2

        if self.weight is not None:
            diff = diff * self.weight

        if self.reduction == 'mean':
            return diff.mean()
        elif self.reduction == 'sum':
            return diff.sum()
        else:
            return diff
```

9.3.2 Relative Error

```python
class RelativeL2Loss(nn.Module):
    """L2 error normalized by target magnitude."""

    def forward(self, pred, target):
        diff = pred - target
        norm_target = torch.sqrt((target ** 2).sum(dim=-1, keepdim=True)
    + 1e-8)
        return (diff / norm_target).pow(2).mean()
```

9.4 Spectral Loss

9.4.1 Motivation

The energy spectrum $E(k)$ is the key diagnostic of turbulence. We want:

$$E_{\text{pred}}(k) \approx E_{\text{true}}(k) \quad \text{for all wavenumbers } k \tag{9.3}$$

9.4.2 Implementation

```python
class SpectralLoss(nn.Module):
    """
    Loss based on matching energy spectrum E(k).
    Penalizes deviations in spectral energy distribution.
    """

    def __init__(self, N, L=2*np.pi, num_bins=None, log_scale=True,
                 weight_low_k=1.0, weight_high_k=1.0):
```

```
        super().__init__()
        self.N = N
        self.L = L
        self.num_bins = num_bins or N // 2
        self.log_scale = log_scale
        self.weight_low_k = weight_low_k
        self.weight_high_k = weight_high_k
        self._setup_wavenumbers()

    def compute_spectrum(self, velocity):
        """Compute shell-averaged energy spectrum."""
        u_hat = torch.fft.fftn(velocity[:, 0], dim=(-3, -2, -1))
        v_hat = torch.fft.fftn(velocity[:, 1], dim=(-3, -2, -1))
        w_hat = torch.fft.fftn(velocity[:, 2], dim=(-3, -2, -1))

        # Energy density
        energy = 0.5 * (u_hat.abs()**2 + v_hat.abs()**2 + w_hat.abs()
    **2)

        # Shell averaging
        E_k = torch.zeros(velocity.shape[0], self.num_bins, device=
    velocity.device)
        dk = self.k_bins[1] - self.k_bins[0]

        for i in range(self.num_bins):
            mask = (self.k_mag >= self.k_bins[i]) & (self.k_mag < self.
    k_bins[i+1])
            E_k[:, i] = energy[:, mask].sum(dim=-1) / self.N**6 / dk

        return E_k

    def forward(self, pred, target):
        E_pred = self.compute_spectrum(pred)
        E_true = self.compute_spectrum(target)

        if self.log_scale:
            E_pred = torch.log(E_pred + 1e-10)
            E_true = torch.log(E_true + 1e-10)

        # Weighted error
        weights = torch.ones(self.num_bins, device=pred.device)
        weights[:self.num_bins//3] *= self.weight_low_k
        weights[2*self.num_bins//3:] *= self.weight_high_k

        loss = ((E_pred - E_true) ** 2 * weights).mean()
        return loss
```

9.5 Divergence Loss

9.5.1 Physical Constraint

Incompressible flow requires:

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{9.4}$$

9.5.2 Implementation

```
class DivergenceLoss(nn.Module):
    """Penalize non-zero divergence (compressibility)."""

    def __init__(self, dx=None, method='spectral'):
        super().__init__()
        self.dx = dx
        self.method = method

    def _spectral_divergence(self, velocity):
        """Compute divergence in Fourier space (exact)."""
        N = velocity.shape[-1]
        L = 2 * np.pi if self.dx is None else self.dx * N

        # Wavenumbers
        k1d = torch.fft.fftfreq(N, d=L/(2*np.pi*N), device=velocity.
    device)
        kx, ky, kz = torch.meshgrid(k1d, k1d, k1d, indexing='ij')

        # FFT of velocity components
        u_hat = torch.fft.fftn(velocity[:, 0], dim=(-3, -2, -1))
        v_hat = torch.fft.fftn(velocity[:, 1], dim=(-3, -2, -1))
        w_hat = torch.fft.fftn(velocity[:, 2], dim=(-3, -2, -1))

        # Divergence in Fourier space
        div_hat = 1j * (kx * u_hat + ky * v_hat + kz * w_hat)
        div = torch.fft.ifftn(div_hat, dim=(-3, -2, -1)).real

        return (div ** 2).mean()

    def forward(self, velocity):
        if self.method == 'spectral':
            return self._spectral_divergence(velocity)
        else:
            return self._finite_diff_divergence(velocity)
```

9.6 Gradient Loss

9.6.1 Motivation

Small-scale turbulence is characterized by **velocity gradients**. Matching gradients ensures small-scale fidelity.

9.6.2 Implementation

```
class GradientLoss(nn.Module):
    """Loss on velocity gradients. Important for small-scale features.
    """

    def __init__(self, order=1, p=2):
        super().__init__()
        self.order = order
        self.p = p

    def _first_order_loss(self, pred, target):
```

```
        """Loss on first-order gradients."""
        loss = 0.0
        for dim in range(3):  # x, y, z derivatives
            for comp in range(3):  # u, v, w
                grad_pred = torch.gradient(pred[:, comp], dim=dim+1)[0]
                grad_true = torch.gradient(target[:, comp], dim=dim+1)
    [0]
                loss += torch.abs(grad_pred - grad_true).pow(self.p).
    mean()
        return loss / 9
```

9.7 Structure Function Loss

9.7.1 Physics

Structure functions measure velocity differences across scales:

$$S_n(r) = \langle |u(x+r) - u(x)|^n \rangle \tag{9.5}$$

Kolmogorov theory predicts $S_2(r) \sim r^{2/3}$ in the inertial range.

9.7.2 Implementation

```
class StructureFunctionLoss(nn.Module):
    """Loss based on structure functions."""

    def __init__(self, orders=[2], separations=[1, 2, 4, 8, 16]):
        super().__init__()
        self.orders = orders
        self.separations = separations

    def compute_structure_function(self, velocity, r, order=2):
        """Compute longitudinal structure function at separation r."""
        delta_u = torch.roll(velocity[:, 0], -r, dims=-3) - velocity[:,
    0]
        S = (torch.abs(delta_u) ** order).mean(dim=(-3, -2, -1))
        return S

    def forward(self, pred, target):
        loss = 0.0
        for order in self.orders:
            for r in self.separations:
                S_pred = self.compute_structure_function(pred, r, order)
                S_true = self.compute_structure_function(target, r,
    order)
                loss += ((S_pred - S_true) / (S_true + 1e-8)).pow(2).
    mean()
        return loss / (len(self.orders) * len(self.separations))
```

9.8 Energy Conservation Loss

9.8.1 Physical Constraint

Total kinetic energy should be conserved (or match target):

$$E = \frac{1}{2}\langle |\mathbf{u}|^2 \rangle \tag{9.6}$$

```python
class EnergyConservationLoss(nn.Module):
    """Penalize deviation in total kinetic energy."""

    def __init__(self, relative=True):
        super().__init__()
        self.relative = relative

    def forward(self, pred, target):
        E_pred = 0.5 * (pred ** 2).sum(dim=1).mean()
        E_true = 0.5 * (target ** 2).sum(dim=1).mean()

        if self.relative:
            return ((E_pred - E_true) / (E_true + 1e-8)) ** 2
        else:
            return (E_pred - E_true) ** 2
```

9.9 Vorticity Loss

Vorticity $\omega = \nabla \times \mathbf{u}$ is central to turbulence dynamics:

```python
class VorticityLoss(nn.Module):
    """Loss on vorticity field."""

    def forward(self, pred, target):
        omega_pred = self._compute_vorticity(pred)
        omega_true = self._compute_vorticity(target)
        return ((omega_pred - omega_true) ** 2).mean()

    def _compute_vorticity(self, velocity):
        u, v, w = velocity[:, 0], velocity[:, 1], velocity[:, 2]
        omega_x = torch.gradient(w, dim=-2)[0] - torch.gradient(v, dim
    =-1)[0]
        omega_y = torch.gradient(u, dim=-1)[0] - torch.gradient(w, dim
    =-3)[0]
        omega_z = torch.gradient(v, dim=-3)[0] - torch.gradient(u, dim
    =-2)[0]
        return torch.stack([omega_x, omega_y, omega_z], dim=1)
```

9.10 Combined Physics Loss

```python
class CombinedPhysicsLoss(nn.Module):
    """Combined loss with multiple physics-informed terms."""

    def __init__(self, N, L=2*np.pi,
                 lambda_data=1.0, lambda_spectral=0.1,
                 lambda_divergence=0.01, lambda_gradient=0.1,
                 lambda_energy=0.01):
        super().__init__()

        self.mse_loss = MSELoss()
        self.spectral_loss = SpectralLoss(N, L)
        self.divergence_loss = DivergenceLoss()
        self.gradient_loss = GradientLoss()
        self.energy_loss = EnergyConservationLoss()
```

```
        self.lambda_data = lambda_data
        self.lambda_spectral = lambda_spectral
        self.lambda_divergence = lambda_divergence
        self.lambda_gradient = lambda_gradient
        self.lambda_energy = lambda_energy

    def forward(self, pred, target, return_components=False):
        L_data = self.mse_loss(pred, target)
        L_spec = self.spectral_loss(pred, target)
        L_div = self.divergence_loss(pred)
        L_grad = self.gradient_loss(pred, target)
        L_energy = self.energy_loss(pred, target)

        total = (self.lambda_data * L_data +
                 self.lambda_spectral * L_spec +
                 self.lambda_divergence * L_div +
                 self.lambda_gradient * L_grad +
                 self.lambda_energy * L_energy)

        if return_components:
            return total, {
                'data': L_data.item(), 'spectral': L_spec.item(),
                'divergence': L_div.item(), 'gradient': L_grad.item(),
                'energy': L_energy.item()
            }
        return total
```

9.11 Loss Weight Scheduling

9.11.1 Dynamic Weighting

```
class LossWeightScheduler:
    """Adjust loss weights during training."""

    def __init__(self, initial_weights, final_weights, warmup_epochs=10)
    :
        self.initial = initial_weights
        self.final = final_weights
        self.warmup = warmup_epochs

    def get_weights(self, epoch):
        if epoch < self.warmup:
            alpha = epoch / self.warmup
        else:
            alpha = 1.0

        return {k: self.initial[k] + alpha * (self.final[k] - self.
    initial[k])
                for k in self.initial}
```

9.12 Best Practices

9.12.1 Weight Selection Guidelines

Loss Term	Typical λ	Notes
Data (MSE)	1.0	Baseline
Spectral	0.01–0.1	Scale to match MSE magnitude
Divergence	0.001–0.01	Soft constraint
Gradient	0.01–0.1	Important for small scales
Energy	0.01	Prevents energy drift

9.12.2 Common Mistakes

1. **Divergence too high**: Network outputs zero (trivially divergence-free)
2. **Spectral too high**: Overfits to spectrum, ignores spatial structure
3. **Not normalizing**: Losses at different scales dominate unpredictably

9.13 Summary

9.13.1 Loss Functions

Loss	Purpose	When to Use
MSE	Basic fidelity	Always
Spectral	Match $E(k)$	Turbulence
Divergence	$\nabla \cdot \mathbf{u} = 0$	Incompressible
Gradient	Small scales	High-frequency content
Structure Function	Scaling laws	Physics validation
Vorticity	Rotation	Vortex-dominated flows

9.13.2 Key Equations

$$L_{\text{total}} = \lambda_{\text{data}} \|\mathbf{V}_{\text{pred}} - \mathbf{V}_{\text{true}}\|^2 + \lambda_{\text{spec}} \|E_{\text{pred}}(k) - E_{\text{true}}(k)\|^2 + \lambda_{\text{div}} \|\nabla \cdot \mathbf{V}_{\text{pred}}\|^2 + \lambda_{\text{grad}} \|\nabla \mathbf{V}_{\text{pred}} - \nabla \mathbf{V}_{\text{true}}\|^2 \tag{9.7}$$

Chapter 10

Training Pipeline

This chapter covers the complete training workflow for turbulence scale reconstruction models.

10.1 Overview

10.1.1 Training Workflow

```
1. Data Generation
   -> TurbulenceParams -> generate_isotropic_turbulence

2. Filtering
   -> SpectralFilter -> V_coarse, V_fine, delta_V

3. Dataset Creation
   -> TurbulenceDataset -> DataLoader

4. Model & Optimizer Setup
   -> ScaleReconstructionNet + CombinedPhysicsLoss

5. Training Loop
   -> Forward -> Loss -> Backward -> Update

6. Validation & Checkpointing
   -> Metrics -> Save best model
```

10.2 Module Structure

```
src/training.py
+-- TurbulenceDataset
+-- create_dataloaders()
+-- Trainer (main class)
|   +-- __init__()
|   +-- train_epoch()
|   +-- validate()
|   +-- fit()
|   +-- save/load_checkpoint()
+-- compute_metrics()
+-- EarlyStopping
+-- LearningRateScheduler
```

10.3 Dataset Class

10.3.1 TurbulenceDataset

```
class TurbulenceDataset(torch.utils.data.Dataset):
    """Dataset for turbulence scale reconstruction."""

    def __init__(self, num_samples, params, filter_params,
                 precompute=True, cache_dir=None):
        """
        Args:
            num_samples: Number of velocity field realizations
            params: TurbulenceParams for generation
            filter_params: Dict with R1, R2, filter_type
            precompute: If True, generate all data upfront
            cache_dir: If set, cache to disk
        """
        self.num_samples = num_samples
        self.params = params
        self.filter_params = filter_params

        self.spectral_filter = SpectralFilter(
            params.N, params.L, params.dims
        )

        if precompute:
            self._precompute_data()

    def _precompute_data(self):
        """Generate and filter all velocity fields."""
        self.data = []
        for seed in tqdm(range(self.num_samples), desc="Generating data"
    ):
            self.params.seed = seed
            velocity = generate_isotropic_turbulence_spectral(self.
    params)

            V_coarse = self.spectral_filter.filter_field(
                velocity, self.filter_params['R1'], self.filter_params['
    filter_type']
            )
            V_fine = self.spectral_filter.filter_field(
                velocity, self.filter_params['R2'], self.filter_params['
    filter_type']
            )

            self.data.append({
                'V_coarse': torch.from_numpy(V_coarse).float(),
                'V_fine': torch.from_numpy(V_fine).float(),
                'delta_V': torch.from_numpy(V_fine - V_coarse).float(),
                'seed': seed
            })

    def __getitem__(self, idx):
        sample = self.data[idx]
        # Transpose to (C, D, H, W) format for PyTorch
        V_coarse = sample['V_coarse'].permute(3, 0, 1, 2)
        V_fine = sample['V_fine'].permute(3, 0, 1, 2)
```

```
        delta_V = sample['delta_V'].permute(3, 0, 1, 2)

        return {'input': V_coarse, 'target': delta_V,
                'V_fine': V_fine, 'seed': sample['seed']}
```

10.3.2 Creating DataLoaders

```
def create_dataloaders(train_dataset, val_dataset, batch_size,
                       num_workers=4, pin_memory=True):
    """Create training and validation DataLoaders."""
    train_loader = DataLoader(
        train_dataset, batch_size=batch_size, shuffle=True,
        num_workers=num_workers, pin_memory=pin_memory, drop_last=True
    )
    val_loader = DataLoader(
        val_dataset, batch_size=batch_size, shuffle=False,
        num_workers=num_workers, pin_memory=pin_memory
    )
    return train_loader, val_loader
```

10.4 Trainer Class

10.4.1 Initialization

```
class Trainer:
    """Training manager for scale reconstruction models."""

    def __init__(self, model, train_loader, val_loader,
                 optimizer, scheduler, loss_fn,
                 device='cuda', config=None):
        self.model = model.to(device)
        self.train_loader = train_loader
        self.val_loader = val_loader
        self.optimizer = optimizer
        self.scheduler = scheduler
        self.loss_fn = loss_fn
        self.device = device
        self.config = config or {}

        # Tracking
        self.current_epoch = 0
        self.best_val_loss = float('inf')
        self.train_losses = []
        self.val_losses = []
        self.metrics_history = []

        # Gradient scaling for mixed precision
        self.scaler = torch.cuda.amp.GradScaler()
```

10.4.2 Training Epoch

```
def train_epoch(self):
    """Train for one epoch."""
```

```
    self.model.train()
    epoch_loss = 0.0
    num_batches = len(self.train_loader)

    pbar = tqdm(self.train_loader, desc=f"Epoch {self.current_epoch}")

    for batch_idx, batch in enumerate(pbar):
        input_coarse = batch['input'].to(self.device)
        target = batch['target'].to(self.device)

        self.optimizer.zero_grad()

        # Mixed precision forward pass
        with torch.cuda.amp.autocast():
            pred = self.model(input_coarse)
            loss = self.loss_fn(pred, target)

        # Backward pass with gradient scaling
        self.scaler.scale(loss).backward()

        # Gradient clipping
        if self.config.get('grad_clip', 0) > 0:
            self.scaler.unscale_(self.optimizer)
            torch.nn.utils.clip_grad_norm_(
                self.model.parameters(), self.config['grad_clip']
            )

        self.scaler.step(self.optimizer)
        self.scaler.update()

        epoch_loss += loss.item()
        pbar.set_postfix({'loss': loss.item()})

    return epoch_loss / num_batches
```

10.4.3 Validation

```
def validate(self):
    """Evaluate on validation set."""
    self.model.eval()
    val_loss = 0.0
    all_metrics = []

    with torch.no_grad():
        for batch in self.val_loader:
            input_coarse = batch['input'].to(self.device)
            target = batch['target'].to(self.device)
            V_fine = batch['V_fine'].to(self.device)

            with torch.cuda.amp.autocast():
                pred = self.model(input_coarse)
                loss = self.loss_fn(pred, target)

            val_loss += loss.item()

            # Compute detailed metrics
            V_reconstructed = input_coarse + pred
```

```
            metrics = compute_metrics(V_reconstructed, V_fine,
    input_coarse)
            all_metrics.append(metrics)

    avg_metrics = {k: np.mean([m[k] for m in all_metrics])
                   for k in all_metrics[0].keys()}

    return val_loss / len(self.val_loader), avg_metrics
```

10.4.4 Main Training Loop

```
def fit(self, num_epochs, checkpoint_dir=None, early_stopping_patience=
    None):
    """Full training loop."""
    early_stopping = EarlyStopping(patience=early_stopping_patience)

    for epoch in range(num_epochs):
        self.current_epoch = epoch

        train_loss = self.train_epoch()
        self.train_losses.append(train_loss)

        val_loss, metrics = self.validate()
        self.val_losses.append(val_loss)
        self.metrics_history.append(metrics)

        if self.scheduler is not None:
            self.scheduler.step(val_loss)

        print(f"Epoch {epoch}: train_loss={train_loss:.4f}, "
              f"val_loss={val_loss:.4f}")

        # Checkpointing
        if checkpoint_dir and val_loss < self.best_val_loss:
            self.best_val_loss = val_loss
            self.save_checkpoint(os.path.join(checkpoint_dir, '
    best_model.pt'))

        if early_stopping(val_loss):
            print(f"Early stopping at epoch {epoch}")
            break

    return self.train_losses, self.val_losses, self.metrics_history
```

10.5 Metrics Computation

```
def compute_metrics(V_pred, V_true, V_coarse):
    """Compute evaluation metrics for turbulence reconstruction."""
    # Basic error metrics
    mse = ((V_pred - V_true) ** 2).mean().item()
    rmse = np.sqrt(mse)

    # Relative error
    rel_error = (torch.norm(V_pred - V_true) / torch.norm(V_true)).item
    ()
```

```
    # Improvement over input
    mse_input = ((V_coarse - V_true) ** 2).mean().item()
    improvement = 1 - mse / mse_input

    # Correlation
    V_pred_flat = V_pred.view(-1)
    V_true_flat = V_true.view(-1)
    correlation = torch.corrcoef(
        torch.stack([V_pred_flat, V_true_flat])
    )[0, 1].item()

    # Energy error
    E_pred = 0.5 * (V_pred ** 2).sum(dim=1).mean().item()
    E_true = 0.5 * (V_true ** 2).sum(dim=1).mean().item()
    energy_error = abs(E_pred - E_true) / E_true

    return {
        'mse': mse, 'rmse': rmse, 'rel_error': rel_error,
        'improvement': improvement, 'correlation': correlation,
        'energy_error': energy_error
    }
```

10.6 Checkpointing

```
def save_checkpoint(self, filepath):
    """Save training checkpoint."""
    checkpoint = {
        'epoch': self.current_epoch,
        'model_state_dict': self.model.state_dict(),
        'optimizer_state_dict': self.optimizer.state_dict(),
        'scheduler_state_dict': (self.scheduler.state_dict()
                                 if self.scheduler else None),
        'best_val_loss': self.best_val_loss,
        'train_losses': self.train_losses,
        'val_losses': self.val_losses,
        'config': self.config
    }
    torch.save(checkpoint, filepath)

def load_checkpoint(self, filepath):
    """Load training checkpoint."""
    checkpoint = torch.load(filepath, map_location=self.device)
    self.model.load_state_dict(checkpoint['model_state_dict'])
    self.optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    if self.scheduler and checkpoint['scheduler_state_dict']:
        self.scheduler.load_state_dict(checkpoint['scheduler_state_dict'
    ])
    self.current_epoch = checkpoint['epoch']
    self.best_val_loss = checkpoint['best_val_loss']
```

10.7 Learning Rate Scheduling

```
# ReduceLROnPlateau - reduces when validation plateaus
```

```
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5, verbose=True
)

# Cosine annealing
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
    optimizer, T_max=num_epochs, eta_min=1e-6
)

# OneCycleLR - fast convergence
scheduler = torch.optim.lr_scheduler.OneCycleLR(
    optimizer, max_lr=0.01, epochs=num_epochs,
    steps_per_epoch=len(train_loader)
)
```

10.8 Early Stopping

```
class EarlyStopping:
    """Stop training when validation loss stops improving."""

    def __init__(self, patience=10, min_delta=1e-4):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = float('inf')

    def __call__(self, val_loss):
        if val_loss < self.best_loss - self.min_delta:
            self.best_loss = val_loss
            self.counter = 0
            return False
        else:
            self.counter += 1
            return self.counter >= self.patience
```

10.9 Complete Training Script

```
# training_script.py
config = {
    'N': 64, 'Re_lambda': 100,
    'R1': 0.5, 'R2': 0.25,
    'batch_size': 4, 'num_epochs': 100,
    'lr': 1e-3, 'grad_clip': 1.0,
    'num_train_samples': 100, 'num_val_samples': 20,
}

device = 'cuda' if torch.cuda.is_available() else 'cpu'
params = TurbulenceParams(N=config['N'], Re_lambda=config['Re_lambda'])
filter_params = {'R1': config['R1'], 'R2': config['R2'],
                 'filter_type': FilterType.GAUSSIAN}

# Datasets
train_dataset = TurbulenceDataset(config['num_train_samples'], params,
    filter_params)
```

```
val_dataset = TurbulenceDataset(config['num_val_samples'], params,
    filter_params)
train_loader, val_loader = create_dataloaders(
    train_dataset, val_dataset, config['batch_size']
)

# Model
model = ScaleReconstructionNet(in_channels=3, out_channels=3,
                               base_features=32, num_levels=3)

# Optimizer & Scheduler
optimizer = torch.optim.AdamW(model.parameters(), lr=config['lr'])
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer,
    patience=5)

# Loss
loss_fn = CombinedPhysicsLoss(N=config['N'], lambda_data=1.0,
                              lambda_spectral=0.1, lambda_divergence
    =0.01)

# Train
trainer = Trainer(model, train_loader, val_loader,
                  optimizer, scheduler, loss_fn, device=device, config=
    config)
trainer.fit(num_epochs=config['num_epochs'], checkpoint_dir='./
    checkpoints',
            early_stopping_patience=15)
```

10.10 Summary

10.10.1 Key Components

Component	Purpose
TurbulenceDataset	Generate/load training pairs
Trainer	Manage training loop
compute_metrics	Evaluate performance
EarlyStopping	Prevent overfitting

10.10.2 Training Checklist

- ☐ Data generation parameters set correctly
- ☐ Filter sizes (R_1, R_2) appropriate for task
- ☐ Loss weights balanced
- ☐ Learning rate reasonable (10^{-4} to 10^{-3})
- ☐ Gradient clipping enabled
- ☐ Checkpointing configured
- ☐ Validation metrics tracked

Chapter 11

Analysis and Diagnostics

This chapter covers the tools for analyzing turbulent velocity fields and validating model predictions.

11.1 Overview

11.1.1 Purpose

The analysis module provides tools to:

- Compute energy spectra $E(k)$
- Calculate structure functions
- Measure turbulence statistics
- Validate physical constraints
- Compare predictions with ground truth

11.1.2 Key Diagnostics

Diagnostic	What it Measures	Why it Matters
Energy spectrum $E(k)$	Energy across scales	Validates $k^{-5/3}$ scaling
Structure functions $S_n(r)$	Velocity increments	Tests Kolmogorov theory
Divergence	$\nabla \cdot \mathbf{u}$	Verifies incompressibility
PDF/moments	Velocity distribution	Checks Gaussianity
Correlation	Spatial structure	Validates coherence

11.2 Energy Spectrum Analysis

11.2.1 3D Energy Spectrum

```
def compute_energy_spectrum_3d(velocity: np.ndarray, L: float = 2*np.pi)
    :
    """
    Compute shell-averaged energy spectrum E(k).
    E(k) is defined such that integral E(k)dk = 0.5<u^2> = TKE
    """
    N = velocity.shape[0]
    dx = L / N
```

```
    # FFT of velocity components
    u_hat = np.fft.fftn(velocity[..., 0])
    v_hat = np.fft.fftn(velocity[..., 1])
    w_hat = np.fft.fftn(velocity[..., 2])

    # Spectral energy density
    energy_hat = 0.5 * (np.abs(u_hat)**2 + np.abs(v_hat)**2 + np.abs(
   w_hat)**2)

    # Wavenumber grid
    k1d = np.fft.fftfreq(N, d=dx) * 2 * np.pi
    kx, ky, kz = np.meshgrid(k1d, k1d, k1d, indexing='ij')
    k_mag = np.sqrt(kx**2 + ky**2 + kz**2)

    # Shell averaging
    k_max = np.sqrt(3) * np.abs(k1d).max()
    num_bins = N // 2
    k_bins = np.linspace(0, k_max, num_bins + 1)
    k_centers = 0.5 * (k_bins[:-1] + k_bins[1:])
    E_k = np.zeros(num_bins)
    dk = k_bins[1] - k_bins[0]

    for i in range(num_bins):
        mask = (k_mag >= k_bins[i]) & (k_mag < k_bins[i + 1])
        if np.sum(mask) > 0:
            E_k[i] = np.sum(energy_hat[mask]) / N**6 / dk

    return k_centers, E_k
```

11.2.2 Spectrum Fitting

```
def fit_spectrum_slope(k, E_k, k_min=None, k_max=None):
    """Fit power law E(k) ~ k^alpha in specified range."""
    valid = (k > 0) & (E_k > 0)
    if k_min is not None:
        valid &= (k >= k_min)
    if k_max is not None:
        valid &= (k <= k_max)

    if np.sum(valid) < 3:
        return np.nan, np.nan, np.nan

    log_k = np.log(k[valid])
    log_E = np.log(E_k[valid])

    # Linear fit in log-log space
    slope, intercept = np.polyfit(log_k, log_E, 1)

    # R^2 calculation
    log_E_fit = slope * log_k + intercept
    ss_res = np.sum((log_E - log_E_fit)**2)
    ss_tot = np.sum((log_E - np.mean(log_E))**2)
    r_squared = 1 - ss_res / ss_tot if ss_tot > 0 else 0

    return slope, intercept, r_squared
```

11.3 Structure Functions

11.3.1 Longitudinal Structure Function

The structure function measures velocity differences across scales:

$$S_n(r) = \langle [u(x+r) - u(x)]^n \rangle \tag{11.1}$$

```
def compute_structure_function(velocity, orders=[2, 3],
                               separations=None, direction='x'):
    """Compute longitudinal structure functions S_n(r)."""
    N = velocity.shape[0]

    if separations is None:
        separations = [1, 2, 4, 8, 16, 32]
        separations = [s for s in separations if s < N//2]

    if direction == 'x':
        u = velocity[..., 0]
        axis = 0
    elif direction == 'y':
        u = velocity[..., 1]
        axis = 1
    else:
        u = velocity[..., 2]
        axis = 2

    results = {n: np.zeros(len(separations)) for n in orders}

    for i, r in enumerate(separations):
        delta_u = np.roll(u, -r, axis=axis) - u
        for n in orders:
            results[n][i] = np.mean(delta_u ** n)

    return results, np.array(separations)
```

11.3.2 Kolmogorov's 4/5 Law

Verify the exact relation $S_3(r) = -\frac{4}{5}\epsilon r$:

```
def verify_45_law(velocity, L, epsilon=None):
    """Verify Kolmogorov's 4/5 law: S_3(r) = -4/5 * epsilon * r"""
    S_n, r_pts = compute_structure_function(velocity, orders=[3])
    S3 = S_n[3]

    dx = L / velocity.shape[0]
    r_physical = r_pts * dx

    if epsilon is None:
        epsilon = estimate_dissipation(velocity, L)

    theoretical = -4/5 * epsilon * r_physical
    deviation = np.abs(S3 - theoretical) / np.abs(theoretical + 1e-10)

    return r_physical, S3, theoretical, deviation
```

11.4 Dissipation Estimation

11.4.1 From Velocity Gradients

$$\epsilon = \nu \langle (\partial u_i / \partial x_j)^2 \rangle = \nu \langle |\nabla \mathbf{u}|^2 \rangle \tag{11.2}$$

```
def estimate_dissipation(velocity, L):
    """Estimate turbulent dissipation rate from velocity gradients."""
    N = velocity.shape[0]
    dx = L / N

    grad_sum = 0.0
    for i in range(3):  # Velocity components
        for j in range(3):  # Spatial directions
            grad = np.gradient(velocity[..., i], dx, axis=j)
            grad_sum += np.mean(grad**2)

    return grad_sum  # Multiply by nu for actual dissipation
```

11.4.2 From Spectrum

$$\epsilon = 2\nu \int k^2 E(k)\, dk \tag{11.3}$$

```
def estimate_dissipation_spectral(k, E_k):
    """Estimate dissipation from energy spectrum."""
    dk = k[1] - k[0] if len(k) > 1 else 1.0
    dissipation_spectrum = 2 * k**2 * E_k
    return np.sum(dissipation_spectrum * dk)
```

11.5 TurbulenceAnalyzer Class

```
class TurbulenceAnalyzer:
    """Complete turbulence analysis toolkit."""

    def __init__(self, N, L=2*np.pi, nu=None):
        self.N = N
        self.L = L
        self.dx = L / N
        self.nu = nu

    def analyze(self, velocity, compute_all=True):
        """Perform comprehensive turbulence analysis."""
        results = {}

        # Basic statistics
        results['u_rms'] = np.sqrt(np.mean(velocity**2))
        results['u_mean'] = np.mean(velocity, axis=(0,1,2))
        results['TKE'] = 0.5 * np.mean(np.sum(velocity**2, axis=-1))

        # Energy spectrum
        k, E_k = compute_energy_spectrum_3d(velocity, self.L)
        results['k'] = k
        results['E_k'] = E_k
```

```
        # Spectrum slope
        k_min = k.max() / 10
        k_max = k.max() / 2
        slope, _, r2 = fit_spectrum_slope(k, E_k, k_min, k_max)
        results['inertial_slope'] = slope
        results['slope_r_squared'] = r2

        if compute_all:
            # Structure functions
            S_n, r = compute_structure_function(velocity, [2, 3, 4])
            results['structure_functions'] = S_n
            results['separations'] = r * self.dx

            # Divergence
            div = self.compute_divergence(velocity)
            results['max_divergence'] = np.max(np.abs(div))

            # Vorticity and enstrophy
            omega = self.compute_vorticity(velocity)
            results['enstrophy'] = 0.5 * np.mean(np.sum(omega**2, axis
    =-1))

            # Higher-order statistics
            results['skewness'] = self.compute_skewness(velocity)
            results['flatness'] = self.compute_flatness(velocity)

        return results

    def compute_divergence(self, velocity):
        """Compute divergence."""
        return (np.gradient(velocity[..., 0], self.dx, axis=0) +
                np.gradient(velocity[..., 1], self.dx, axis=1) +
                np.gradient(velocity[..., 2], self.dx, axis=2))

    def compute_vorticity(self, velocity):
        """Compute vorticity."""
        u, v, w = velocity[..., 0], velocity[..., 1], velocity[..., 2]
        omega_x = np.gradient(w, self.dx, axis=1) - np.gradient(v, self.
    dx, axis=2)
        omega_y = np.gradient(u, self.dx, axis=2) - np.gradient(w, self.
    dx, axis=0)
        omega_z = np.gradient(v, self.dx, axis=0) - np.gradient(u, self.
    dx, axis=1)
        return np.stack([omega_x, omega_y, omega_z], axis=-1)

    def compute_skewness(self, velocity):
        """Velocity derivative skewness."""
        du_dx = np.gradient(velocity[..., 0], self.dx, axis=0)
        return np.mean(du_dx**3) / np.mean(du_dx**2)**1.5

    def compute_flatness(self, velocity):
        """Velocity derivative flatness (kurtosis)."""
        du_dx = np.gradient(velocity[..., 0], self.dx, axis=0)
        return np.mean(du_dx**4) / np.mean(du_dx**2)**2
```

11.6 Reconstruction Quality Metrics

```
def evaluate_reconstruction(V_pred, V_true, V_coarse, L=2*np.pi):
    """Evaluate quality of scale reconstruction."""
    N = V_true.shape[0]

    # Point-wise errors
    mse = np.mean((V_pred - V_true)**2)
    mse_input = np.mean((V_coarse - V_true)**2)

    metrics = {
        'MSE': mse,
        'RMSE': np.sqrt(mse),
        'relative_error': np.sqrt(mse) / np.sqrt(np.mean(V_true**2)),
        'improvement_factor': mse_input / mse,
        'skill_score': 1 - mse / mse_input,
    }

    # Spectral comparison
    k, E_true = compute_energy_spectrum_3d(V_true, L)
    _, E_pred = compute_energy_spectrum_3d(V_pred, L)
    _, E_coarse = compute_energy_spectrum_3d(V_coarse, L)

    # Weighted mean absolute percentage error
    valid = E_true > E_true.max() * 1e-6
    weights = E_true[valid] / E_true[valid].sum()
    metrics['spectrum_WMAPE'] = np.sum(
        weights * np.abs(E_pred[valid] - E_true[valid]) / E_true[valid]
    )

    return metrics
```

11.7 Visualization Helpers

```
def plot_energy_spectrum(k, E_k, params=None, ax=None, **kwargs):
    """Plot energy spectrum with reference lines."""
    import matplotlib.pyplot as plt

    if ax is None:
        fig, ax = plt.subplots(figsize=(8, 6))

    ax.loglog(k[k > 0], E_k[k > 0], **kwargs)

    # K41 reference line
    k_ref = k[(k > k.max()/10) & (k < k.max()/2)]
    if len(k_ref) > 0:
        E_ref = E_k[k > 0][0] * (k_ref / k[k > 0][0])**(-5/3)
        ax.loglog(k_ref, E_ref, 'k--', alpha=0.5, label=r'$k^{-5/3}$')

    ax.set_xlabel(r'Wavenumber $k$')
    ax.set_ylabel(r'Energy spectrum $E(k)$')
    ax.legend()
    ax.grid(True, alpha=0.3)

    return ax
```

11.8 Summary Report

```python
def generate_analysis_report(velocity, params, save_path=None):
    """Generate comprehensive analysis report."""
    analyzer = TurbulenceAnalyzer(params.N, params.L, params.nu)
    results = analyzer.analyze(velocity, compute_all=True)

    report = []
    report.append("=" * 60)
    report.append("TURBULENCE ANALYSIS REPORT")
    report.append("=" * 60)
    report.append(f"\nRMS velocity: {results['u_rms']:.4f}")
    report.append(f"TKE: {results['TKE']:.4f}")
    report.append(f"\nInertial range slope: {results['inertial_slope
    ']:.3f}")
    report.append(f"(Expected: -1.667)")
    report.append(f"Max divergence: {results['max_divergence']:.2e}")
    report.append(f"Derivative skewness: {results['skewness']:.3f} (
    Expected: ~-0.4)")
    report.append(f"Derivative flatness: {results['flatness']:.3f} (
    Gaussian: 3.0)")

    report_text = '\n'.join(report)
    if save_path:
        with open(save_path, 'w') as f:
            f.write(report_text)
    print(report_text)
    return results
```

11.9 Summary

11.9.1 Key Functions

Function	Purpose
`compute_energy_spectrum_3d`	$E(k)$ via shell averaging
`compute_structure_function`	Velocity increments $S_n(r)$
`estimate_dissipation`	ϵ from gradients
`TurbulenceAnalyzer.analyze`	Full statistical analysis
`evaluate_reconstruction`	Model quality metrics

11.9.2 Key Metrics for Model Validation

Metric	Good Value	What it Checks
Inertial slope	≈ -1.67	K41 scaling
Max divergence	$< 10^{-6}$	Incompressibility
Spectrum WMAPE	$< 10\%$	Spectral fidelity
Skewness	≈ -0.4	Physical realism

Chapter 12

Accelerated Computing

This chapter covers GPU acceleration and performance optimization for large-scale turbulence computations.

12.1 Overview

12.1.1 Why Acceleration Matters

Turbulence simulations are computationally intensive:

Grid Size	Points	FFT Cost	Memory
64^3	262K	∼1 ms	4 MB
128^3	2.1M	∼10 ms	32 MB
256^3	16.8M	∼100 ms	256 MB
512^3	134M	∼1 s	2 GB

GPU acceleration provides **10–50x speedup** for:

- FFT operations (cuFFT)
- Convolutions (cuDNN)
- Data generation
- Spectrum computation

12.2 Module Structure

```
src/accelerate.py
+-- GPU detection/setup
+-- AcceleratedSpectralFilter
+-- AcceleratedTurbulenceGenerator
+-- AcceleratedAnalyzer
+-- Memory management utilities
+-- Batched operations
+-- Benchmarking tools
```

12.3 GPU Detection and Setup

```
import torch
import numpy as np

def get_device(prefer_gpu=True):
    """Get best available device."""
    if prefer_gpu and torch.cuda.is_available():
        device = torch.device('cuda')
        print(f"Using GPU: {torch.cuda.get_device_name(0)}")
        print(f"Memory: {torch.cuda.get_device_properties(0).
   total_memory / 1e9:.1f} GB")
    else:
        device = torch.device('cpu')
        print("Using CPU")
    return device

def check_gpu_memory():
    """Report GPU memory usage."""
    if torch.cuda.is_available():
        allocated = torch.cuda.memory_allocated() / 1e9
        reserved = torch.cuda.memory_reserved() / 1e9
        total = torch.cuda.get_device_properties(0).total_memory / 1e9
        print(f"GPU Memory: {allocated:.2f} / {total:.1f} GB allocated")
        return allocated, reserved, total
    return 0, 0, 0

def clear_gpu_cache():
    """Clear GPU cache to free memory."""
    if torch.cuda.is_available():
        torch.cuda.empty_cache()
        torch.cuda.synchronize()
```

12.4 Accelerated Spectral Filter

```
class AcceleratedSpectralFilter:
    """GPU-accelerated spectral filtering."""

    def __init__(self, N, L=2*np.pi, device='cuda'):
        self.N = N
        self.L = L
        self.device = torch.device(device)
        self._setup_wavenumbers()

    def _setup_wavenumbers(self):
        """Setup wavenumber arrays on GPU."""
        k1d = torch.fft.fftfreq(self.N, d=self.L/(2*np.pi*self.N),
                                device=self.device)
        kx, ky, kz = torch.meshgrid(k1d, k1d, k1d, indexing='ij')
        self.k_mag = torch.sqrt(kx**2 + ky**2 + kz**2)
        self.kx, self.ky, self.kz = kx, ky, kz

    @torch.no_grad()
    def filter_batch(self, velocity_batch, R, filter_type='gaussian'):
        """
        Filter a batch of velocity fields on GPU.
```

```
        Args:
            velocity_batch: (B, 3, N, N, N) tensor on GPU
            R: Filter size
            filter_type: 'gaussian', 'sharp', or 'box'
        """
        if filter_type == 'gaussian':
            G_hat = torch.exp(-self.k_mag**2 * R**2 / 24)
        elif filter_type == 'sharp':
            k_cutoff = np.pi / R
            G_hat = (self.k_mag <= k_cutoff).float()
        else:  # box
            kR = self.k_mag * R + 1e-10
            G_hat = 3 * (torch.sin(kR) - kR * torch.cos(kR)) / (kR**3)

        B, C = velocity_batch.shape[:2]
        filtered = torch.zeros_like(velocity_batch)

        for c in range(C):
            v_hat = torch.fft.fftn(velocity_batch[:, c], dim=(-3, -2,
    -1))
            filtered[:, c] = torch.fft.ifftn(v_hat * G_hat, dim=(-3, -2,
     -1)).real

        return filtered
```

12.5 Accelerated Turbulence Generation

```
class AcceleratedTurbulenceGenerator:
    """GPU-accelerated synthetic turbulence generation."""

    def __init__(self, N, L=2*np.pi, device='cuda'):
        self.N = N
        self.L = L
        self.device = torch.device(device)
        self._setup_spectral_arrays()

    def _setup_spectral_arrays(self):
        """Pre-compute spectral arrays on GPU."""
        k1d = torch.fft.fftfreq(self.N, d=self.L/(2*np.pi*self.N),
                                device=self.device)
        self.kx, self.ky, self.kz = torch.meshgrid(k1d, k1d, k1d,
    indexing='ij')
        self.k_mag = torch.sqrt(self.kx**2 + self.ky**2 + self.kz**2)
        self.k_mag[0, 0, 0] = 1.0  # Avoid division by zero

        self.k_sq = self.kx**2 + self.ky**2 + self.kz**2
        self.k_sq_safe = torch.where(self.k_sq > 0, self.k_sq,
                                     torch.ones_like(self.k_sq))

    @torch.no_grad()
    def generate_batch(self, batch_size, epsilon=1.0, u_rms=1.0, seed=
    None):
        """Generate batch of turbulent velocity fields."""
        if seed is not None:
            torch.manual_seed(seed)
```

```
        # Energy spectrum (Kolmogorov)
        C_k = 1.5
        dk = 2 * np.pi / self.L
        E_k = C_k * epsilon**(2/3) * self.k_mag**(-5/3)
        E_k[0, 0, 0] = 0

        # Mode energy and amplitude
        mode_energy = E_k * dk**3 / (4 * np.pi * self.k_mag**2 + 1e-10)
    / 3
        amplitude = torch.sqrt(torch.clamp(mode_energy, min=0))

        # Generate random complex fields
        velocity_hat = torch.zeros(batch_size, 3, self.N, self.N, self.N
    ,
                                   dtype=torch.cfloat, device=self.
    device)

        for b in range(batch_size):
            for c in range(3):
                real = torch.randn(self.N, self.N, self.N, device=self.
    device)
                imag = torch.randn(self.N, self.N, self.N, device=self.
    device)
                velocity_hat[b, c] = amplitude * (real + 1j * imag) / np
    .sqrt(2)

        # Divergence-free projection
        velocity_hat = self._project_divergence_free(velocity_hat)

        # Transform to physical space
        velocity = torch.zeros(batch_size, 3, self.N, self.N, self.N,
                               device=self.device)
        for c in range(3):
            velocity[:, c] = torch.fft.ifftn(
                velocity_hat[:, c], dim=(-3, -2, -1)
            ).real * self.N**3

        # Normalize to target u_rms
        u_rms_actual = torch.sqrt((velocity**2).mean(dim=(1,2,3,4),
    keepdim=True))
        velocity = velocity * u_rms / (u_rms_actual + 1e-10)

        return velocity
```

12.6 Accelerated Analysis

```
class AcceleratedAnalyzer:
    """GPU-accelerated turbulence analysis."""

    def __init__(self, N, L=2*np.pi, device='cuda'):
        self.N = N
        self.L = L
        self.device = torch.device(device)
        self._setup_wavenumbers()
        self._setup_bins()
```

```
    def _setup_bins(self):
        k_max = np.sqrt(3) * (np.pi * self.N / self.L)
        self.num_bins = self.N // 2
        self.k_bins = torch.linspace(0, k_max, self.num_bins + 1, device
    =self.device)
        self.k_centers = 0.5 * (self.k_bins[:-1] + self.k_bins[1:])
        self.dk = self.k_bins[1] - self.k_bins[0]

        # Pre-compute bin indices for fast shell averaging
        self.bin_indices = torch.bucketize(self.k_mag.flatten(), self.
    k_bins) - 1
        self.bin_indices = torch.clamp(self.bin_indices, 0, self.
    num_bins - 1)

    @torch.no_grad()
    def compute_spectrum_batch(self, velocity_batch):
        """Compute energy spectrum for batch of velocity fields."""
        B = velocity_batch.shape[0]

        u_hat = torch.fft.fftn(velocity_batch[:, 0], dim=(-3, -2, -1))
        v_hat = torch.fft.fftn(velocity_batch[:, 1], dim=(-3, -2, -1))
        w_hat = torch.fft.fftn(velocity_batch[:, 2], dim=(-3, -2, -1))

        energy = 0.5 * (u_hat.abs()**2 + v_hat.abs()**2 + w_hat.abs()
    **2)
        energy_flat = energy.view(B, -1)

        # Fast shell averaging using scatter_add
        E_k = torch.zeros(B, self.num_bins, device=self.device)
        E_k.scatter_add_(1, self.bin_indices.unsqueeze(0).expand(B, -1),
     energy_flat)
        E_k = E_k / self.N**6 / self.dk

        return self.k_centers.cpu().numpy(), E_k.cpu().numpy()
```

12.7 Memory Management

```
def process_large_dataset(velocity_files, processor, batch_size=4,
    device='cuda'):
    """Process large dataset with memory management."""
    results = []

    for i in range(0, len(velocity_files), batch_size):
        batch_files = velocity_files[i:i+batch_size]
        batch = torch.stack([load_velocity(f) for f in batch_files])
        batch = batch.to(device)

        with torch.no_grad():
            result = processor(batch)

        results.append(result.cpu())

        del batch
        torch.cuda.empty_cache()
```

```
    return torch.cat(results, dim=0)


class MemoryEfficientDataLoader:
    """DataLoader that manages GPU memory carefully."""

    def __init__(self, dataset, batch_size, device='cuda', prefetch=2):
        self.dataset = dataset
        self.batch_size = batch_size
        self.device = torch.device(device)
        self.prefetch = prefetch

    def __iter__(self):
        indices = list(range(len(self.dataset)))
        np.random.shuffle(indices)

        for i in range(0, len(indices), self.batch_size):
            batch_idx = indices[i:i+self.batch_size]
            batch = [self.dataset[j] for j in batch_idx]

            batch_tensor = torch.stack([b['velocity'] for b in batch])
            batch_tensor = batch_tensor.to(self.device, non_blocking=
    True)

            yield batch_tensor

            del batch_tensor
            if i % (self.batch_size * 10) == 0:
                torch.cuda.empty_cache()
```

12.8 Benchmarking Tools

```
def benchmark_operation(operation, *args, num_runs=10, warmup=3, device=
    'cuda'):
    """Benchmark a GPU operation."""
    # Warmup
    for _ in range(warmup):
        _ = operation(*args)
        if device == 'cuda':
            torch.cuda.synchronize()

    # Timed runs
    times = []
    for _ in range(num_runs):
        if device == 'cuda':
            torch.cuda.synchronize()
            start = torch.cuda.Event(enable_timing=True)
            end = torch.cuda.Event(enable_timing=True)
            start.record()

        _ = operation(*args)

        if device == 'cuda':
            end.record()
            torch.cuda.synchronize()
            times.append(start.elapsed_time(end))
```

```
    times = np.array(times)
    return times.mean(), times.std()


def benchmark_all_operations(N=128, device='cuda'):
    """Run comprehensive benchmark suite."""
    print(f"\n{'='*60}")
    print(f"BENCHMARK RESULTS (N={N}, device={device})")
    print(f"{'='*60}")

    gen = AcceleratedTurbulenceGenerator(N, device=device)
    filt = AcceleratedSpectralFilter(N, device=device)
    analyzer = AcceleratedAnalyzer(N, device=device)

    mean_t, std_t = benchmark_operation(gen.generate_batch, 1)
    print(f"Generate 1 field:     {mean_t:.2f} +/- {std_t:.2f} ms")

    mean_t, std_t = benchmark_operation(gen.generate_batch, 8)
    print(f"Generate 8 fields:    {mean_t:.2f} +/- {std_t:.2f} ms")
```

12.9 Multi-GPU Support

```
def setup_distributed(rank, world_size):
    """Initialize distributed training."""
    import torch.distributed as dist

    dist.init_process_group(
        backend='nccl', init_method='env://',
        world_size=world_size, rank=rank
    )
    torch.cuda.set_device(rank)


class DistributedTurbulenceGenerator:
    """Generate turbulence across multiple GPUs."""

    def __init__(self, N, L=2*np.pi, world_size=1, rank=0):
        self.N = N
        self.L = L
        self.world_size = world_size
        self.rank = rank
        self.device = torch.device(f'cuda:{rank}')
        self.local_gen = AcceleratedTurbulenceGenerator(N, L, self.
   device)

    def generate_distributed_batch(self, total_batch_size, seed=None):
        """Generate batch distributed across GPUs."""
        local_batch_size = total_batch_size // self.world_size

        if seed is not None:
            local_seed = seed + self.rank * 1000
        else:
            local_seed = None
```

```
        return self.local_gen.generate_batch(local_batch_size, seed=
    local_seed)
```

12.10 Performance Tips

12.10.1 Best Practices

Tip	Speedup	When to Use
Use `torch.no_grad()`	2–3x	Inference/analysis
Batch operations	5–10x	Multiple samples
Pre-compute wavenumbers	2x	Repeated operations
Use `float32` not `float64`	2x	Most cases
Pin memory for CPU→GPU	1.5x	Data loading
Use `non_blocking=True`	1.2x	Async transfers

12.10.2 Memory Optimization

```
# Use mixed precision
with torch.cuda.amp.autocast():
    output = model(input)

# Gradient checkpointing for training
from torch.utils.checkpoint import checkpoint
output = checkpoint(expensive_layer, input)

# In-place operations where safe
x.add_(1)  # Instead of x = x + 1
```

12.10.3 Typical Speedups

Operation	CPU Time	GPU Time	Speedup
Generate 128^3	500 ms	30 ms	17x
FFT 128^3	50 ms	3 ms	17x
Filter 128^3	100 ms	6 ms	17x
Spectrum 128^3	80 ms	5 ms	16x
Generate 256^3	4000 ms	200 ms	20x

12.11 Summary

12.11.1 Key Classes

Class	Purpose
`AcceleratedSpectralFilter`	GPU filtering
`AcceleratedTurbulenceGenerator`	GPU synthesis
`AcceleratedAnalyzer`	GPU analysis

12.11.2 Key Optimizations

1. **Pre-compute arrays** on GPU at initialization time
2. **Batch operations** where possible
3. **Use `torch.no_grad()`** for inference
4. **Clear cache** periodically for long runs
5. **Pin memory** for data loading

Quick Reference

Key Equations

Equation	Description
$\boldsymbol{\nabla} \cdot \mathbf{u} = 0$	Incompressibility constraint
$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \boldsymbol{\nabla})\mathbf{u} = -\boldsymbol{\nabla} p + \nu \nabla^2 \mathbf{u}$	Navier-Stokes
$\mathrm{Re} = UL/\nu$	Reynolds number
$\eta = (\nu^3/\varepsilon)^{1/4}$	Kolmogorov scale
$E(k) = C_K \varepsilon^{2/3} k^{-5/3}$	Energy spectrum (K41)
$S_3(r) = -\frac{4}{5}\varepsilon r$	Four-fifths law (exact)

Key Parameters

Symbol	Name	Typical Value
N	Grid resolution	64–256
Re_λ	Taylor Reynolds number	100–500
η	Kolmogorov scale	~ 0.01–0.1
C_K	Kolmogorov constant	≈ 1.5
R	Filter width	0.1–0.5

Code Quick Start

```
# Generate turbulence
from src.synthetic_turbulence import TurbulenceParams,
    generate_isotropic_turbulence_spectral
params = TurbulenceParams(N=128, Re_lambda=200, seed=42)
velocity = generate_isotropic_turbulence_spectral(params)

# Filter
from src.filtering import SpectralFilter, FilterType
sf = SpectralFilter(params.N, params.L)
V_coarse = sf.filter_field(velocity, R=0.3, filter_type=FilterType.
    GAUSSIAN)

# Analyze
from src.analysis import compute_energy_spectrum_3d
k, E_k = compute_energy_spectrum_3d(velocity, params.L)
```