近似GKP态FNO恢复系统代码文档

项目概述

本项目实现了基于傅里叶神经算子(FNO)的近似GKP(Gottesman-Kitaev-Preskill)量子纠错码态恢复系统，用于从带噪声的Wigner函数中恢复量子态或预测纠错位移。

物理背景

- **近似GKP态**：物理上实际可制备的GKP态不是理想的δ函数和，而是带有高斯包络的有限能量态
- **数学表达**：$\tilde{|}\mu\_L\rangle \propto \exp(-\Delta^2\hat{n})|\mu\_L\rangle$，其中Δ是压缩参数
- **Wigner函数**：相空间中表现为"高斯格点阵列"，具有周期性晶格结构
- **噪声模型**：包括高斯位移噪声、光子损耗和退相位

文件结构与功能说明

1. `__init__.py` - 包初始化文件

功能：定义包的公共接口，导出核心类和函数。

导出内容：

- `Config`，`get_default_config` - 配置管理
- `ApproxGKPSimulator` - 物理模拟器
- `FNODisplacementDecoder`，`FNOStateReconstructor`，`FNOHybridModel`，`create_model` - FNO模型

版本信息：`__version__ = '0.1.0'`

2. `config.py` - 配置管理模块

功能：使用数据类(dataclass)定义系统的所有配置参数，提供多种预设配置。

主要类

2.1 `PhysicsConfig` - 物理参数配置

核心属性:

- `n_hilbert: int = 50` - Hilbert空间截断维数
- `delta: float = 0.3` - 压缩参数Δ（越小越接近理想GKP态，实验范围0.2-0.4对应~10-14 dB）
- `delta_range: Tuple[float, float]` - 训练数据增强的Δ范围
- `grid_size: int = 64` - Wigner函数计算的网格尺寸
- `phase_space_extent: float = 6.0` - 相空间范围
- `noise_sigma: float = 0.15` - 高斯位移噪声标准差
- `kappa: float = 0.01` - 光子损耗率
- `kappa_phi: float = 0.005` - 退相位率

2.2 `ModelConfig` - FNO模型配置

核心属性:

- `in_channels: int = 1` - 输入通道数（Wigner函数为单通道）
- `width: int = 32` - 隐藏通道宽度
- `modes: int = 16` - 保留的傅里叶模态数（决定能捕捉多精细的干涉条纹）
- `n_layers: int = 4` - 傅里叶层数量
- `output_mode: str = 'displacement'` - 输出模式：'displacement'(2D位移向量)或'reconstruction'(Wigner函数重建)
- `activation: str = 'gelu'` - 激活函数
- `use_residual: bool = True` - 是否使用残差连接

2.3 `TrainingConfig` - 训练配置

核心属性:

- `batch_size: int = 32` - 批大小
- `epochs: int = 200` - 训练轮数
- `learning_rate: float = 1e-3` - 学习率
- `scheduler: str = 'cosine'` - 学习率调度器类型
- `lambda_mse: float = 1.0` - MSE损失权重
- `lambda_fidelity: float = 0.5` - 保真度损失权重
- `lambda_stabilizer: float = 0.1` - 稳定器约束损失权重
- `train_samples: int = 5000` - 训练样本数
- `online_learning: bool = False` - 是否在线生成数据
- `checkpoint_dir: str = './checkpoints'` - 检查点保存目录

- `early_stopping_patience: int = 30` - 早停耐心值

2.4 `EvaluationConfig` - 评估配置

核心属性:

- `compute_fidelity: bool = True` - 是否计算保真度
- `compare_homodyne_binning: bool = True` - 是否与基线方法比较
- `plot_wigner: bool = True` - 是否绘制Wigner函数

2.5 `Config` - 主配置类

整合所有子配置，包含:

- `physics: PhysicsConfig`
- `model: ModelConfig`
- `training: TrainingConfig`
- `evaluation: EvaluationConfig`
- `exp_name: str` - 实验名称

预设配置函数

- `get_default_config()` - 默认配置
- `get_high_squeezing_config()` - 高压缩配置(Δ=0.2, ~14dB)
- `get_low_squeezing_config()` - 低压缩配置(Δ=0.4, ~8dB)
- `get_reconstruction_config()` - 态重建模式配置

3. `physics_simulator.py` - 物理模拟器

功能：基于PRX Quantum论文Sec II B实现近似GKP态的生成和噪声通道模拟。

主要类和数据结构

3.1 `GKPStateData` - GKP态数据容器

属性:

- `wigner: np.ndarray` - Wigner函数 (grid_size × grid_size)
- `displacement: np.ndarray` - 施加的位移噪声 (u, v)
- `delta: float` - 压缩参数
- `logical_value: int` - 逻辑值 (0代表|0_L〉, 1代表|1_L〉, -1代表叠加态)

- `superposition_coeffs: Optional[Tuple[complex, complex]]` - 叠加态系数

3.2 `ApproxGKPSimulator` - 近似GKP态模拟器

初始化参数:

- `n_hilbert: int = 50` - Hilbert空间维数
- `delta: float = 0.3` - 压缩参数
- `grid_size: int = 64` - Wigner函数网格尺寸
- `phase_space_extent: float = 6.0` - 相空间范围

核心方法:

3.2.1 `_construct_approx_state(logical_val: int) -> qt.Qobj`

构造近似GKP码态，实现论文Eq. (11)。

- **输入**：logical_val (0或1)
- **输出**：归一化的QuTiP量子态
- **实现细节**：
  - 使用压缩真空态的加权和
  - 晶格位置：$(2k + \mu) \times \sqrt{\pi}$
  - 高斯包络权重：$\exp(-0.5 \times (\Delta \times \text{shift})^2)$
  - 自动确定求和范围以适应Hilbert空间截断

3.2.2 `get_logical_state(...) -> qt.Qobj`

获取逻辑GKP态（基态或叠加态）。

- **参数**:
  - `logical_val`：0或1表示基态，None表示叠加态
  - `alpha, beta`：叠加态系数
  - `delta`：可选的不同Δ值
- **输出**：QuTiP量子态

3.2.3 `apply_displacement_noise(state, sigma) -> Tuple[qt.Qobj, np.ndarray]`

施加高斯位移噪声通道。

- **物理公式**：$D(\zeta)$，其中$\zeta \sim N(0, \sigma^2)$，基于论文Eq. (26)
- **输入**：量子态、噪声标准差
- **输出**：(噪声态, 位移向量[u, v])

3.2.4 `apply_loss_channel(state, kappa, time) -> qt.Qobj`

通过Lindblad演化施加光子损耗通道。

- **物理公式**：dρ/dt = κ × D[a]ρ，基于论文Eq. (16)
- **输入**：量子态、损耗率、演化时间
- **输出**：演化后的密度矩阵

3.2.5 `apply_dephasing_channel(state, kappa_phi, time) -> qt.Qobj`

施加退相位通道。

- **物理公式**：κ_φ × D[a†a]ρ
- **输入**：量子态、退相位率、演化时间
- **输出**：演化后的密度矩阵

3.2.6 `compute_wigner(state, xvec, pvec) -> np.ndarray`

计算量子态的Wigner函数。

- **输入**：量子态、相空间网格点
- **输出**：2D Wigner函数数组

3.2.7 `generate_sample(noise_sigma, noise_type, ...) -> GKPStateData`

生成单个训练样本。

- **参数**：
  - `noise_type`：'displacement', 'loss', 'combined'
  - `random_logical`：是否随机选择|0_L〉或|1_L〉
  - `random_superposition`：是否包含随机叠加态
- **输出**：GKPStateData对象

3.2.8 `generate_batch(batch_size, ...) -> Dict[str, np.ndarray]`

生成批量训练数据。

- **输出字典**：
  - `'wigner'`：(batch, 1, grid, grid) 噪声Wigner函数
  - `'displacement'`：(batch, 2) 位移向量
  - `'delta'`：(batch,) 压缩参数
  - `'wigner_clean'`：干净Wigner函数（可选）

辅助函数

- `compute_squeezing_db(delta: float) -> float` - 将Δ转换为dB单位的压缩度
- `delta_from_squeezing_db(s_db: float) -> float` - 从dB转换回Δ

4. `fno_model.py` - FNO神经网络模型

功能：实现傅里叶神经算子，用于GKP态恢复。结合了频谱卷积和多模态特征融合概念。

核心组件

4.1 `SpectralConv2d` - 2D频谱卷积层

FNO的核心模块，在傅里叶空间中进行卷积。

初始化参数：

- `in_channels: int` - 输入通道维度
- `out_channels: int` - 输出通道维度
- `modes1, modes2: int` - 保留的傅里叶模态数

核心属性：

- `weights1, weights2: nn.Parameter` - 复数张量权重，用于频域模态的学习

关键方法：

- `compl_mul2d(input, weights)` - 频域复数乘法
  - 使用爱因斯坦求和约定： `"bixy,ioxy->boxy"`
- `forward(x)` - 前向传播
  i. 通过 `rfft2` 变换到傅里叶空间
  ii. 对保留的低频模态乘以学习权重（左上角和左下角，处理共轭对称性）
  iii. 通过 `irfft2` 逆变换回物理空间

物理意义：GKP态的周期晶格结构对应频域中的特定模式，FNO能有效捕捉这些全局周期特性。

4.2 `FNOBlock` - FNO模块

单个FNO块，包含频谱卷积和残差连接。

组件：

- `spectral_conv: SpectralConv2d` - 频谱卷积路径
- `conv: nn.Conv2d` - 1×1卷积路径（局部通道混合）
- `norm: nn.InstanceNorm2d` - 实例归一化
- `activation` - 激活函数（GELU/ReLU/SiLU）

前向传播:

```
output = spectral_conv(x) + conv(x)  # 双路径融合
output = norm(output)
output = activation(output)
if use_residual:
    output = output + x  # 残差连接
```

4.3 `FNODisplacementDecoder` - 位移预测解码器

功能：从噪声Wigner函数预测纠错位移向量。

网络结构:

- **输入**：(B, 1, H, W) 噪声Wigner函数
- **输出**：(B, 2) 位移向量(u, v)

架构:

1. `lift`：1×1卷积，将输入提升到高维特征空间
2. `fno_blocks`：n_layers个FNOBlock
3. `pool`：全局平均池化，聚合空间信息
4. `head`：MLP投影头
   - Linear(width → width×2) + GELU + Dropout
   - Linear(width×2 → width) + GELU
   - Linear(width → 2)

设计原理:

- FNO捕捉全局位移在频域的相位特征
- 全局池化确保位移预测对整个相空间敏感
- 多层MLP提取抽象位移信息

4.4 `FNOStateReconstructor` - 态重建器

功能：从噪声Wigner函数重建干净的Wigner函数。

网络结构:

- **输入**：(B, 1, H, W) 噪声Wigner函数
- **输出**：(B, 1, H, W) 重建的干净Wigner函数

架构：

1. `lift`：提升层
2. `fno_blocks`：FNO层序列
3. `project`：投影层
   - Conv2d(width → width/2) + GELU
   - Conv2d(width/2 → 1)

应用场景：研究FNO是否能学到近似态的流形结构。

4.5 `FNOHybridModel` - 混合模型

功能：同时输出位移和重建态，提供更强的训练信号。

输出：

- `displacement`：(B, 2) 位移向量
- `reconstruction`：(B, 1, H, W) 重建Wigner函数

架构：

- 共享编码器（FNO blocks）
- 双分支头：
  - 位移分支：全局池化 + MLP
  - 重建分支：卷积投影

工厂函数

`create_model(config) -> nn.Module`

根据配置创建相应的FNO模型。

- `config.output_mode == 'displacement'` → FNODisplacementDecoder
- `config.output_mode == 'reconstruction'` → FNOStateReconstructor
- `config.output_mode == 'hybrid'` → FNOHybridModel

辅助函数

- `count_parameters(model)` - 统计模型可训练参数数量

5. `dataset.py` - 数据集模块

功能：提供离线和在线两种数据生成方式，支持PyTorch数据加载。

主要类

5.1 `GKPDataset` - GKP数据集

支持预生成数据和即时生成两种模式。

初始化参数：

- `simulator: ApproxGKPSimulator` - 物理模拟器实例
- `n_samples: int` - 样本数量
- `noise_sigma: float` - 噪声标准差
- `noise_type: str` - 噪声类型
- `delta_range: Optional[Tuple]` - Δ采样范围
- `online: bool` - 是否在线生成
- `return_clean: bool` - 是否返回干净态
- `data_path: Optional[str]` - 数据保存/加载路径

核心方法：

- `_generate_offline_data()` - 离线生成全部数据
- `_save_data(path)` - 保存数据到磁盘（pickle格式）
- `_load_data(path)` - 从磁盘加载数据
- `__getitem__(idx)` - 获取单个样本
  - **在线模式**：即时生成样本
  - **离线模式**：从预生成数据读取
  - **返回格式**：字典包含'wigner', 'displacement', 'delta'

5.2 `OnlineGKPDataset` - 在线数据集

每个epoch生成新样本，更节省内存并防止过拟合。

特点：

- 使用缓存机制（默认缓存100个样本）
- `_refill_cache()` - 缓存耗尽时重新填充
- 适合大规模数据集训练

5.3 `create_dataloaders(...)` - 数据加载器工厂函数

一站式创建训练、验证、测试数据加载器。

参数：

- 样本数量： `train_samples` ， `val_samples` ， `test_samples`
- 数据参数： `batch_size` ， `noise_sigma` ， `noise_type` ， `delta_range`
- 模式参数： `online` ， `return_clean`
- PyTorch参数： `num_workers`
- 存储： `data_dir` - 数据保存/加载目录

返回： `(train_loader, val_loader, test_loader)`

5.4 `DataNormalizer` - 数据归一化器

对Wigner函数数据进行归一化，提升训练效果。

方法：

- `'standard'` - 零均值单位方差归一化：(x - μ) / σ
- `'minmax'` - 最小-最大归一化：(x - min) / (max - min)

核心方法：

- `fit(data)` - 拟合归一化参数
- `transform(data)` - 应用归一化
- `inverse_transform(data)` - 逆变换

6. `train.py` - 训练脚本

功能：完整的训练流程，包括损失函数、训练器类、命令行接口。

核心组件

6.1 `CombinedLoss` - 组合损失函数

整合多种损失以确保物理一致性。

损失组件：

1. **MSE损失**： `lambda_mse × MSE(pred, target)`

2. **Sobolev平滑损失**（重建模式）：
    - 计算空间梯度： `∂W/∂x` ， `∂W/∂y`
    - 约束梯度误差： `MSE(∇pred, ∇target)`
    - 保持物理平滑性

初始化参数：

- `lambda_mse: float = 1.0` - MSE权重
- `lambda_smooth: float = 0.1` - 平滑性权重
- `output_mode: str` - 输出模式

方法：

- `gradient_loss(pred, target)` - 计算梯度（Sobolev）损失
- `forward(pred, target, pred_recon, target_recon)` - 组合损失

6.2 `Trainer` - 训练器类

封装完整训练流程。

初始化（ `__init__` ）：

1. 创建实验目录
2. 保存配置文件（JSON格式）
3. 初始化物理模拟器
4. 初始化FNO模型
5. 创建数据加载器
6. 初始化优化器和调度器
7. 初始化损失函数
8. 可选：TensorBoard日志

核心方法：

6.2.1 `_init_simulator()`

初始化物理模拟器，使用配置参数。

6.2.2 `_init_model()`

创建并初始化FNO模型，打印参数数量。

6.2.3 `_init_dataloaders()`

根据输出模式决定是否返回干净态，创建数据加载器。

6.2.4 `_init_optimizer()`

初始化AdamW优化器和学习率调度器：

- **Cosine退火**： `CosineAnnealingLR`
- **阶梯衰减**： `StepLR`
- **平台衰减**： `ReduceLROnPlateau`

6.2.5 `_init_loss()`

初始化组合损失函数。

6.2.6 `train_epoch(epoch: int) -> float`

训练一个epoch。

- 遍历训练数据批次
- 根据输出模式选择前向传播路径
- 梯度裁剪（max_norm=1.0）
- 优化器更新
- 返回平均损失

6.2.7 `validate() -> dict`

验证模型性能。

- 无梯度计算
- 计算验证损失
- **位移模式额外指标**：
  - MAE（平均绝对误差）
  - RMSE（均方根误差）
  - 分量误差（mae_u, mae_v）
- 返回指标字典

6.2.8 `train()`

完整训练循环。

- **流程**：
  i. 训练一个epoch
  ii. 验证
  iii. 更新学习率调度器
  iv. TensorBoard日志记录
  v. 保存最佳模型

vi. 定期保存检查点
vii. 早停检查

- **早停机制**：连续N个epoch验证损失未改善则停止
- **检查点**：保存best.pt, epoch_N.pt, final.pt

6.2.9 `_save_checkpoint(filename, epoch, metrics)`

保存模型检查点，包含：

- epoch
- model_state_dict
- optimizer_state_dict
- metrics（验证指标）
- config（物理和模型配置）

命令行接口（`main()`）

参数：

- `--config` - 配置预设（default, high_squeezing, low_squeezing, reconstruction）
- `--epochs` - 覆盖训练轮数
- `--batch_size` - 覆盖批大小
- `--lr` - 覆盖学习率
- `--delta` - 覆盖压缩参数
- `--noise_sigma` - 覆盖噪声标准差
- `--device` - 覆盖设备
- `--exp_name` - 覆盖实验名称

流程：

1. 加载配置预设
2. 应用命令行参数覆盖
3. 设置随机种子
4. 创建Trainer并开始训练

7. `evaluate.py` - 评估脚本

功能：模型评估、基线对比、可视化。

基线方法

7.1 `homodyne_binning_decoder(wigner, xvec, pvec)`

基于零差测量+分箱的基线解码器。

算法（基于论文Sec II D1）：

1. 找到Wigner函数峰值位置
2. 舍入到最近的GKP晶格点（间隔$\sqrt{\pi}$）
3. 计算偏移量作为位移估计

输入：Wigner函数、相空间网格
输出：估计位移(u, v)

7.2 `compute_logical_error_rate(true_disp, pred_disp, threshold)`

计算逻辑错误率。

定义：当纠错后的残余位移跨越晶格边界时发生逻辑错误。

算法：

- 残余位移 = 真实位移 - 预测位移
- 阈值默认为$\sqrt{\pi}/2$
- 错误率 = 残余位移超过阈值的比例

评估函数

7.3 `evaluate_model(model, simulator, n_samples, noise_sigmas, ...)`

全面评估模型性能。

评估流程：

1. 检测模型输出模式（位移/重建）
2. 对每个噪声水平：
    - 生成测试数据
    - FNO预测
    - 基线预测（零差+分箱）
    - 计算指标：
        - **MAE**（平均绝对误差）
        - **RMSE**（均方根误差）
        - **逻辑错误率**

3. 返回所有噪声水平的结果字典

7.4 `evaluate_reconstruction_model(model, simulator, ...)`

评估重建模型。

指标：

- **FNO重建MSE**：与干净态的均方误差
- **PSNR**：峰值信噪比（dB）
- **噪声MSE**：基线比较

7.5 `evaluate_across_deltas(model, deltas, ...)`

评估模型在不同压缩水平下的性能。

输出：

- 每个Δ值的MAE和逻辑错误率
- 对应的压缩度（dB）

可视化函数

7.6 `plot_results(results, save_path)`

绘制评估结果对比图。

三个子图：

1. MAE vs 噪声水平
2. RMSE vs 噪声水平
3. 逻辑错误率 vs 噪声水平

每张图包含FNO和基线方法的对比曲线。

7.7 `plot_delta_results(results, save_path)`

绘制不同压缩水平的性能。

两个子图：

1. MAE vs 压缩度（dB）
2. 逻辑错误率 vs 压缩度（dB）

7.8 `plot_reconstruction_results(results, save_path)`

绘制重建模型结果。

两个子图：

1. MSE vs 噪声水平（对数坐标）
2. PSNR vs 噪声水平

7.9 `visualize_prediction(model, simulator, ...)`

可视化单个预测样例。

三个子图：

1. 干净GKP态的Wigner函数
2. 噪声态的Wigner函数
3. 位移预测对比柱状图（真实值、FNO、Binning）

模型加载

7.10 `load_model(checkpoint_path, device)`

从检查点加载训练好的模型。

返回：(model, config_dict)

主程序（`main()`）

命令行参数：

- `--checkpoint` - 模型检查点路径（必需）
- `--n_samples` - 测试样本数
- `--device` - 计算设备
- `--save_dir` - 结果保存目录

评估流程：

1. 加载模型和配置
2. 创建物理模拟器
3. 跨噪声水平评估
4. 保存结果（JSON）和图表（PNG）
5. 如果是位移模式：
    - 跨Δ值评估

- 可视化示例预测

8. `test_quick.py` - 快速测试脚本

功能：验证所有组件正常工作的集成测试。

测试函数

8.1 `test_physics_simulator()`

测试物理模拟器。

- 创建小规模模拟器（n_hilbert=30）
- 测试单样本生成
- 测试批量生成
- 验证输出形状

8.2 `test_fno_model()`

测试FNO模型。

- 测试位移解码器
- 测试态重建器
- 测试混合模型
- 验证输入输出形状
- 打印参数数量

8.3 `test_dataset(simulator)`

测试数据集。

- 创建离线数据集
- 测试样本获取
- 测试DataLoader批处理

8.4 `test_training_step(simulator, model)`

测试单步训练。

- 创建小数据集
- 设置优化器和损失
- 执行一次前向-反向传播

- 验证损失计算

8.5 `test_evaluation()`

测试评估函数。

- 测试零差分箱解码器
- 测试逻辑错误率计算

8.6 `run_quick_training()`

运行快速训练测试（3个epoch）。

- 小规模配置
- 简化训练循环
- 验证训练和验证流程

8.7 `main()`

运行所有测试。

- 顺序执行各测试函数
- 异常处理和追踪
- 成功时打印使用说明

9. `test_speed.py` - 速度测试脚本

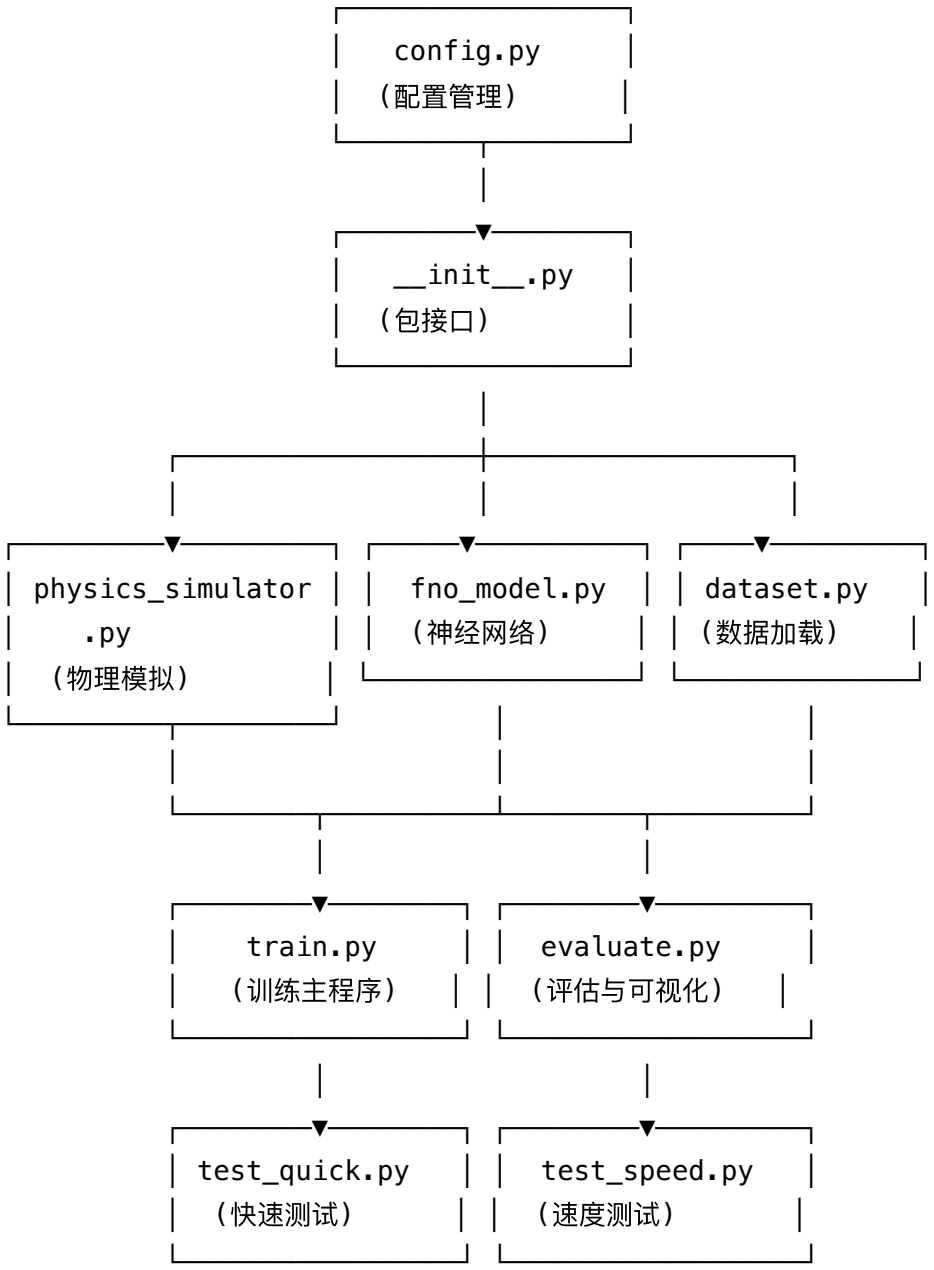
功能：测试数据生成速度。

测试内容：

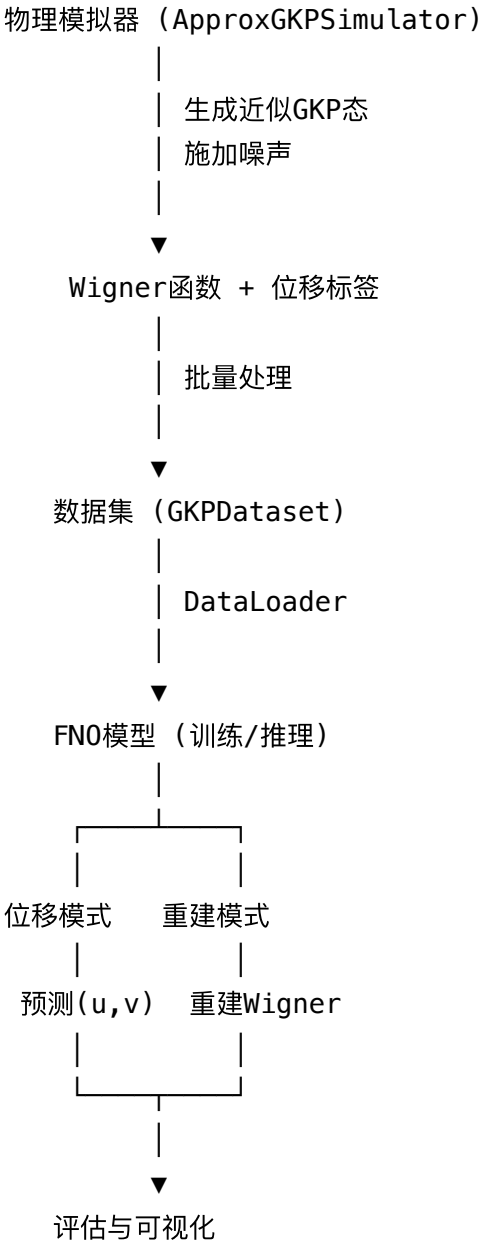
- 模拟器创建时间
- 批量生成50个样本的时间
- 验证输出形状

用途：快速检查Wigner函数计算效率（QuTiP的Wigner变换是CPU密集型操作）。

模块间关系图


config.py
(配置管理)
__init__.py
(包接口)
physics_simulator.py
(物理模拟)
fno_model.py
(神经网络)
dataset.py
(数据加载)
train.py
(训练主程序)
evaluate.py
(评估与可视化)
test_quick.py
(快速测试)
test_speed.py
(速度测试)

数据流图



关键交互流程

训练流程

1. **配置加载**： `config.py` → 选择预设或自定义配置
2. **物理初始化**： `physics_simulator.py` → 预计算逻辑基态
3. **数据生成**： `dataset.py` → 在线/离线生成训练数据
4. **模型创建**： `fno_model.py` → 根据配置构建FNO

5. **训练循环**： `train.py` → Trainer类执行训练
   - 每个epoch：train_epoch() → validate()
   - 保存检查点
   - TensorBoard日志
6. **评估**： `evaluate.py` → 加载最佳模型进行评估

评估流程

1. **加载模型**： `evaluate.py` → load_model()
2. **重建模拟器**：使用保存的配置
3. **生成测试数据**：physics_simulator生成不同噪声水平数据
4. **FNO预测**：模型前向传播
5. **基线对比**：零差+分箱方法
6. **指标计算**：MAE, RMSE, 逻辑错误率
7. **可视化**：matplotlib绘图

物理到代码的映射

物理概念	代码实现	位置
近似GKP态	˜μ_L〉	`_construct_approx_state()`
压缩参数Δ	`delta`	配置和模拟器
高斯包络	`exp(-0.5 * (delta * shift)²)`	态构造循环
GKP晶格	`(2k + μ) × √π`	位移位置计算
Wigner函数	`qt.wigner(state, xvec, pvec)`	compute_wigner()
位移噪声	`D(ζ)` where ζ~N(0,σ²)	apply_displacement_noise()
光子损耗	Lindblad: `κ×D[a]ρ`	apply_loss_channel()
纠错位移	(u, v)	FNO输出
频谱卷积	傅里叶模态学习	SpectralConv2d
逻辑错误	跨越√π/2边界	compute_logical_error_rate()

使用示例

快速开始

```
# 1. 快速测试所有组件
python test_quick.py

# 2. 运行默认配置训练
python train.py --epochs 100

# 3. 高压缩配置训练
python train.py --config high_squeezing --epochs 200

# 4. 评估训练好的模型
python evaluate.py --checkpoint checkpoints/fno_approx_gkp/best.pt --n_samples 500

# 5. 速度测试
python test_speed.py
```

自定义配置训练

```
python train.py \
    --delta 0.25 \
    --noise_sigma 0.2 \
    --batch_size 64 \
    --lr 5e-4 \
    --epochs 300 \
    --exp_name custom_experiment
```

扩展开发指南

添加新的噪声模型

在 `physics_simulator.py` 中添加新方法：

```
def apply_custom_noise(self, state, params):
    # 实现自定义噪声通道
    pass
```

添加新的FNO架构

在 `fno_model.py` 中创建新类：

```
class CustomFNOModel(nn.Module):
    def __init__(self, ...):
        # 自定义架构
        pass
```

在 `create_model()` 中添加新分支。

添加新的评估指标

在 `evaluate.py` 中添加新函数：

```
def compute_custom_metric(pred, target):
    # 计算自定义指标
    return metric_value
```

性能优化建议

1. **Wigner函数计算**：
    - 使用多进程并行化（ `multiprocessing.Pool` ）
    - 预生成数据集而非在线生成
    - 降低grid_size以加速（精度权衡）
2. **FNO模型**：
    - 调整 `modes` 参数平衡精度和速度
    - 使用混合精度训练（ `torch.cuda.amp` ）
    - 减少 `n_layers` 用于快速原型
3. **数据加载**：
    - 使用 `num_workers > 0` 进行多线程加载
    - 启用 `pin_memory=True` （GPU训练）

- 考虑在线数据集减少内存占用

常见问题

Q1: 训练损失不下降?

- 检查学习率（尝试1e-4到1e-3）
- 验证数据归一化
- 确保Δ和noise_sigma匹配实验场景
- 增加模型width和modes

Q2: 评估时逻辑错误率很高?

- 检查噪声水平是否过强
- 对比基线方法确认数据质量
- 可能需要更多训练数据或更长训练时间

Q3: 内存不足?

- 减小batch_size
- 减小grid_size
- 减小n_hilbert
- 使用在线数据集而非离线
- 降低train_samples数量

Q4: Wigner函数计算太慢?

- 降低grid_size（如32或48）
- 减小n_hilbert
- 使用预生成数据
- 运行test_speed.py检查基准性能

参考文献

- PRX Quantum论文Section II B：近似GKP态理论

- 方案1：频谱卷积和FNO架构设计
- 方案2：多模态特征融合和损失函数设计

版本信息

- **版本**：0.1.0
- **Python要求**：≥ 3.8
- **主要依赖**：
  - PyTorch ≥ 1.10
  - QuTiP ≥ 4.6
  - NumPy ≥ 1.20
  - Matplotlib ≥ 3.3

总结

本代码库实现了一个完整的基于FNO的近似GKP态恢复系统，涵盖：

1. **物理层**：精确的量子态模拟和噪声通道
2. **模型层**：先进的傅里叶神经算子架构
3. **数据层**：灵活的在线/离线数据生成
4. **训练层**：完整的训练流程和检查点管理
5. **评估层**：全面的性能评估和可视化

所有模块设计遵循物理原理，代码结构清晰，易于扩展和定制。