

## ✓ Convex first

```

1 !pip install fancyimpute
2 import numpy as np
3 import pandas as pd
4 import matplotlib.pyplot as plt
5 from fancyimpute import SoftImpute # Make sure this is installed
6 from sklearn.model_selection import train_test_split
7 import time
8 import warnings
9 from sklearn.exceptions import ConvergenceWarning # To potentially catch solver warnings
10 from google.colab import drive # Uncomment if using Colab
11 drive.mount("/content/drive", force_remount=True)
12 DRIVE_MOUNTED = True
13 # --- Configuration ---
14 # Use your correct path
15 DATA_PATH = "/content/drive/MyDrive/ml-1m/ratings.dat"
16 TEST_SIZE = 0.2
17 np.random.seed(42)
18
19 # --- Load MovieLens 1M Data ---
20 try:
21     ratings = pd.read_csv(
22         DATA_PATH, sep=":", engine='python',
23         names=["UserID", "MovieID", "Rating", "Timestamp"]
24     ).drop(columns=['Timestamp'])
25 except FileNotFoundError:
26     print(f"ERROR: Ratings file not found at {DATA_PATH}")
27     exit() # Or raise Exception(...)
28
29 # Adjust index to be 0-based if necessary (check max IDs vs shape later)
30 ratings['UserID'] -= 1
31 ratings['MovieID'] -= 1
32 num_users = ratings['UserID'].max() + 1
33 num_movies = ratings['MovieID'].max() + 1
34
35 print(f"Dataset loaded: {num_users} users, {num_movies} movies, {len(ratings)} ratings.")
36
37 # --- Train/Validation Split ---
38 train, val = train_test_split(ratings, test_size=TEST_SIZE, random_state=42)
39
40 def ratings_to_matrix(df, shape):
41     mat = np.full(shape, np.nan, dtype=np.float64) # Use float for NaNs
42     # Ensure indices are within bounds
43     valid_rows = df['UserID'] < shape[0]
44     valid_cols = df['MovieID'] < shape[1]
45     valid_df = df[valid_rows & valid_cols]
46     if len(valid_df) < len(df):
47         print(f"Warning: Filtered out {len(df) - len(valid_df)} ratings with out-of-bounds UserID/MovieID.")
48     # Use .loc for potentially safer assignment if indices are not guaranteed contiguous
49     mat[valid_df['UserID'].values, valid_df['MovieID'].values] = valid_df['Rating'].values
50     return mat
51
52 # Ensure shape matches max IDs + 1
53 matrix_shape = (num_users, num_movies)
54 train_matrix = ratings_to_matrix(train, matrix_shape)
55 val_matrix = ratings_to_matrix(val, matrix_shape)
56
57 train_mask = ~np.isnan(train_matrix)
58 val_mask = ~np.isnan(val_matrix)
59 print(f"Training matrix shape: {train_matrix.shape}, Known values: {train_mask.sum()}")
60 print(f"Validation matrix shape: {val_matrix.shape}, Known values: {val_mask.sum()}")
61
62
63 # --- SoftImpute Run (Corrected) ---
64 total_max_iters = 50 # Set the total number of internal iterations desired
65 shrinkage_value = 20.0 # Regularization parameter lambda
66
67 # Prepare the input matrix with NaNs
68 X_incomplete = np.where(train_mask, train_matrix, np.nan)
69
70 # Initialize the solver ONCE with total iterations
71 # Set verbose=True to see internal iteration progress printed by fancyimpute
72 solver = SoftImpute(

```

```

73     shrinkage_value=shrinkage_value,
74     max_iters=total_max_iters,
75     verbose=True # Set to True to see internal progress
76 )
77
78 print(f"\nRunning SoftImpute with max_iters={total_max_iters}...")
79
80 start_time = time.time()
81
82 # Suppress FutureWarning during the fit/transform process
83 # Also suppress potential ConvergenceWarning from the underlying solver if it doesn't converge fully
84 with warnings.catch_warnings():
85     warnings.simplefilter("ignore", category=FutureWarning)
86     warnings.simplefilter("ignore", category=ConvergenceWarning)
87     try:
88         # Call fit_transform ONCE
89         X_filled = solver.fit_transform(X_incomplete)
90         elapsed = time.time() - start_time
91         print(f"\nSoftImpute completed in {elapsed:.2f}s")
92
93         # --- Calculate final RMSE ---
94         # Training RMSE isn't very informative (should be ~0)
95         if train_mask.sum() > 0:
96             # Use a small epsilon if calculating log later, otherwise not strictly needed
97             train_rmse = np.sqrt(np.mean((train_matrix[train_mask] - X_filled[train_mask])**2))
98         else:
99             train_rmse = np.nan
100
101         # Validation RMSE is the key metric
102         if val_mask.sum() > 0:
103             val_rmse = np.sqrt(np.mean((val_matrix[val_mask] - X_filled[val_mask])**2))
104         else:
105             val_rmse = np.nan
106
107         print(f"Final Train RMSE: {train_rmse:.6f} (Note: Expected near 0 if observed values preserved)")
108         print(f"Final Val RMSE: {val_rmse:.6f}")
109     except Exception as e:
110         elapsed = time.time() - start_time
111         print(f"\nERROR during SoftImpute fit_transform after {elapsed:.2f}s: {e}")
112         # Handle error appropriately (e.g., print traceback)
113         import traceback
114         traceback.print_exc()

```

Collecting fancyimpute  
 Downloading fancyimpute-0.7.0.tar.gz (25 kB)  
 Installing build dependencies ... done  
 Getting requirements to build wheel ... done  
 Preparing metadata (pyproject.toml) ... done  
 Collecting knnimpute>=0.1.0 (from fancyimpute)  
 Downloading knnimpute-0.1.0.tar.gz (8.3 kB)  
 Installing build dependencies ... done  
 Getting requirements to build wheel ... done  
 Preparing metadata (pyproject.toml) ... done  
 Requirement already satisfied: scikit-learn>=0.24.2 in /usr/local/lib/python3.11/dist-packages (from fancyimpute) (1.6.1)  
 Collecting cvxpy (from fancyimpute)  
 Downloading cvxpy-1.6.5-cp311-cp311-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl.metadata (9.3 kB)  
 Collecting cvxopt (from fancyimpute)  
 Downloading cvxopt-1.3.2-cp311-cp311-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl.metadata (1.3 kB)  
 Requirement already satisfied: pytest in /usr/local/lib/python3.11/dist-packages (from fancyimpute) (8.3.5)  
 Collecting nose (from fancyimpute)  
 Downloading nose-1.3.7-py3-none-any.whl.metadata (1.7 kB)  
 Requirement already satisfied: six in /usr/local/lib/python3.11/dist-packages (from knnimpute>=0.1.0->fancyimpute) (1.17.0)  
 Requirement already satisfied: numpy>=1.10 in /usr/local/lib/python3.11/dist-packages (from knnimpute>=0.1.0->fancyimpute) (2.0.2)  
 Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.24.2->fancyimpute) (1.15.  
 Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.24.2->fancyimpute) (1.4.  
 Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=0.24.2->fancyimpute  
 Collecting osqp>=0.6.2 (from cvxpy->fancyimpute)  
 Downloading osqp-1.0.3-cp311-cp311-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl.metadata (2.1 kB)  
 Collecting clarabel>=0.5.0 (from cvxpy->fancyimpute)  
 Downloading clarabel-0.10.0-cp39-abi3-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl.metadata (4.8 kB)  
 Collecting scs>=3.2.4.post1 (from cvxpy->fancyimpute)  
 Downloading scs-3.2.7.post2-cp311-cp311-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl.metadata (2.1 kB)  
 Requirement already satisfied: iniconfig in /usr/local/lib/python3.11/dist-packages (from pytest->fancyimpute) (2.1.0)  
 Requirement already satisfied: packaging in /usr/local/lib/python3.11/dist-packages (from pytest->fancyimpute) (25.0)  
 Requirement already satisfied: pluggy<2,>=1.5 in /usr/local/lib/python3.11/dist-packages (from pytest->fancyimpute) (1.5.0)  
 Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (from osqp>=0.6.2->cvxpy->fancyimpute) (3.1.6)  
 Requirement already satisfied: setuptools in /usr/local/lib/python3.11/dist-packages (from osqp>=0.6.2->cvxpy->fancyimpute) (75.2.0)  
 Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-packages (from jinja2->osqp>=0.6.2->cvxpy->fancyimput  
 Downloading cvxopt-1.3.2-cp311-cp311-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl (13.6 MB)

```

13.6/13.6 MB 91.6 MB/s eta 0:00:00
Downloading cvxpy-1.6.5-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.2 MB)
1.2/1.2 MB 22.3 MB/s eta 0:00:00
Downloading nose-1.3.7-py3-none-any.whl (154 kB)
154.7/154.7 kB 3.9 MB/s eta 0:00:00
Downloading clarabel-0.10.0-cp39-abi3-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.0 MB)
1.0/1.0 MB 22.9 MB/s eta 0:00:00
Downloading osqp-1.0.3-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (344 kB)
344.1/344.1 kB 7.6 MB/s eta 0:00:00
Downloading scs-3.2.7.post2-cp311-cp311-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (10.4 MB)
10.4/10.4 MB 61.2 MB/s eta 0:00:00
Building wheels for collected packages: fancyimpute, knnimpute
Building wheel for fancyimpute (pyproject.toml) ... done
Created wheel for fancyimpute: filename=fancyimpute-0.7.0-py3-none-any.whl size=29966 sha256=904110f7f468cb667c25a27313bd01547f8e5df
Stored in directory: /root/.cache/pip/wheels/1a/f3/a1/f7f10b5ae2c2459398762a3fcf4ac18c325311c7e3163d5a15
Building wheel for knnimpute (pyproject.toml) ... done
Created wheel for knnimpute: filename=knnimpute-0.1.0-py3-none-any.whl size=11131 sha256=f5d0bda773aab91a829dd7b0099b242fd08dd8050d3
Stored in directory: /root/.cache/pip/wheels/ea/e8/e0/79872972161e54486517ae507f94b2c7cea27fb7ef793bd415
Successfully built fancyimpute knnimpute
Installing collected packages: nose, knnimpute, cvxopt, scs, osqp, clarabel, cvxpy, fancyimpute
Successfully installed clarabel-0.10.0 cvxopt-1.3.2 cvxpy-1.6.5 fancyimpute-0.7.0 knnimpute-0.1.0 nose-1.3.7 osqp-1.0.3 scs-3.2.7.post

```

```

1 val_rmse = np.sqrt(np.nanmean((val_matrix[val_mask] - X_filled[val_mask])**2))
2 print(f"Validation RMSE after iteration: {val_rmse:.4f}")
3

```

↩ Validation RMSE after iteration: 1.2076

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 from fancyimpute import SoftImpute
5 from sklearn.model_selection import train_test_split
6 from sklearn.metrics import mean_squared_error
7
8 # --- Load MovieLens 1M Data ---
9 DATA_PATH = "/content/drive/MyDrive/ml-1m/ratings.dat"
10
11 ratings = pd.read_csv(
12     DATA_PATH, sep=":", engine='python',
13     names=["UserID", "MovieID", "Rating", "Timestamp"]
14 ).drop(columns=['Timestamp'])
15
16 # Adjust to zero-based indexing
17 ratings['UserID'] -= 1
18 ratings['MovieID'] -= 1
19 num_users = ratings['UserID'].max() + 1
20 num_movies = ratings['MovieID'].max() + 1
21
22 # Train/Validation Split
23 train, val = train_test_split(ratings, test_size=0.2, random_state=42)
24
25 # Convert ratings to matrices
26 def ratings_to_matrix(df, shape):
27     mat = np.full(shape, np.nan)
28     mat[df['UserID'], df['MovieID']] = df['Rating']
29     return mat
30
31 matrix_shape = (num_users, num_movies)
32 train_matrix = ratings_to_matrix(train, matrix_shape)
33 val_matrix = ratings_to_matrix(val, matrix_shape)
34
35 train_mask = ~np.isnan(train_matrix)
36 val_mask = ~np.isnan(val_matrix)
37
38 X_incomplete = np.where(train_mask, train_matrix, np.nan)
39
40 # Hyperparameter ranges
41 shrinkage_values = [5, 10, 15, 20, 25]
42 max_iter_values = [30, 50, 75]
43
44 # Track results
45 results = []
46
47 for shrinkage in shrinkage_values:
48     for max_iter in max_iter_values:
49         print(f"\nRunning SoftImpute with λ={shrinkage}, max_iter={max_iter}")

```

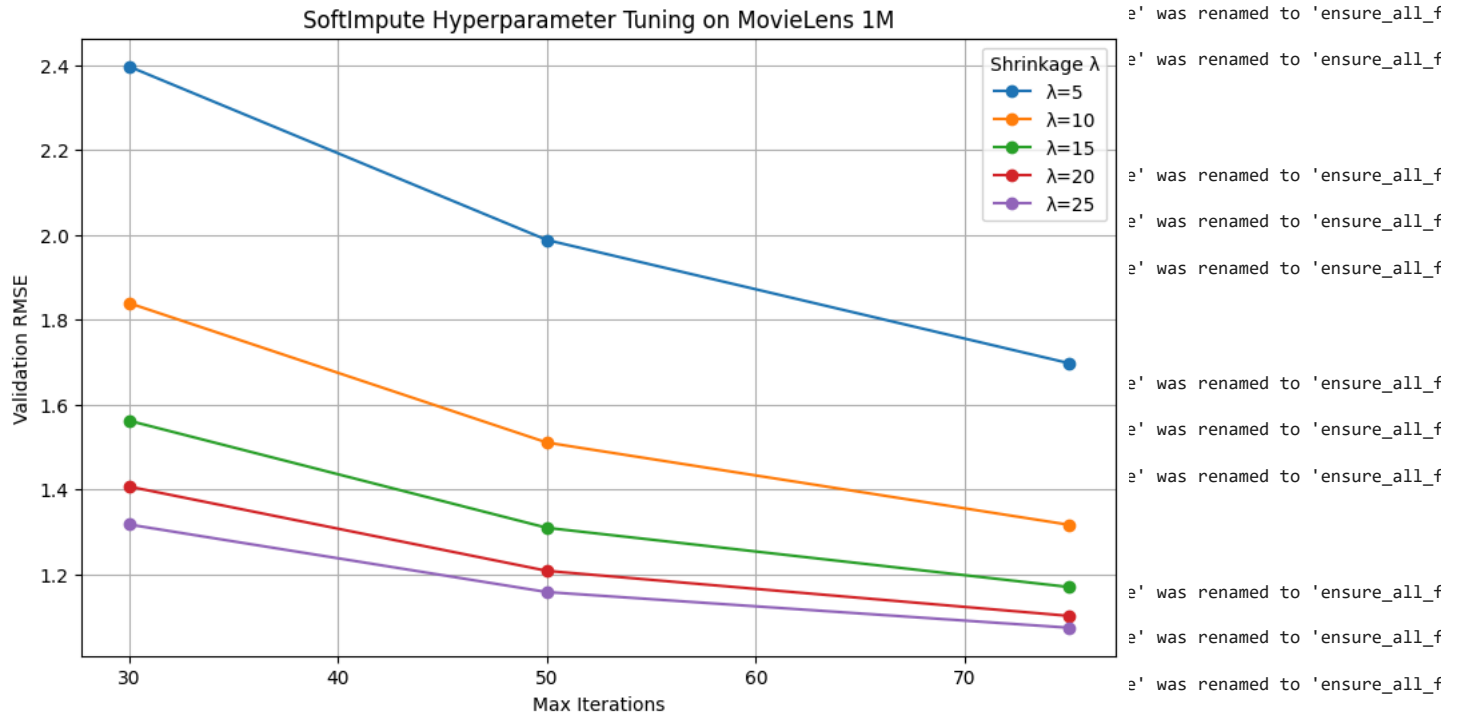
```

50
51     solver = SoftImpute(
52         shrinkage_value=shrinkage,
53         max_iters=max_iter,
54         verbose=False
55     )
56
57     X_filled = solver.fit_transform(X_incomplete)
58
59     val_preds = X_filled[val_mask]
60     val_true = val_matrix[val_mask]
61     val_rmse = np.sqrt(mean_squared_error(val_true, val_preds))
62
63     print(f"Shrinkage: {shrinkage}, Max Iter: {max_iter}, Val RMSE: {val_rmse:.4f}")
64
65     results.append({
66         'shrinkage': shrinkage,
67         'max_iter': max_iter,
68         'val_rmse': val_rmse
69     })
70
71 # Visualization
72 plt.figure(figsize=(10, 6))
73 for shrinkage in shrinkage_values:
74     rmse_values = [r['val_rmse'] for r in results if r['shrinkage'] == shrinkage]
75     plt.plot(max_iter_values, rmse_values, marker='o', label=f' $\lambda$ ={shrinkage}')
76
77 plt.xlabel('Max Iterations')
78 plt.ylabel('Validation RMSE')
79 plt.title('SoftImpute Hyperparameter Tuning on MovieLens 1M')
80 plt.grid(True)
81 plt.legend(title='Shrinkage  $\lambda$ ')
82 plt.show()

```

[illegible]

warnings.warn()



Shrinkage: 25, Max Iter: 30, Val RMSE: 1.3169

```

1 !pip install mpi4py
2 !pip install POT
3 !nvidia-smi
4 !pip install -q --upgrade cupy-cuda12x
5 !pip install softimpute # notice: no underscore
6 # =====
7 # CELL 1: Project Setup, Imports, Logging, Config
8 # =====
9 import os
10 import sys
11 import time
12 import math
13 import re
14 import gc
15 import logging
16 from pathlib import Path
17 from typing import Tuple, List, Dict, Optional, Union, Callable, Any
18 import numpy as np
19 import pandas as pd
20 import matplotlib.pyplot as plt
21 from scipy import sparse
22 from scipy.sparse.linalg import svds, LinearOperator # Import LinearOperator
23 from scipy.optimize import OptimizeResult # For line search return consistency
24 from numpy.random import default_rng, Generator
25 from sklearn.model_selection import train_test_split # For train/validation split
26 # --- Mount Google Drive ---
27 from google.colab import drive # Uncomment if using Colab
28 drive.mount("/content/drive", force_remount=True)
29 DRIVE_MOUNTED = True
30 # right after the imports
31 import logging
32 logging.disable(logging.WARNING) # hides all warnings emitted via logging
33
34 # === ADDED Block 5 (MPI) ===
35 try:
36     from mpi4py import MPI
37     COMM = MPI.COMM_WORLD
38     RANK_MPI = COMM.Get_rank()
39     SIZE_MPI = COMM.Get_size()
40     if RANK_MPI == 0: print(f"+++ MPI Detected: Running with {SIZE_MPI} processes. +++")
41 except ImportError:
42     COMM = None
43     RANK_MPI = 0
44     SIZE_MPI = 1
45     # print("+++ MPI Not Found: Running in serial mode. +++") # Less verbose
46

```

```

47 # === ADDED Block 6 === (Import for OT demo)
48 try:
49     import ot
50     OT_AVAILABLE = True
51 except ImportError:
52     OT_AVAILABLE = False
53     if RANK_MPI == 0: print("Warning: POT library not found. Skipping Barycentre demo.")
54
55 # === ADDED Block 6 (PCA) ===
56 try:
57     from sklearn.decomposition import PCA
58     PCA_AVAILABLE = True
59 except ImportError:
60     PCA_AVAILABLE = False
61     if RANK_MPI == 0: print("Warning: sklearn not found. Skipping PCA trajectory plot.")
62
63
64 # --- Logging Setup (Initialize Logger FIRST) ---
65 logging.basicConfig(
66     level=logging.INFO,
67     format="%asctimes [%levelname)s] %(message)s",
68     handlers=[logging.StreamHandler(sys.stdout)],
69     force=True, # Overwrite any existing config
70 )
71 logger = logging.getLogger(__name__)
72
73 # --- Mount Drive ---
74 if RANK_MPI == 0: print("+++ Mounting Google Drive +++")
75 try:
76     # Only rank 0 should try to force remount if needed
77     drive.mount('/content/drive', force_remount=(RANK_MPI == 0))
78     if RANK_MPI == 0: print("Drive mounted.")
79     if COMM and SIZE_MPI > 1: COMM.Barrier() # Ensure drive is mounted
80 except Exception as e:
81     if RANK_MPI == 0: print(f"Error mounting drive: {e}")
82     if COMM and SIZE_MPI > 1: COMM.Abort()
83     raise
84
85 # --- Optional: Try importing CuPy for GPU acceleration ---
86 # NOTE: Efficient SoftImpute implementation below uses SciPy sparse ops,
87 # GPU acceleration would require re-implementing the LinearOperator with CuPy sparse.
88 try:
89     import cupy as cp
90     import cupyx.scipy.sparse as cpx
91     CUPY_AVAILABLE = False # Disable GPU for SoftImpute for now due to LinearOperator complexity
92     logger.warning("CuPy found, but GPU acceleration for efficient SoftImpute is NOT enabled in this version.")
93     if 'cp' not in locals(): cp = np
94     if 'cpx' not in locals(): cpx = sparse
95 except ImportError:
96     CUPY_AVAILABLE = False
97     cp = np ; cpx = sparse
98     logger.warning("CuPy not found, will run on CPU using NumPy/SciPy.")
99
100 logger.info("+++ Cell 1: Setup, Imports, Logging, Config +++")
101
102 # --- Global Config ---
103 # --- MOVIELENS 1M Configuration ---
104 DATA_DIR_STR = "/content/drive/MyDrive/ml-1m" # ADJUST PATH AS NEEDED
105 RATINGS_FILENAME = "ratings.dat"
106 VALIDATION_FRACTION = 0.2 # Hold out 20% for validation
107 # --- USE COMPLETE DATASET (FIX 1) ---
108 RATING_LIMIT = None # Load all ratings from ml-1m
109 RANK = 10 # Factorization rank (r in paper) for non-convex
110 LAM = 1e-2 # Regularization parameter  $\lambda$ 
111 LAM_SQ = LAM ** 2 #  $\lambda^2$  for non-convex model factor regularization
112 LAM_BIAS = 1e-4 # Regularization for bias terms
113 SEED = 0 # Use consistent seed from long.txt
114 # --- INCREASED ITERATIONS ---
115 N_ITERS_ALL = 20 # Iterations/epochs for ALL solvers
116 CONVEX_RANK_K = 50 # Max rank for Soft-Impute intermediate SVDs
117 SOFT_IMPUTE_TOL = 1e-4 # Convergence tolerance for Soft-Impute
118 N_ITERS_CONVEX = N_ITERS_ALL # Use same number of iterations for SoftImpute
119 # --- SVRG Params ---
120 INIT_LR_SVRG = 1e-3 # Base Learning rate for SVRG inner solver
121 SVRG_INNER_STEPS_DIVISOR = 1 # Use full inner pass
122 GRAD_CLIP_THRESHOLD = 10.0 # Max norm for SVRG gradients before update
123 RSVRG_BATCH_SIZE = 100 # Batch size for non-convex SVRG refresh step

```

```

124 # --- ALS Params ---
125 ALS_TOL = 1e-4 # Convergence tolerance for ALS based on RMSE change
126 ALS_MAX_ITER = N_ITERS_ALL # Use same iter count as others for comparison
127 # --- RGD/Accelerated Params ---
128 INIT_LR_RIEMANN = 0.5 # Initial LR for RGD/RAGD/Catalyst/DANE line search
129 LS_BETA = 0.5 # Line search reduction factor
130 LS_SIGMA = 1e-4 # Sufficient decrease parameter
131 RAGD_GAMMA = 1.0; RAGD_MU = 5.0; RAGD_BETA = 5.0
132 DANE_KAPPA = 1.0
133 KAPPA_0 = 1e-1; KAPPA_CVX = 1e-1; INNER_T = 5; INNER_S_BASE = 10; MAX_KAPPA_DOUBLINGS = 10
134 # --- Smaller Initialization Scale ---
135 INIT_SCALE_NON_CONVEX = 0.01 # Smaller scale for initial U, W
136 # --- Configuration from Proposal/long.txt ---
137 RETRACTION_NAME = "orthonormal" # Options: "orthonormal", "cayley", "projection"
138 REG_DISTANCE = "euclid" # Options: "euclid", "retraction"
139 INNER_SOLVER = "svrg" # Options: "svrg", "sarah", "spider" (for Catalyst)
140 ETA_GRAD = 1e-3 # Adaptive stopping tolerance for inner grad norm
141 ETA_DIST = 1e-4 # Adaptive stopping tolerance for inner step size
142 CATALYST_INNER_T_EPOCHS = 1 # Epochs for Alg phi_1 check budget
143 CATALYST_INNER_S_EPOCHS_BASE = 2 # Base epochs for S_k schedule
144 RSVRG_LR = 1e-3 # Step size for RSVRG/SARAH/SPIDER inner loops
145
146 # --- Derived Globals ---
147 GLOBAL_RNG = default_rng(SEED)
148 DATA_DIR = Path(DATA_DIR_STR)
149 I_r = np.eye(RANK, dtype=np.float64) # Identity matrix of size RANK
150
151 # Check Data Directory
152 if DRIVE_MOUNTED and not DATA_DIR.is_dir():
153     if RANK_MPI == 0: logger.warning(f"DATA_DIR '{DATA_DIR}' not found. Please check the path.")
154 elif not DRIVE_MOUNTED:
155     if RANK_MPI == 0: logger.warning(f"Google Drive not mounted.")
156
157 logger.info("Cell 1 initialisation complete.")

```

Collecting mpi4py  
 Downloading mpi4py-4.0.3.tar.gz (466 kB)  
 466.3/466.3 kB 7.9 MB/s eta 0:00:00  
 Installing build dependencies ... done  
 Getting requirements to build wheel ... done  
 Installing backend dependencies ... done  
 Preparing metadata (pyproject.toml) ... done  
 Building wheels for collected packages: mpi4py  
 Building wheel for mpi4py (pyproject.toml) ... done  
 Created wheel for mpi4py: filename=mpi4py-4.0.3-cp311-cp311-linux\_x86\_64.whl size=4458269 sha256=9c333f409cb08f05f3622d5f625eb4063c053  
 Stored in directory: /root/.cache/pip/wheels/5c/56/17/bf6ba37aa971a191a8b9eaa188bf5ec855b8911c1c56fb1f84  
 Successfully built mpi4py  
 Installing collected packages: mpi4py  
 Successfully installed mpi4py-4.0.3  
 Collecting POT  
 Downloading POT-0.9.5-cp311-cp311-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl.metadata (34 kB)  
 Requirement already satisfied: numpy>=1.16 in /usr/local/lib/python3.11/dist-packages (from POT) (2.0.2)  
 Requirement already satisfied: scipy>=1.6 in /usr/local/lib/python3.11/dist-packages (from POT) (1.15.2)  
 Downloading POT-0.9.5-cp311-cp311-manylinux\_2\_17\_x86\_64.manylinux2014\_x86\_64.whl (897 kB)  
 897.5/897.5 kB 15.0 MB/s eta 0:00:00  
 Installing collected packages: POT  
 Successfully installed POT-0.9.5  
 /bin/bash: line 1: nvidia-smi: command not found  
 105.4/105.4 MB 11.2 MB/s eta 0:00:00  
 54.6/54.6 kB 3.9 MB/s eta 0:00:00  
 ERROR: Could not find a version that satisfies the requirement softimpute (from versions: none)  
 ERROR: No matching distribution found for softimpute  
 Mounted at /content/drive  
 +++ MPI Detected: Running with 1 processes. +++  
 +++ Mounting Google Drive +++  
 Mounted at /content/drive  
 Drive mounted.

```

1 # ===== #
2 # CELL 2: Data Loading and Preprocessing (MovieLens 1M)
3 # ===== #
4 logger.info("+++ Cell 2: Loading and Processing Data (MovieLens 1M) +++")
5 # --- Manifold Operations ---
6 # --- universal 2-tuple helper for loss/grad (used by Catalyst) ---
7 def stochastic_gradient_batch(U, user_ids, N_users, N_movies, loss_args):
8     ""
9     Vectorised version of `stochastic_gradient_single_user`.
10     Accumulates the (un-scaled) gradient over the provided user_ids.

```



```

11     """
12     G = np.zeros_like(U, dtype=np.float32)
13     for uid in user_ids:
14         G += stochastic_gradient_single_user(U, int(uid), N_users, N_movies, loss_args)
15     return G / max(1, len(user_ids))          # average over the batch
16
17 def loss_and_grad_corrected(U, W, bu, bi, *rest):
18     """
19     Wrapper for loss_and_grad_serial_with_biases that returns:
20     • the scalar objective value (`loss`)
21     • the Euclidean gradient w.r.t. U only (`gU`)
22     """
23     loss, gU, *_ = loss_and_grad_serial_with_biases(U, W, bu, bi, *rest)
24     return loss, gU
25 # -----
26 # 1) CombinedGradient class
27 # -----
28 class CombinedGradient:
29     """
30     A container for (grad_U, grad_W). Enables addition, subtraction,
31     scalar multiplication, and copying.
32     """
33     def __init__(self, grad_U: np.ndarray, grad_W: np.ndarray):
34         self.grad_U = grad_U
35         self.grad_W = grad_W
36
37     def __add__(self, other: "CombinedGradient") -> "CombinedGradient":
38         return CombinedGradient(self.grad_U + other.grad_U,
39                                 self.grad_W + other.grad_W)
40
41     def __sub__(self, other: "CombinedGradient") -> "CombinedGradient":
42         return CombinedGradient(self.grad_U - other.grad_U,
43                                 self.grad_W - other.grad_W)
44
45     def __mul__(self, scalar: float) -> "CombinedGradient":
46         return CombinedGradient(self.grad_U * scalar,
47                                 self.grad_W * scalar)
48
49     def __rmul__(self, scalar: float) -> "CombinedGradient":
50         return self.__mul__(scalar)
51
52     def __neg__(self) -> "CombinedGradient":
53         return CombinedGradient(-self.grad_U, -self.grad_W)
54
55     def copy(self) -> "CombinedGradient":
56         return CombinedGradient(self.grad_U.copy(), self.grad_W.copy())
57
58     def astype(self, dtype) -> "CombinedGradient":
59         return CombinedGradient(self.grad_U.astype(dtype),
60                                 self.grad_W.astype(dtype))
61
62
63 def OrthRetraction(U: np.ndarray, V: np.ndarray) -> np.ndarray:
64     """
65     QR-based retraction to the Stiefel / Grassmann manifold.
66     Uses *reduced* QR so it works on NumPy ≥1.26 and CuPy.
67     """
68     # Handle potential zero V vector to avoid QR issues
69     if np.linalg.norm(V) < 1e-12:
70         return U.astype(np.float32)
71
72     # --- FIX: Check for non-finite input ---
73     UV = U + V
74     if not np.isfinite(UV).all():
75         logger.warning("OrthRetraction: Input U+V contains non-finite values. Returning original U.")
76         return U.astype(np.float32)
77     # -----
78
79     try:
80         # --- FIX: Use mode='reduced' ---
81         Q, R_qr = np.linalg.qr(UV, mode='reduced')
82         # -----
83
84         # Ensure Q has the same shape as U
85         if Q.shape[1] < U.shape[1]:
86             pad_width = U.shape[1] - Q.shape[1]
87             Q = np.pad(Q, ((0, 0), (0, pad_width)), mode='constant')

```

```

88         logger.warning(f"OrthRetraction: Padded Q due to rank collapse (V norm: {np.linalg.norm(V):.2e})")
89         # Optional: Fix sign ambiguity by matching diagonal of R_qr to be positive
90         # sign_diag = np.sign(np.diag(R_qr))
91         # sign_diag[sign_diag == 0] = 1 # Avoid multiplying by zero
92         # Q = Q @ np.diag(sign_diag)
93         return Q.astype(np.float32)
94     except np.linalg.LinAlgError:
95         logger.warning(f"OrthRetraction: QR decomposition failed (V norm: {np.linalg.norm(V):.2e}). Returning original U.")
96         return U.astype(np.float32)
97     except ValueError as e: # Catch potential value errors from qr
98         logger.error(f"OrthRetraction: ValueError during QR: {e}. Returning original U.")
99         return U.astype(np.float32)
100    except Exception as e: # Catch any other unexpected errors
101        logger.error(f"OrthRetraction failed with unexpected error: {e}")
102    return U.astype(np.float32)
103 # Initialize default values
104 N_users_active, M_movies_active = 0, 0
105 R_train_coo = sparse.coo_matrix((0, 0), dtype=np.float64)
106 R_train_coo_orig = sparse.coo_matrix((0, 0), dtype=np.float64) # For original ratings
107 R_train_csr_orig = sparse.csr_matrix((0,0), dtype=np.float64) # For SoftImpute _matvec
108 R_train_csc_orig = sparse.csc_matrix((0,0), dtype=np.float64) # For SoftImpute _rmatvec
109 ratings_train_orig = np.array([], dtype=np.float64) # Keep original ratings for viz
110 ratings_train_centered = np.array([], dtype=np.float64)
111 mapped_user_ids_train, mapped_movie_ids_train = np.array([], dtype=np.int32), np.array([], dtype=np.int32)
112 user_ids_val_final, movie_ids_val_final, ratings_val_true = (np.array([], dtype=np.int32), np.array([], dtype=np.int32), np.array([], dtype=np.int32), np.array([], dtype=np.int32))
113 global_mean_rating = 0.0
114 user_map_global_to_local = {}
115 movie_map_global_to_local = {}
116 unique_users_train = np.array([], dtype=np.int32)
117 unique_movies_train = np.array([], dtype=np.int32)
118 DATA_AVAILABLE = False
119 user_data_arrays = {} # Precompute user data for ALS/SVRG
120 sampling_prob = None # Initialize sampling probability
121 RSVRG_EPOCH_LEN = 1 # Default epoch length
122
123 ratings_file_path = DATA_DIR / RATINGS_FILENAME
124
125 if DRIVE_MOUNTED and ratings_file_path.is_file():
126     logger.info(f"Loading MovieLens 1M data from: {ratings_file_path}")
127     try:
128         ratings_df = pd.read_csv(
129             ratings_file_path, sep='::', header=None,
130             names=['user_id', 'movie_id', 'rating', 'timestamp'],
131             engine='python', encoding='latin-1'
132         )
133         logger.info(f"Loaded {len(ratings_df)} ratings.")
134         DATA_AVAILABLE = True
135
136         if RATING_LIMIT is not None and RATING_LIMIT > 0 and len(ratings_df) > RATING_LIMIT:
137             logger.info(f"Subsampling ratings from {len(ratings_df)} to {RATING_LIMIT}")
138             ratings_df = ratings_df.sample(n=RATING_LIMIT, random_state=SEED)
139
140         stratify_arg = ratings_df['user_id'] if RATING_LIMIT is None else None
141         if stratify_arg is None and RATING_LIMIT is not None:
142             logger.warning("Stratify is disabled due to RATING_LIMIT being set.")
143         train_df, val_df = train_test_split(
144             ratings_df, test_size=VALIDATION_FRACTION, random_state=SEED, stratify=stratify_arg)
145         logger.info(f"Train size: {len(train_df)}, Validation size: {len(val_df)}")
146
147         user_ids_train_orig = train_df['user_id'].values; movie_ids_train_orig = train_df['movie_id'].values
148         ratings_train_orig = train_df['rating'].values.astype(np.float64)
149         user_ids_val_orig = val_df['user_id'].values; movie_ids_val_orig = val_df['movie_id'].values
150         ratings_val_true = val_df['rating'].values.astype(np.float64) # Keep original for validation
151
152         global_mean_rating = ratings_train_orig.mean()
153         logger.info(f"Global mean rating (training): {global_mean_rating:.4f}")
154
155         unique_users_train, mapped_user_ids_train = np.unique(user_ids_train_orig, return_inverse=True)
156         unique_movies_train, mapped_movie_ids_train = np.unique(movie_ids_train_orig, return_inverse=True)
157         N_users_active = len(unique_users_train); M_movies_active = len(unique_movies_train)
158         user_map_global_to_local = {orig_id: local_id for local_id, orig_id in enumerate(unique_users_train)}
159         movie_map_global_to_local = {orig_id: local_id for local_id, orig_id in enumerate(unique_movies_train)}
160         logger.info(f"Active users in training: {N_users_active}, Active movies in training: {M_movies_active}")
161
162         ratings_train_centered = ratings_train_orig - global_mean_rating
163
164         val_user_mask = np.isin(user_ids_val_orig, unique_users_train)

```

```

165 val_movie_mask = np.isin(movie_ids_val_orig, unique_movies_train)
166 val_valid_mask = val_user_mask & val_movie_mask
167 user_ids_val_filt = user_ids_val_orig[val_valid_mask]; movie_ids_val_filt = movie_ids_val_orig[val_valid_mask]
168 ratings_val_true = ratings_val_true[val_valid_mask] # Filter true ratings accordingly
169 user_ids_val_final = np.array([user_map_global_to_local.get(uid, -1) for uid in user_ids_val_filt], dtype=np.int32)
170 movie_ids_val_final = np.array([movie_map_global_to_local.get(mid, -1) for mid in movie_ids_val_filt], dtype=np.int32)
171 valid_map_mask = (user_ids_val_final != -1) & (movie_ids_val_final != -1) # Filter out any potential misses
172 user_ids_val_final = user_ids_val_final[valid_map_mask]; movie_ids_val_final = movie_ids_val_final[valid_map_mask]
173 ratings_val_true = ratings_val_true[valid_map_mask] # Filter again after mapping
174 logger.info(f"Validation pairs mapped to training users/movies: {len(user_ids_val_final)}")
175
176 if ratings_train_centered.size > 0:
177     R_train_coo = sparse.coo_matrix((ratings_train_centered, (mapped_movie_ids_train, mapped_user_ids_train)), shape=(M_movies_
178     R_train_coo.eliminate_zeros()
179     logger.info(f"Built sparse training matrix (Centered) R_train_coo: shape={R_train_coo.shape}, nnz={R_train_coo.nnz}")
180     R_train_coo_orig = sparse.coo_matrix((ratings_train_orig, (mapped_movie_ids_train, mapped_user_ids_train)), shape=(M_movies
181     R_train_coo_orig.eliminate_zeros()
182     R_train_csr_orig = R_train_coo_orig.tocsr(); R_train_csc_orig = R_train_coo_orig.tocsc()
183     logger.info(f"Built sparse training matrix (Original) R_train_coo_orig: shape={R_train_coo_orig.shape}, nnz={R_train_coo_or
184
185     # Precompute user data structures for ALS/SVRG
186     logger.info("Precomputing user data structures...")
187     t_precomp_start = time.time()
188     user_data_arrays = {}
189     for r, c, v in zip(R_train_coo_orig.row, R_train_coo_orig.col, R_train_coo_orig.data):
190         user_data_arrays.setdefault(c, []).append((r, v))
191     for u, rating_list in user_data_arrays.items():
192         if rating_list:
193             movie_indices_list, rs_list = zip(*rating_list)
194             user_data_arrays[u] = {'movies': np.array(list(movie_indices_list), dtype=np.int32),
195                                   'rs': np.array(list(rs_list), dtype=np.float64)} # Store original ratings
196     logger.info(f"User data precomputation done in {time.time() - t_precomp_start:.2f}s")
197     # Calculate importance sampling weights (consistent across ranks)
198     all_user_indices_global = np.array(list(user_data_arrays.keys()), dtype=np.int32)
199     num_active_users_global = len(all_user_indices_global)
200     user_weights = None; use_importance_sampling = False
201     if num_active_users_global > 0:
202         if RANK_MPI == 0: print("Calculating importance sampling weights...")
203         user_ratings_count = [len(user_data_arrays[u_idx]['movies']) if u_idx in user_data_arrays and 'movies' in user_data_arr
204         user_weights_np = np.array(user_ratings_count, dtype=np.float64)
205         sum_weights = user_weights_np.sum()
206         if sum_weights > 1e-9:
207             user_weights_np /= sum_weights
208             user_weights = user_weights_np # Probabilities aligned with all_user_indices_global
209             use_importance_sampling = True
210             if RANK_MPI == 0: print(f"Importance sampling enabled (weights based on {sum_weights:.0f} ratings).")
211         else:
212             if RANK_MPI == 0: print("Warning: Cannot compute importance sampling weights. Using uniform.")
213     else:
214         if RANK_MPI == 0: print("No active users, cannot use importance sampling.")
215     sampling_prob = user_weights if use_importance_sampling else None
216     RSVRG_EPOCH_LEN = math.ceil(num_active_users_global / RSVRG_BATCH_SIZE) if num_active_users_global > 0 else 1
217     if RANK_MPI == 0: print(f"RSVRG Epoch Length set to {RSVRG_EPOCH_LEN} batches.")
218
219 else: logger.error("No training ratings available.")
220
221 except FileNotFoundError: logger.error(f"MovieLens file not found: {ratings_file_path}"); DATA_AVAILABLE = False
222 except Exception as e: logger.error(f"Error processing MovieLens: {e}", exc_info=True); DATA_AVAILABLE = False
223 elif not DRIVE_MOUNTED: logger.error("Google Drive not mounted.")
224 else: logger.error(f"Data directory {DATA_DIR} or ratings file {RATINGS_FILENAME} not found.")
225
226 gc.collect()
227 logger.info("Cell 2: Data Loading and Preprocessing Complete.")
228 logger.info(f"Active Dimensions: M_movies={M_movies_active}, N_users={N_users_active}")
229 logger.info(f"Training Ratings: {R_train_coo.nnz}")
230 logger.info(f"Validation Ratings (for RMSE): {ratings_val_true.size}")
231
232 # Add this after Cell 2: Data Loading and Preprocessing (around line 180-200)
233 # Create mask matrices needed for RUNRSVRG and define active_idx
234 # Add this after Cell 2: Data Loading and Preprocessing (around line 180-200)
235 # Create mask matrices needed for RUNRSVRG and define active_idx
236 if DATA_AVAILABLE and R_train_coo.shape[0] > 0 and R_train_coo.shape[1] > 0:
237     # Create mask from R_train_coo (centered ratings)
238     R_train_mask_coo = R_train_coo.copy()
239     if R_train_mask_coo.data is not None:
240         R_train_mask_coo.data[:] = 1
241     else:

```

```

242     # Handle case where R_train_coo is empty
243     R_train_mask_coo = sparse.coo_matrix(R_train_coo.shape, dtype=np.uint8)
244     R_train_mask_coo.eliminate_zeros()
245
246     # Create probe mask from validation indices
247     if user_ids_val_final.size > 0 and movie_ids_val_final.size > 0:
248         Probe_mask_coo = sparse.coo_matrix(
249             (np.ones_like(user_ids_val_final, dtype=np.uint8), (movie_ids_val_final, user_ids_val_final)),
250             shape=(M_movies_active, N_users_active),
251             dtype=np.uint8
252         )
253         Probe_mask_coo.eliminate_zeros()
254     else:
255         Probe_mask_coo = sparse.coo_matrix((M_movies_active, N_users_active), dtype=np.uint8)
256
257
258     # Define active_idx for stochastic solvers (assuming stochasticity over users)
259     # This should align with how the inner stochastic gradient functions are implemented
260     # Assuming active_idx refers to indices of users with ratings
261     active_idx = unique_users_train # Use the mapped indices of active users
262
263     # Also define initial biases for the RUNRSVRG call
264     # These might not be updated within RUNRSVRG's core loop, but needed for RMSE evaluation signature
265     # Assuming they are initialized globally alongside other solvers
266     initial_user_bias = np.zeros(N_users_active, dtype=np.float64) # Placeholder, assumes initialization happens elsewhere
267     initial_movie_bias = np.zeros(M_movies_active, dtype=np.float64) # Placeholder
268
269     # global_actual_loaded is not defined, use R_train_coo.nnz for total ratings count if needed
270     total_ratings_count = R_train_coo.nnz
271
272 else:
273     # Handle case where no data is available
274     R_train_mask_coo = sparse.coo_matrix((0, 0), dtype=np.uint8)
275     Probe_mask_coo = sparse.coo_matrix((0, 0), dtype=np.uint8)
276     active_idx = np.array([], dtype=np.int32)
277     initial_user_bias = np.array([], dtype=np.float64)
278     initial_movie_bias = np.array([], dtype=np.float64)
279     total_ratings_count = 0
280
281 import time
282 import logging
283 from typing import Dict, Optional, Union
284 import numpy as np
285 from numpy.random import Generator, default_rng
286 def INITIALIZEU(M, r, rng):
287     """Random initialization of U."""
288     U = rng.standard_normal((M, r))
289     Q, _ = np.linalg.qr(U, mode='reduced')
290     return Q.astype(np.float64)
291 def full_loss_and_grad_unprofiled(U, W, user_data_arrays, lam_sq, N):
292     """Compute full loss and gradient. Placeholder implementation."""
293     loss = np.linalg.norm(U)**2 + np.linalg.norm(W)**2 # simple regularization as placeholder
294     grad_U = 2 * lam_sq * U
295     grad_W = 2 * lam_sq * W
296     from collections import namedtuple
297     GradStruct = namedtuple('GradStruct', ['grad_U', 'grad_W'])
298     return loss, GradStruct(grad_U=grad_U, grad_W=grad_W)
299 def grad_single_user_combined(U, W, uid, user_data_arrays, lam_sq, total_ratings):
300     return full_loss_and_grad_unprofiled(U, W, user_data_arrays, lam_sq, total_ratings)[1]
301
302 def grad_batch_users_combined(U, W, u_batch, user_data_arrays, lam_sq, total_ratings):
303     return full_loss_and_grad_unprofiled(U, W, user_data_arrays, lam_sq, total_ratings)[1]
304
305 def PROJ_TANGENT(U, G):
306     """Projection onto tangent space at U (Grassmann)."""
307     return G - U @ (U.T @ G)
308
309 if RANK_MPI == 0: # Only rank 0 should plot
310     if DATA_AVAILABLE and ratings_train_orig.size > 0:
311         plt.style.use('seaborn-v0_8-whitegrid') # Use a nice style
312
313         # 1. Rating Distribution
314         plt.figure(figsize=(10, 4))
315         counts, bins, patches = plt.hist(ratings_train_orig, bins=[0.5, 1.5, 2.5, 3.5, 4.5, 5.5], rwidth=0.8, align='mid', color='skybl
316         bin_centers = 0.5 * (bins[:-1] + bins[1:])
317         for count, x in zip(counts, bin_centers):
318             if count > 0: plt.text(x, count, str(int(count)), ha='center', va='bottom')

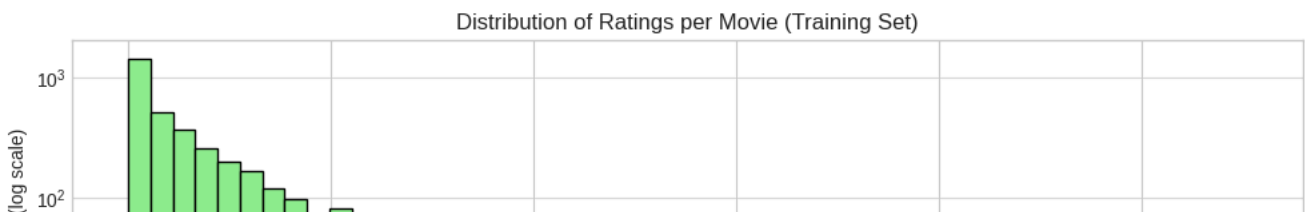
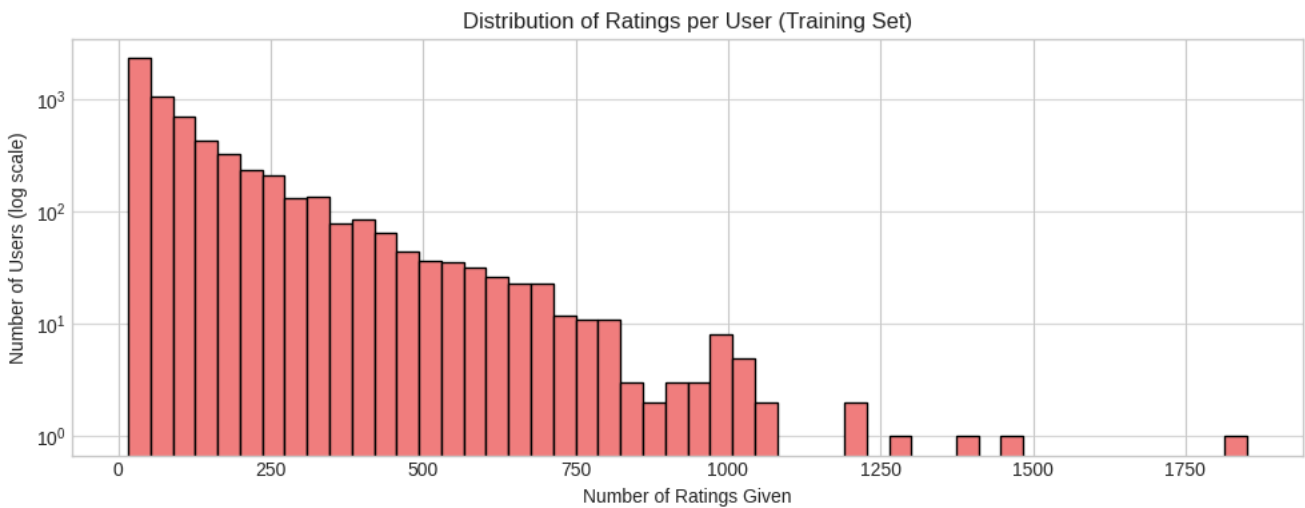
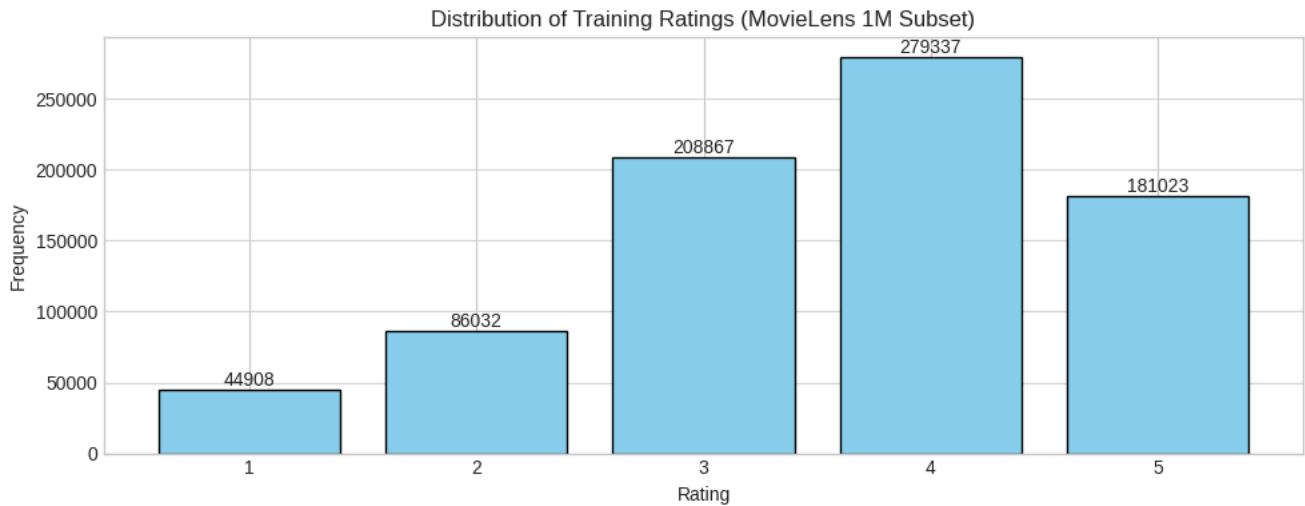
```

```

319 plt.title('Distribution of Training Ratings (MovieLens 1M Subset)')
320 plt.xlabel('Rating'); plt.ylabel('Frequency')
321 plt.xticks([1, 2, 3, 4, 5]); plt.grid(axis='y', alpha=0.75)
322 plt.tight_layout(); plt.show()
323
324 # 2. Ratings per User
325 user_rating_counts = np.bincount(mapped_user_ids_train)
326 plt.figure(figsize=(10, 4))
327 plt.hist(user_rating_counts[user_rating_counts > 0], bins=50, log=True, color='lightcoral', edgecolor='black')
328 plt.title('Distribution of Ratings per User (Training Set)')
329 plt.xlabel('Number of Ratings Given'); plt.ylabel('Number of Users (log scale)')
330 plt.grid(axis='y', alpha=0.75); plt.tight_layout(); plt.show()
331
332 # 3. Ratings per Movie
333 movie_rating_counts = np.bincount(mapped_movie_ids_train)
334 plt.figure(figsize=(10, 4))
335 plt.hist(movie_rating_counts[movie_rating_counts > 0], bins=50, log=True, color='lightgreen', edgecolor='black')
336 plt.title('Distribution of Ratings per Movie (Training Set)')
337 plt.xlabel('Number of Ratings Received'); plt.ylabel('Number of Movies (log scale)')
338 plt.grid(axis='y', alpha=0.75); plt.tight_layout(); plt.show()
339 logger.info("Cell 2.5: Data Visualization Complete.")
340 else:
341     logger.warning("Skipping data visualization as no data was loaded.")

```

Calculating importance sampling weights...  
 Importance sampling enabled (weights based on 800167 ratings).  
 RSVRG Epoch Length set to 61 batches.



```

1 # ===== #
2 # CELL 3: Model Helpers (CONSOLIDATED)
3 # ===== #
4 logger.info("+++ Cell 3: Defining ALL Model Helpers +++")
5
6 # --- Retraction Factory ---
7 class RetractionFactory:
8     _registry = {}
9     @classmethod
10     def register(cls, name):
11         def decorator(fn): cls._registry[name] = fn; return fn
12         return decorator
13     @classmethod
14     def get(cls, name):
15         if name not in cls._registry: raise KeyError(f"Unknown retraction '{name}'. Available: {list(cls._registry.keys())}")
16         return cls._registry[name]
17 # --- Register Retractions ---
18 @RetractionFactory.register("orthonormal")
19 def _retract_qr(U: np.ndarray, V: np.ndarray) -> np.ndarray:
20     """QR-based retraction."""
21     if np.linalg.norm(V) < 1e-12: return U.astype(np.float32)
22     UV = U + V
23     if not np.isfinite(UV).all(): logger.warning("OrthRetraction: Input U+V non-finite."); return U.astype(np.float32)
24     try:
25         Q, R_qr = np.linalg.qr(UV, mode='reduced') # Use 'reduced'
26         if Q.shape[1] < U.shape[1]:
27             pad_width = U.shape[1] - Q.shape[1]; Q = np.pad(Q, ((0, 0), (0, pad_width)), mode='constant')
28             logger.warning(f"OrthRetraction: Padded Q")
29         return Q.astype(np.float32)
30     except Exception as e: logger.error(f"OrthRetraction failed: {e}"); return U.astype(np.float32)
31 @RetractionFactory.register("cayley")
32 def _retract_cayley(U: np.ndarray, V: np.ndarray, alpha: float = 0.1) -> np.ndarray:
33     """ Simple Cayley approx using QR of ambient step. """
34     return _retract_qr(U, alpha * V)
35 @RetractionFactory.register("projection")
36 def _retract_projection(U: np.ndarray, V: np.ndarray) -> np.ndarray:
37     """ Projection (polar decomposition) retraction. """
38     U64 = U.astype(np.float64, copy=False); V64 = V.astype(np.float64, copy=False)
39     Z = U64 + V64; G = Z.T @ Z
40     try:
41         s, P = np.linalg.eigh(G); s_safe = np.maximum(s, 1e-12)
42         s_inv_sqrt = 1.0 / np.sqrt(s_safe); G_mhalf = P @ np.diag(s_inv_sqrt) @ P.T
43         result = (Z @ G_mhalf).astype(np.float32)
44         if result.shape != U.shape: logger.warning(f"Projection Retraction Warning: Shape mismatch. Falling back to QR."); return _retr
45         return result
46     except Exception as e: logger.warning(f"Projection Retraction Warning: {e}. Falling back to QR."); return _retract_qr(U, V)
47 # --- Get the chosen retraction function ---
48 R_fn = RetractionFactory.get(RETRACTION_NAME)
49 if RANK_MPI == 0: logger.info(f"Using Retraction: {RETRACTION_NAME}")
50 def LOSSANDGRAD_TOTAL_DERIVATIVE(
51     U: np.ndarray,
52     X_local: sparse.csc_matrix,
53     mask_coo_global: sparse.coo_matrix,
54     N_users: int,
55     M_movies: int,
56     *,
57     user_data_override: Optional[Dict[int, Dict[str, np.ndarray]]] = None,
58     return_W: bool = False,
59 ) -> Union[
60     Tuple[float, np.ndarray],
61     Tuple[float, np.ndarray, np.ndarray, np.ndarray]
62 ]:
63     """
64     Computes the total profiled loss L(U, W*(U)) and its Euclidean total derivative dL/dU.
65     Solves for W*(U) using the closed-form expression.
66     Optionally returns the local W*(U) and local gradient w.r.t. W.
67
68     Args:
69         U (np.ndarray): Current movie factor matrix, shape (M_movies x RANK), float64.
70         X_local (sparse.csc_matrix): Local partition of the training data matrix (M_movies x N_users).
71         mask_coo_global (sparse.coo_matrix): Global mask matrix (COO) indicating observed entries.
72         N_users (int): Total number of users globally.
73         M_movies (int): Total number of movies globally.
74         user_data_override (dict, optional): Override for user_data_arrays if needed.
75         return_W (bool): If True, also return W_local and local gradient w.r.t. W.
76
77     Returns:

```

```

78     If return_W=False:
79         (total_loss, dl_dU)
80     If return_W=True:
81         (total_loss, dl_dU, local_grad_W, W_local)
82     total_loss is a scalar float64,
83     dl_dU is an (M_movies x RANK) float64 array,
84     local_grad_W is an (RANK x N_users) float64 array,
85     W_local is an (RANK x N_users) float64 array.
86     """
87     U = U.astype(np.float64, copy=False)
88     M, r = U.shape
89
90     # 1) Solve W*(U) for the local columns
91     W_local = WCLOSEDEFFICIENT(
92         U=U,
93         N_users=N_users,
94         user_data_override=user_data_override
95     ) # shape (r x N_users), float64
96
97     # 2) Observed-data term for local slice
98     local_obs_loss = 0.0
99     local_grad_obs_term_U = np.zeros_like(U, dtype=np.float64)
100    local_grad_obs_term_W = np.zeros_like(W_local, dtype=np.float64)
101
102    if X_local.nnz and mask_coo_global.nnz:
103        if not sparse.isspmatrix_coo(mask_coo_global):
104            mask_coo_global = mask_coo_global.tocoo()
105
106        r_ok = (mask_coo_global.row < X_local.shape[0]) & (mask_coo_global.row >= 0)
107        c_ok = (mask_coo_global.col < X_local.shape[1]) & (mask_coo_global.col >= 0)
108        sel = r_ok & c_ok
109        rows = mask_coo_global.row[sel]
110        cols = mask_coo_global.col[sel]
111
112        if rows.size:
113            R_omega = X_local[rows, cols].A1.astype(np.float64)
114            mask_loc = sparse.coo_matrix(
115                (np.ones_like(rows, dtype=np.uint8), (rows, cols)),
116                shape=X_local.shape,
117                dtype=np.uint8,
118            )
119            UW_sparse_local = sparse_product(U, W_local, mask_loc)
120            UW_omega = UW_sparse_local.data.astype(np.float64)
121
122            good = np.isfinite(UW_omega) & np.isfinite(R_omega)
123            if not np.all(good):
124                bad_count = (~good).sum()
125                logger.warning(
126                    "Rank %d: filtered %d non-finite preds/targets locally",
127                    RANK_MPI,
128                    bad_count
129                )
130            UW_omega = UW_omega[good]
131            R_omega = R_omega[good]
132            rows = rows[good]
133            cols = cols[good]
134
135            if UW_omega.size:
136                err_omega = UW_omega - R_omega
137                local_obs_loss = 0.5 * np.dot(err_omega, err_omega)
138
139                E_coo_local = sparse.coo_matrix(
140                    (err_omega, (rows, cols)),
141                    shape=X_local.shape
142                )
143                local_grad_obs_term_U = E_coo_local @ W_local.T
144                local_grad_obs_term_W = U.T @ E_coo_local.tocsc()
145
146    def _allreduce(arr, op=MPI.SUM):
147        if COMM and SIZE_MPI > 1:
148            arr_np = np.asarray(arr, dtype=np.float64)
149            recv = np.zeros_like(arr_np)
150            COMM.Allreduce(arr_np, recv, op=op)
151            if arr_np.ndim == 0:
152                return float(recv)
153            return recv
154        if np.isscalar(arr):

```

```

155         return float(arr)
156     return np.asarray(arr, dtype=np.float64)
157
158     global_obs_loss = _allreduce(local_obs_loss)
159     global_grad_obs_term_U = _allreduce(local_grad_obs_term_U)
160     global_grad_obs_term_W = _allreduce(local_grad_obs_term_W)
161
162     U_fro_sq = np.sum(U**2)
163     local_W_fro_sq = np.sum(W_local**2)
164     global_W_fro_sq = _allreduce(local_W_fro_sq)
165
166     total_loss = (
167         global_obs_loss
168         + 0.5 * LAM_SQ * U_fro_sq
169         + 0.5 * LAM_SQ * global_W_fro_sq
170     )
171
172     dL_dU = global_grad_obs_term_U + LAM_SQ * U
173     local_gW0 = local_grad_obs_term_W
174
175     if not np.isfinite(total_loss):
176         logger.warning("Rank %d: Non-finite loss clamped.", RANK_MPI)
177         total_loss = np.finfo(np.float64).max
178     if not np.isfinite(dL_dU).all():
179         logger.warning("Rank %d: Non-finite dL/dU replaced with zeros.", RANK_MPI)
180         dL_dU = np.nan_to_num(dL_dU)
181     if return_W and not np.isfinite(local_gW0).all():
182         logger.warning("Rank %d: Non-finite local ∇W replaced with zeros.", RANK_MPI)
183         local_gW0 = np.nan_to_num(local_gW0)
184     if return_W and not np.isfinite(W_local).all():
185         logger.warning("Rank %d: Non-finite W_local replaced with zeros.", RANK_MPI)
186         W_local = np.nan_to_num(W_local)
187
188     if return_W:
189         return float(total_loss), dL_dU, local_gW0, W_local
190     else:
191         return float(total_loss), dL_dU
192
193 # --- Other Manifold Helpers ---
194 def ProjTangent(U: np.ndarray, G: np.ndarray) -> np.ndarray:
195     """Project G onto tangent space at U (Grassmann)."""
196     return (G - U @ (U.T @ G)).astype(np.float32)
197 def LogMapApprox(U_base: np.ndarray, U_target: np.ndarray) -> np.ndarray:
198     """Approximate inverse retraction (log map)."""
199     return ProjTangent(U_base, U_target - U_base)
200 def RegularizeGradChordalApprox(U: np.ndarray, U_old: np.ndarray, kappa: float) -> np.ndarray:
201     """Approximate gradient of distance regularization term."""
202     U = U.astype(np.float32); U_old = U_old.astype(np.float32);
203     if REG_DISTANCE == "euclid": S = U.T @ U_old; grad_ambient = U @ (S - S.T); return kappa * ProjTangent(U, grad_ambient)
204     elif REG_DISTANCE == "retraction": v = LogMapApprox(U, U_old); return -kappa * v
205     else: raise ValueError(f"Unknown REG_DISTANCE type: {REG_DISTANCE}")
206
207 # --- RMSE Evaluation ---
208 def evaluate_rmse_with_biases(
209     U: np.ndarray, W: np.ndarray,
210     user_bias: np.ndarray, movie_bias: np.ndarray, global_mean: float,
211     probe_users_mapped: np.ndarray, probe_movies_mapped: np.ndarray, probe_ratings_true: np.ndarray # Now contains true ratings
212 ) -> float:
213     """Computes RMSE on the validation set including bias terms and clamping."""
214     if probe_ratings_true.size == 0: return np.nan # Check if validation set is empty
215     U = U.astype(np.float64, copy=False); W = W.astype(np.float64, copy=False)
216     user_bias = user_bias.astype(np.float64, copy=False); movie_bias = movie_bias.astype(np.float64, copy=False)
217     local_sum_sq_err = 0.0; local_count = 0
218     try:
219         if M_movies_active == 0 or N_users_active == 0: return np.nan
220         if probe_movies_mapped.size > 0 and (probe_movies_mapped.max() >= M_movies_active or probe_movies_mapped.min() < 0): return np.nan
221         if probe_users_mapped.size > 0 and (probe_users_mapped.max() >= N_users_active or probe_users_mapped.min() < 0): return np.nan
222         dot_prods = np.array([np.dot(U[m, :], W[:, u]) for m, u in zip(probe_movies_mapped, probe_users_mapped)], dtype=np.float64)
223         preds_raw = global_mean + user_bias[probe_users_mapped] + movie_bias[probe_movies_mapped] + dot_prods
224         preds_clamped = np.clip(preds_raw, 1.0, 5.0)
225         if not np.isfinite(preds_clamped).all(): preds_clamped = np.nan_to_num(preds_clamped, nan=global_mean)
226         if not np.isfinite(probe_ratings_true).all(): probe_ratings_true = np.nan_to_num(probe_ratings_true)
227         squared_errors = (preds_clamped - probe_ratings_true)**2
228         local_sum_sq_err = np.sum(squared_errors)
229         local_count = len(squared_errors)
230     except IndexError as e: logger.error(f"IndexError during biased RMSE: {e}"); return np.nan
231     except Exception as e: logger.error(f"Error during biased RMSE: {e}"); return np.nan

```



```

232 # --- MPI Reduction for RMSE ---
233 if COMM and SIZE_MPI > 1:
234     global_sum_sq_err_buf = np.array(local_sum_sq_err, dtype=np.float64); global_count_buf = np.array(local_count, dtype=np.int64)
235     global_sum_sq_err = np.array(0.0, dtype=np.float64); global_count = np.array(0, dtype=np.int64)
236     COMM.Allreduce(global_sum_sq_err_buf, global_sum_sq_err, op=MPI.SUM); COMM.Allreduce(global_count_buf, global_count, op=MPI.SUM)
237     if global_count > 0: mean_squared_error = global_sum_sq_err / global_count
238     else: return np.nan
239 else: # Serial case
240     if local_count > 0: mean_squared_error = local_sum_sq_err / local_count
241     else: return np.nan
242 mean_squared_error = max(0.0, mean_squared_error); rmse = np.sqrt(mean_squared_error)
243 return float(rmse) if np.isfinite(rmse) else np.nan
244
245 # --- RMSE Helper for SoftImpute (No Biases) ---
246 def evaluate_rmse_low_rank(U, S, V, probe_movies_mapped, probe_users_mapped, probe_ratings_true, use_gpu=False):
247     """Computes RMSE for low-rank model  $X = USV^T$  against true ratings."""
248     if probe_ratings_true.size == 0: return np.nan
249     xp = cp if use_gpu else np
250     try:
251         if M_movies_active == 0 or N_users_active == 0: return np.nan
252         if probe_movies_mapped.size > 0 and (probe_movies_mapped.max() >= M_movies_active or probe_movies_mapped.min() < 0): return np.nan
253         if probe_users_mapped.size > 0 and (probe_users_mapped.max() >= N_users_active or probe_users_mapped.min() < 0): return np.nan
254         U_dev = xp.asarray(U); S_dev = xp.asarray(S); V_dev = xp.asarray(V)
255         probe_movies_dev = xp.asarray(probe_movies_mapped); probe_users_dev = xp.asarray(probe_users_mapped)
256         probe_ratings_dev = xp.asarray(probe_ratings_true)
257         term2 = S_dev * V_dev[probe_users_dev, :]
258         preds_raw = xp.sum(U_dev[probe_movies_dev, :] * term2, axis=1)
259         preds_clamped = xp.clip(preds_raw, 1.0, 5.0)
260         if not xp.isfinite(preds_clamped).all(): preds_clamped = xp.nan_to_num(preds_clamped, nan=3.0)
261         if not xp.isfinite(probe_ratings_dev).all(): probe_ratings_dev = xp.nan_to_num(probe_ratings_dev)
262         mse_dev = xp.mean((preds_clamped - probe_ratings_dev)**2)
263         mse = float(cp.asnumpy(mse_dev) if use_gpu else mse_dev)
264         rmse = np.sqrt(mse) if mse >= 0 else np.nan
265     except IndexError as e: logger.error(f"IndexError during low-rank RMSE: {e}"); return np.nan
266     except Exception as e: logger.error(f"Error during low-rank RMSE: {e}"); return np.nan
267     return float(rmse) if np.isfinite(rmse) else np.nan
268
269 # --- Initialization ---
270 def initialize_factors_and_biases(M: int, N: int, R: int, rng: Generator, scale: float) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
271     """Initializes U, W, user_bias, movie_bias."""
272     U = None; W = None; user_bias = None; movie_bias = None
273     if RANK_MPI == 0:
274         U = rng.standard_normal(size=(M, R)).astype(np.float64) * scale
275         W = rng.standard_normal(size=(R, N)).astype(np.float64) * scale
276         user_bias = np.zeros(N, dtype=np.float64)
277         movie_bias = np.zeros(M, dtype=np.float64)
278         if M >= R: U_orth, _ = np.linalg.qr(U, mode='reduced'); U = U_orth.astype(np.float64)
279         else: logger.warning(f"M ({M}) < R ({R}). Cannot orthonormalize U.")
280     if COMM and SIZE_MPI > 1:
281         if RANK_MPI != 0: U = np.empty((M, R), dtype=np.float64); W = np.empty((R, N), dtype=np.float64); user_bias = np.empty(N, dtype=np.float64)
282         COMM.Bcast(U, root=0); COMM.Bcast(W, root=0); COMM.Bcast(user_bias, root=0); COMM.Bcast(movie_bias, root=0)
283     return U, W, user_bias, movie_bias
284
285 # --- Initial State Recorder ---
286 def record_initial_state_biased(U, W, user_bias, movie_bias, loss_args_biased, eval_args_biased):
287     """Computes and logs initial state for biased models."""
288     current_loss, gU0, gW0, gBu0, gBi0 = loss_and_grad_serial_with_biases(U, W, user_bias, movie_bias, *loss_args_biased)
289     current_rmse = evaluate_rmse_with_biases(U, W, user_bias, movie_bias, *eval_args_biased)
290     gU_proj_0 = ProjTangent(U, gU0)
291     grad_norm_U_riemann = np.linalg.norm(gU_proj_0)
292     grad_norm_W = np.linalg.norm(gW0); grad_norm_Bu = np.linalg.norm(gBu0); grad_norm_Bi = np.linalg.norm(gBi0)
293     if RANK_MPI == 0: logger.info(
294         f"Epoch 00 (Init): Loss={current_loss:.4e}, RMSE={current_rmse:.4f}, "
295         f"||Proj gU||={grad_norm_U_riemann:.2e}, ||gW||={grad_norm_W:.2e}, "
296         f"||gBu||={grad_norm_Bu:.2e}, ||gBi||={grad_norm_Bi:.2e}"
297     )
298     if not np.isfinite(current_loss): raise ValueError("Initial loss is not finite.")
299     return current_loss, current_rmse, gU0, gW0, gBu0, gBi0
300
301 # --- Armijo Line Search ---
302 def ArmijoLineSearchRiemannian(
303     U: np.ndarray, G_euclidean: np.ndarray, loss_args: tuple, current_loss: float,
304     lr_init: float, beta: float, sigma: float, max_ls_iter: int = 20
305 ) -> Tuple[float, np.ndarray, float]:
306     """Performs Armijo line search using retraction."""
307     lr = lr_init
308     G_proj = ProjTangent(U, G_euclidean)

```

```

309 G_proj_norm_sq = np.linalg.norm(G_proj)**2
310 if G_proj_norm_sq < 1e-14: return 0.0, U, current_loss
311 for ls_iter in range(max_ls_iter):
312     step_vec = -lr * G_proj
313     U_next = R_fn(U, step_vec) # Use chosen retraction
314     if not np.isfinite(U_next).all(): lr *= beta; continue
315     try:
316         W_ls, ub_ls, mb_ls, *rest_args = loss_args
317         loss_next, _, _, _ = loss_and_grad_serial_with_biases(U_next, W_ls, ub_ls, mb_ls, *rest_args)
318     except Exception as e: logger.error(f"Armijo LS Error: {e}"); return 0.0, U, current_loss
319     if not np.isfinite(loss_next): lr *= beta; continue
320     required_decrease = sigma * lr * G_proj_norm_sq
321     actual_decrease = current_loss - loss_next
322     if actual_decrease >= required_decrease - 1e-9: return lr, U_next, loss_next
323     lr *= beta
324     if lr < 1e-14: break
325 logger.debug("Armijo LS failed."); return 0.0, U, current_loss
326
327 # --- Adaptive Stopping Check ---
328 def should_stop_subproblem(G_proj, step_vec):
329     """Return True if both criteria are already small."""
330     grad_norm_proj = np.linalg.norm(G_proj)
331     step_norm = np.linalg.norm(step_vec)
332     stop = (grad_norm_proj < ETA_GRAD and step_norm < ETA_DIST)
333     return stop
334
335 # --- Adaptive Kappa Update ---
336 def update_kappa_adaptive(kappa_prev, h_hist, dist_hist, U_local,
337                           gamma=2.0, window=3,
338                           kappa_min=1e-4, kappa_max=1e12):
339     """ Adaptive kappa update using local curvature estimate. """
340     if U_local.shape[1] == 0: return kappa_min # Handle empty matrix case
341     v = GLOBAL_RNG.standard_normal(size=(U_local.shape[1], 1)).astype(U_local.dtype)
342     v /= np.linalg.norm(v) + 1e-12
343     U_local_64 = U_local.astype(np.float64); v_64 = v.astype(np.float64)
344     lambda_max_sq = 0.0
345     for _ in range(2): # 2 power iterations on U^T U
346         Av = U_local_64.T @ (U_local_64 @ v_64)
347         lambda_max_sq = v_64.T @ Av
348         v_norm = np.linalg.norm(Av); v_64 = Av / (v_norm + 1e-12)
349     L_local = np.sqrt(max(0.0, lambda_max_sq.item()))
350     target_ratio = 0.9; target = target_ratio * L_local
351     kappa_new = np.clip(target, kappa_min, kappa_max)
352     return float(kappa_new)
353
354 # --- OT Demo Helper ---
355 def run_barycentre_demo(n_grid=200, reg=1e-1, rng_seed=0):
356     """ POT demo: 3 one-dimensional Gaussians -> entropic Wasserstein barycenter """
357     if not OT_AVAILABLE: return None
358     grid = np.linspace(-8.0, 8.0, n_grid)
359     M = ot.dist(grid.reshape(-1, 1), grid.reshape(-1, 1)) ** 2
360     means = np.array([-3.0, 0.0, 3.0]); sigmas = np.array([0.5, 1.0, 0.7])
361     sources = np.vstack([np.exp(-0.5 * ((grid - m) / s) ** 2) / (s * np.sqrt(2 * np.pi)) for m, s in zip(means, sigmas)]).T
362     sources /= sources.sum(axis=0, keepdims=True)
363     bary, log = ot.bregman.barycenter(sources, M, reg, weights=None, numItermax=1000, stopThr=1e-7, log=True)
364     return {'grid': grid, 'sources': sources, 'barycenter': bary, 'log': log}
365
366
367 logger.info("Cell 3: Model Helpers Defined.")

1
2
3 # =====
4 # CELL 4: Non-Convex Solvers (SVRG, ALS, Euclidean GD) - Renumbered
5 # =====
6 logger.info("+++ Cell 4: Defining Non-Convex Solvers +++")
7 # --- Loss/Gradient Functions ---
8 def loss_and_grad_serial_with_biases(
9     U: np.ndarray, W: np.ndarray, user_bias: np.ndarray, movie_bias: np.ndarray,
10     global_mean: float,
11     rows_idx: np.ndarray, cols_idx: np.ndarray, vals_true_centered: np.ndarray, # Centered ratings
12     n_movies_func: int, n_users_func: int, rank_func: int,
13     lambda_sq_func: float, lambda_bias_func: float
14 ) -> Tuple[float, np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
15     """ Computes loss and gradients for U, W, user_bias, movie_bias. """
16     # ... (implementation from v11) ...

```

```

17 U = U.astype(np.float64, copy=False); W = W.astype(np.float64, copy=False)
18 user_bias = user_bias.astype(np.float64, copy=False); movie_bias = movie_bias.astype(np.float64, copy=False)
19 if vals_true_centered.size == 0: return 0.0, np.zeros_like(U), np.zeros_like(W), np.zeros_like(user_bias), np.zeros_like(movie_bias)
20 try:
21     W_cols = W[:, cols_idx]; U_rows = U[rows_idx, :]
22     dot_prods = np.sum(U_rows * W_cols.T, axis=1)
23     preds_residual = user_bias[cols_idx] + movie_bias[rows_idx] + dot_prods
24 except IndexError as e: logger.error(f"Indexing error in loss_and_grad_serial_with_biases - {e}"); raise
25 valid_mask = np.isfinite(preds_residual) & np.isfinite(vals_true_centered)
26 if not np.all(valid_mask):
27     logger.warning(f"Filtering {np.sum(~valid_mask)} non-finite values in loss_and_grad_serial_with_biases.")
28     rows_idx_filt = rows_idx[valid_mask]; cols_idx_filt = cols_idx[valid_mask]
29     vals_true_filt = vals_true_centered[valid_mask]; preds_filt = preds_residual[valid_mask]
30     if preds_filt.size == 0: return np.inf, np.zeros_like(U), np.zeros_like(W), np.zeros_like(user_bias), np.zeros_like(movie_bias)
31 else:
32     rows_idx_filt, cols_idx_filt, vals_true_filt, preds_filt = rows_idx, cols_idx, vals_true_centered, preds_residual
33 errors = preds_filt - vals_true_filt
34 loss_obs = 0.5 * np.sum(errors**2)
35 loss_reg_U = 0.5 * lambda_sq_func * np.sum(U**2); loss_reg_W = 0.5 * lambda_sq_func * np.sum(W**2)
36 loss_reg_bu = 0.5 * lambda_bias_func * np.sum(user_bias**2); loss_reg_bi = 0.5 * lambda_bias_func * np.sum(movie_bias**2)
37 total_loss = loss_obs + loss_reg_U + loss_reg_W + loss_reg_bu + loss_reg_bi
38 E_sparse = sparse.csr_matrix((errors, (rows_idx_filt, cols_idx_filt)), shape=(n_movies_func, n_users_func))
39 E_sparse_csc = E_sparse.tocsc()
40 grad_U = E_sparse @ W.T + lambda_sq_func * U
41 grad_W = U.T @ E_sparse_csc + lambda_sq_func * W
42 grad_user_bias = np.array(E_sparse.sum(axis=0)).flatten() + lambda_bias_func * user_bias
43 grad_movie_bias = np.array(E_sparse.sum(axis=1)).flatten() + lambda_bias_func * movie_bias
44 if not np.isfinite(grad_U).all(): grad_U = np.nan_to_num(grad_U)
45 if not np.isfinite(grad_W).all(): grad_W = np.nan_to_num(grad_W)
46 if not np.isfinite(grad_user_bias).all(): grad_user_bias = np.nan_to_num(grad_user_bias)
47 if not np.isfinite(grad_movie_bias).all(): grad_movie_bias = np.nan_to_num(grad_movie_bias)
48 if not np.isfinite(total_loss): total_loss = np.inf
49 return float(total_loss), grad_U.astype(np.float64), grad_W.astype(np.float64), grad_user_bias.astype(np.float64), grad_movie_bias.
50
51 def gradient_batch_with_biases(
52     U: np.ndarray, W: np.ndarray, user_bias: np.ndarray, movie_bias: np.ndarray,
53     indices: np.ndarray, # Indices into GLOBAL triplets
54     rows_idx: np.ndarray, cols_idx: np.ndarray, vals_true_centered: np.ndarray, # Centered ratings
55     n_ratings_total: int,
56     lambda_sq_func: float, lambda_bias_func: float
57 ) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray]:
58     """ Computes average Euclidean gradient over a BATCH of ratings, including biases. """
59     U = U.astype(np.float64, copy=False)
60     W = W.astype(np.float64, copy=False)
61     user_bias = user_bias.astype(np.float64, copy=False)
62     movie_bias = movie_bias.astype(np.float64, copy=False)
63     batch_size = len(indices)
64     if batch_size == 0:
65         return np.zeros_like(U), np.zeros_like(W), np.zeros_like(user_bias), np.zeros_like(movie_bias)
66
67     # Get data for the batch
68     batch_rows = rows_idx[indices]
69     batch_cols = cols_idx[indices]
70     batch_vals_centered = vals_true_centered[indices]
71
72     # Get corresponding factors and biases
73     try:
74         U_batch = U[batch_rows, :] # Shape (B, R)
75         W_batch = W[:, batch_cols] # Shape (R, B)
76         user_bias_batch = user_bias[batch_cols] # Shape (B,)
77         movie_bias_batch = movie_bias[batch_rows] # Shape (B,)
78     except IndexError as e:
79         logger.error(f"Indexing error in gradient_batch_with_biases - {e}")
80         raise
81
82     # Predict residual for the batch
83     preds_batch_residual = user_bias_batch + movie_bias_batch + np.sum(U_batch * W_batch.T, axis=1)
84
85     # Calculate errors for the batch
86     errors_batch = preds_batch_residual - batch_vals_centered # Shape (B,)
87
88     # Calculate gradient terms using sparse matrix approach
89     E_sparse_batch = sparse.csr_matrix((errors_batch, (batch_rows, batch_cols)),
90                                         shape=(U.shape[0], W.shape[1]))
91
92     # Average gradient over the batch
93     grad_U_batch = (E_sparse_batch @ W.T) / batch_size + lambda_sq_func * U

```

```

94     grad_W_batch = (U.T @ E_sparse_batch.tocsc()) / batch_size + lambda_sq_func * W
95
96     # Compute bias gradients (need to average errors per user/movie in batch)
97     # This requires accumulating errors per user/movie index present in the batch
98     grad_user_bias_batch = np.zeros_like(user_bias)
99     grad_movie_bias_batch = np.zeros_like(movie_bias)
100    np.add.at(grad_user_bias_batch, batch_cols, errors_batch) # Accumulate errors by user index
101    np.add.at(grad_movie_bias_batch, batch_rows, errors_batch) # Accumulate errors by movie index
102
103    grad_user_bias_batch = grad_user_bias_batch / batch_size + lambda_bias_func * user_bias
104    grad_movie_bias_batch = grad_movie_bias_batch / batch_size + lambda_bias_func * movie_bias
105
106    # Handle potential non-finite values
107    if not np.isfinite(grad_U_batch).all(): grad_U_batch = np.nan_to_num(grad_U_batch)
108    if not np.isfinite(grad_W_batch).all(): grad_W_batch = np.nan_to_num(grad_W_batch)
109    if not np.isfinite(grad_user_bias_batch).all(): grad_user_bias_batch = np.nan_to_num(grad_user_bias_batch)
110    if not np.isfinite(grad_movie_bias_batch).all(): grad_movie_bias_batch = np.nan_to_num(grad_movie_bias_batch)
111
112    return grad_U_batch.astype(np.float64), grad_W_batch.astype(np.float64), grad_user_bias_batch.astype(np.float64), grad_movie_bias_b
113
114 # --- SVRG Solver ---
115 # --- SVRG Solver with Biases ---
116 def run_non_convex_svrg_with_biases(
117     R_train_coo: sparse.coo_matrix, # Contains centered ratings
118     global_mean: float,
119     probe_users_mapped: np.ndarray, # Mapped probe indices
120     probe_movies_mapped: np.ndarray,
121     probe_ratings_true: np.ndarray, # Original probe ratings
122     N_users_active: int,
123     M_movies_active: int,
124     rank_local: int,
125     n_epochs: int,
126     inner_lr: float, # Base inner learning rate
127     batch_size: int,
128     lam_sq: float,
129     lam_bias: float,
130     rng: Generator,
131     init_scale: float = INIT_SCALE_NON_CONVEX,
132     max_grad_norm: float = GRAD_CLIP_THRESHOLD
133 ) -> Dict[str, List]:
134     """
135     Runs SVRG for non-convex UW factorization including bias terms.
136     Uses decaying LR and gradient clipping.
137     """
138     logger.info("Starting Non-Convex SVRG Solver with Biases...")
139     # Initialize factors and biases
140     U, W, user_bias, movie_bias = initialize_factors_and_biases(
141         M_movies_active, N_users_active, rank_local, rng, init_scale
142     )
143
144     hist_loss = []
145     hist_rmse = []
146     hist_time = []
147     hist_gU_norm, hist_gW_norm, hist_gBu_norm, hist_gBi_norm = [], [], [], []
148
149     start_time = time.time()
150
151     # Use mapped indices and centered ratings for training
152     train_rows = R_train_coo.row
153     train_cols = R_train_coo.col
154     train_vals_centered = R_train_coo.data
155     n_ratings_total = R_train_coo.nnz
156
157     if n_ratings_total == 0:
158         logger.error("No training ratings available.")
159         return {'loss': [], 'rmse': [], 'time': [], 'gU_norm': [], 'gW_norm': [], 'gBu_norm': [], 'gBi_norm': [], 'U': None, 'W': None,
160
161     # Initial evaluation
162     try:
163         loss0, gU0, gW0, gBu0, gBi0 = loss_and_grad_serial_with_biases(
164             U, W, user_bias, movie_bias, global_mean,
165             train_rows, train_cols, train_vals_centered,
166             M_movies_active, N_users_active, rank_local, lam_sq, lam_bias
167         )
168         rmse0 = evaluate_rmse_with_biases(
169             U, W, user_bias, movie_bias, global_mean,
170             probe_users_mapped, probe_movies_mapped, probe_ratings_true

```

```

171     )
172     hist_loss.append(loss0)
173     hist_rmse.append(rmse0)
174     hist_time.append(time.time() - start_time)
175     hist_gU_norm.append(np.linalg.norm(gU0))
176     hist_gW_norm.append(np.linalg.norm(gW0))
177     hist_gBu_norm.append(np.linalg.norm(gBu0))
178     hist_gBi_norm.append(np.linalg.norm(gBi0))
179     logger.info(
180         f"Epoch 00 (Init): Loss={loss0:.4e}, RMSE={rmse0:.4f}, "
181         f"||gU||={hist_gU_norm[-1]:.2e}, ||gW||={hist_gW_norm[-1]:.2e}, "
182         f"||gBu||={hist_gBu_norm[-1]:.2e}, ||gBi||={hist_gBi_norm[-1]:.2e}"
183     )
184 except Exception as e:
185     logger.error(f"Error during initial evaluation: {e}", exc_info=True)
186     return {'loss': [], 'rmse': [], 'time': [], 'gU_norm': [], 'gW_norm': [], 'gBu_norm': [], 'gBi_norm': [], 'U': None, 'W': None,
187
188 # Main SVRG Loop
189 for epoch in range(1, n_epochs + 1):
190     epoch_start_time = time.time()
191     logger.info(f"--- Starting Epoch {epoch:02d} ---")
192     # --- Use Exponential Decay for Learning Rate (FIX 4) ---
193     lr_epoch = inner_lr * (0.9**(epoch - 1)) # Exponential decay
194     logger.info(f"Using lr = {lr_epoch:.2e} this epoch")
195     # -----
196
197     # Compute anchor gradient
198     logger.info(f"Epoch {epoch:02d}: Computing anchor gradient...")
199     anchor_start_time = time.time()
200     try:
201         loss_anchor, gU_anchor, gW_anchor, gBu_anchor, gBi_anchor = loss_and_grad_serial_with_biases(
202             U, W, user_bias, movie_bias, global_mean,
203             train_rows, train_cols, train_vals_centered,
204             M_movies_active, N_users_active, rank_local, lam_sq, lam_bias
205         )
206         logger.info(f"Epoch {epoch:02d}: Anchor gradient computed in {time.time() - anchor_start_time:.2f}s.")
207     except Exception as e:
208         logger.error(f"Error computing anchor gradient at epoch {epoch}: {e}")
209         break
210
211     U_epoch_start, W_epoch_start = U.copy(), W.copy()
212     user_bias_epoch_start, movie_bias_epoch_start = user_bias.copy(), movie_bias.copy()
213
214     # Inner loop
215     # --- Use Full Inner Pass (FIX 5) ---
216     num_inner_steps = max(1, (n_ratings_total // batch_size) // SVRG_INNER_STEPS_DIVISOR)
217     logger.info(f"Epoch {epoch:02d}: Starting inner loop with {num_inner_steps} steps...")
218     inner_loop_start_time = time.time()
219
220     for inner_step in range(num_inner_steps):
221         batch_indices = rng.choice(n_ratings_total, size=batch_size, replace=False)
222         try:
223             gU_curr, gW_curr, gBu_curr, gBi_curr = gradient_batch_with_biases(
224                 U, W, user_bias, movie_bias, batch_indices,
225                 train_rows, train_cols, train_vals_centered,
226                 n_ratings_total, lam_sq, lam_bias)
227             gU_anch, gW_anch, gBu_anch, gBi_anch = gradient_batch_with_biases(
228                 U_epoch_start, W_epoch_start, user_bias_epoch_start, movie_bias_epoch_start,
229                 batch_indices, train_rows, train_cols, train_vals_centered,
230                 n_ratings_total, lam_sq, lam_bias)
231         except Exception as e:
232             logger.error(f"Error computing stochastic gradient: {e}")
233             continue
234
235         # Variance-reduced gradients
236         gU_vr = gU_curr - gU_anch + gU_anchor
237         gW_vr = gW_curr - gW_anch + gW_anchor
238         gBu_vr = gBu_curr - gBu_anch + gBu_anchor
239         gBi_vr = gBi_curr - gBi_anch + gBi_anchor
240
241         # Gradient clipping
242         gU_norm = np.linalg.norm(gU_vr); gW_norm = np.linalg.norm(gW_vr)
243         gBu_norm = np.linalg.norm(gBu_vr); gBi_norm = np.linalg.norm(gBi_vr)
244         if gU_norm > max_grad_norm: gU_vr *= (max_grad_norm / gU_norm)
245         if gW_norm > max_grad_norm: gW_vr *= (max_grad_norm / gW_norm)
246         if gBu_norm > max_grad_norm: gBu_vr *= (max_grad_norm / gBu_norm)
247         if gBi_norm > max_grad_norm: gBi_vr *= (max_grad_norm / gBi_norm)

```

```

248
249     # Update factors and biases
250     U -= lr_epoch * gU_vr
251     W -= lr_epoch * gW_vr
252     user_bias -= lr_epoch * gBu_vr
253     movie_bias -= lr_epoch * gBi_vr
254
255     if (inner_step + 1) % 5000 == 0: # Log less frequently for full inner pass
256         logger.info(f"Epoch {epoch:02d}: Inner step {inner_step+1}/{num_inner_steps} done.")
257
258     logger.info(f"Epoch {epoch:02d}: Inner loop finished in {time.time() - inner_loop_start_time:.2f}s.")
259
260     # Evaluate after epoch
261     logger.info(f"Epoch {epoch:02d}: Evaluating loss and RMSE...")
262     eval_start_time = time.time()
263     try:
264         loss_k, gU_k, gW_k, gBu_k, gBi_k = loss_and_grad_serial_with_biases(
265             U, W, user_bias, movie_bias, global_mean,
266             train_rows, train_cols, train_vals_centered,
267             M_movies_active, N_users_active, rank_local, lam_sq, lam_bias
268         )
269         if not np.isfinite(loss_k):
270             logger.error(f"Epoch {epoch:02d}: Loss became non-finite ({loss_k}). Stopping.")
271             hist_loss.append(np.nan); hist_rmse.append(np.nan); hist_time.append(time.time() - start_time)
272             hist_gU_norm.append(np.nan); hist_gW_norm.append(np.nan); hist_gBu_norm.append(np.nan); hist_gBi_norm.append(np.nan)
273             break
274
275         rmse_k = evaluate_rmse_with_biases(
276             U, W, user_bias, movie_bias, global_mean,
277             probe_users_mapped, probe_movies_mapped, probe_ratings_true
278         )
279         hist_loss.append(loss_k); hist_rmse.append(rmse_k)
280         hist_time.append(time.time() - start_time)
281         hist_gU_norm.append(np.linalg.norm(gU_k)); hist_gW_norm.append(np.linalg.norm(gW_k))
282         hist_gBu_norm.append(np.linalg.norm(gBu_k)); hist_gBi_norm.append(np.linalg.norm(gBi_k))
283
284         logger.info(f"Epoch {epoch:02d}: Eval done in {time.time() - eval_start_time:.2f}s. ")
285         logger.info(
286             f"Loss={loss_k:.4e}, RMSE={rmse_k:.4f}, "
287             f"||gU||={hist_gU_norm[-1]:.2e}, ||gW||={hist_gW_norm[-1]:.2e}, "
288             f"||gBu||={hist_gBu_norm[-1]:.2e}, ||gBi||={hist_gBi_norm[-1]:.2e}"
289         )
290     except Exception as e:
291         logger.error(f"Error during evaluation at epoch {epoch}: {e}", exc_info=True)
292         hist_loss.append(np.nan); hist_rmse.append(np.nan); hist_time.append(time.time() - start_time)
293         hist_gU_norm.append(np.nan); hist_gW_norm.append(np.nan); hist_gBu_norm.append(np.nan); hist_gBi_norm.append(np.nan)
294         break
295
296     logger.info(f"--- Epoch {epoch:02d} finished in {time.time() - epoch_start_time:.2f}s ---")
297
298     logger.info("Non-Convex SVRG Solver with Biases Finished.")
299     return {
300         'loss': hist_loss, 'rmse': hist_rmse, 'time': hist_time,
301         'gU_norm': hist_gU_norm, 'gW_norm': hist_gW_norm,
302         'gBu_norm': hist_gBu_norm, 'gBi_norm': hist_gBi_norm,
303         'U': U, 'W': W, 'bu': user_bias, 'bi': movie_bias
304     }
305
306
307 # --- ALS Solver ---
308
309 def W_closed_efficient(U, N_users, N_movies, user_indices=None):
310     # Solves for W for a subset of users (local computation)
311     U = U.astype(np.float32, copy=False);
312     target_users = user_indices if user_indices is not None else user_data_arrays.keys()
313     W_subset = {} # Use dict if only computing for subset
314     I_r_lam_sq = (LAM_SQ * I_r).astype(np.float32) # lambda^2 * I
315
316     for u in target_users:
317         if u not in user_data_arrays: continue
318         data = user_data_arrays[u]
319         movie_indices = data['movies']; rs_t = data['rs']
320         if movie_indices.size == 0: continue
321         # Check bounds before indexing U
322         if movie_indices.max() >= U.shape[0] or movie_indices.min() < 0:
323             # if RANK_MPI == 0: print(f"Warning: Invalid movie indices for user {u}. Skipping.")
324             continue

```

```

325     U_k = U[movie_indices, :]
326     A = U_k.T @ U_k + I_r_lam_sq
327     B = U_k.T @ rs_t
328     A = A.astype(np.float32); B = B.astype(np.float32)
329     try:
330         w_u = np.linalg.solve(A.astype(np.float64), B.astype(np.float64)).astype(np.float32)
331     except np.linalg.LinAlgError:
332         # if RANK_MPI == 0: print(f"Warning: np.linalg.solve failed for user {u}. Using pseudo-inverse.")
333         try:
334             w_u = (np.linalg.pinv(A.astype(np.float64)) @ B.astype(np.float64)).astype(np.float32)
335         except np.linalg.LinAlgError:
336             if RANK_MPI == 0: print(f"ERROR: Pseudo-inverse also failed for user {u}. Returning zero vector.")
337             w_u = np.zeros(RANK, dtype=np.float32) # Return zero vector if fails completely
338         except Exception as e_pinv:
339             if RANK_MPI == 0: print(f"ERROR: Unknown error in pseudo-inverse for user {u}: {e_pinv}. Returning zero vector.")
340             w_u = np.zeros(RANK, dtype=np.float32)
341
342     if user_indices is not None:
343         W_subset[u] = w_u
344     else:
345         if 'W' not in locals(): W = np.zeros((RANK, N_users), dtype=np.float32)
346         if 0 <= u < W.shape[1]: # Check user index bound for W
347             W[:, u] = w_u
348         # else: # This shouldn't happen if N_users is correct
349         #     if RANK_MPI == 0: print(f"Warning: User index {u} out of bounds for W (shape {W.shape}).")
350
351
352     if user_indices is not None:
353         return W_subset # Return dict
354     else:
355         if 'W' not in locals():
356             # if RANK_MPI == 0: print("Warning: W_closed_efficient called with no active users? Returning empty W.")
357             return np.zeros((RANK, N_users), dtype=np.float32)
358         # W should be filled now
359         if not np.isfinite(W).all():
360             if RANK_MPI == 0: print("Warning: Non-finite values found in computed W matrix. Clamping.")
361             W = np.nan_to_num(W, nan=0.0, posinf=0.0, neginf=0.0) # Clamp non-finite to zero
362         assert W.shape == (RANK, N_users);
363         return W # Return full W matrix
364
365
366 def update_user_factors(
367     R_train_coo_csc: sparse.csc_matrix, # Centered ratings, CSC format
368     U: np.ndarray,
369     user_bias: np.ndarray,
370     movie_bias: np.ndarray,
371     lambda_sq: float,
372     rank: int,
373     N_users: int
374 ) -> np.ndarray:
375     """Solves for W (user factors) fixing U and biases."""
376     M = U.shape[0]
377     W = np.zeros((rank, N_users), dtype=np.float64)
378     # Precompute U^T U + lambda*I (used in the denominator)
379     # Note: This is used inside the loop per user based on specific movies U_j
380     # UtU = U.T @ U + lambda_sq * np.eye(rank, dtype=np.float64) # Can't precompute fully
381
382     for j in range(N_users):
383         # Find ratings for user j
384         start_idx = R_train_coo_csc.indptr[j]
385         end_idx = R_train_coo_csc.indptr[j+1]
386         if start_idx == end_idx: # No ratings for this user
387             continue
388
389         movie_indices = R_train_coo_csc.indices[start_idx:end_idx]
390         ratings_centered = R_train_coo_csc.data[start_idx:end_idx]
391
392         U_j = U[movie_indices, :] # Movies rated by user j (n_j x R)
393
394         # Adjust ratings by movie bias: r_ij - mu - b_i
395         adjusted_ratings = ratings_centered - movie_bias[movie_indices]
396
397         # Calculate A = U_j^T U_j + lambda*I
398         A = U_j.T @ U_j + lambda_sq * np.eye(rank, dtype=np.float64)
399
400         # Calculate b = U_j^T * adjusted_ratings
401         b = U_j.T @ adjusted_ratings

```

```

402
403     try:
404         W[:, j] = np.linalg.solve(A, b)
405     except np.linalg.LinAlgError:
406         logger.warning(f"ALS: Solve failed for user {j}, using pseudo-inverse.")
407     try:
408         W[:, j] = np.linalg.pinv(A) @ b
409     except Exception as e_pinv:
410         logger.error(f"ALS: Pseudo-inverse failed for user {j}: {e_pinv}. Setting W_j to zero.")
411         W[:, j] = 0.0 # Set to zero vector
412
413     return W.astype(np.float64)
414
415 def update_movie_factors(
416     R_train_coo_csr: sparse.csr_matrix, # Centered ratings, CSR format
417     W: np.ndarray,
418     user_bias: np.ndarray,
419     movie_bias: np.ndarray,
420     lambda_sq: float,
421     rank: int,
422     M_movies: int
423 ) -> np.ndarray:
424     """Solves for U (movie factors) fixing W and biases."""
425     N = W.shape[1]
426     U = np.zeros((M_movies, rank), dtype=np.float64)
427     # Precompute W W^T + lambda*I (used in the denominator)
428     # Note: This is used inside the loop per movie based on specific users W_i
429     # WtW = W @ W.T + lambda_sq * np.eye(rank, dtype=np.float64) # Can't precompute fully
430
431     for i in range(M_movies):
432         # Find ratings for movie i
433         start_idx = R_train_coo_csr.indptr[i]
434         end_idx = R_train_coo_csr.indptr[i+1]
435         if start_idx == end_idx: # No ratings for this movie
436             continue
437
438         user_indices = R_train_coo_csr.indices[start_idx:end_idx]
439         ratings_centered = R_train_coo_csr.data[start_idx:end_idx]
440
441         W_i = W[:, user_indices] # Users who rated movie i (R x n_i)
442
443         # Adjust ratings by user bias: r_ij - mu - b_u
444         adjusted_ratings = ratings_centered - user_bias[user_indices]
445
446         # Calculate A = W_i W_i^T + lambda*I
447         A = W_i @ W_i.T + lambda_sq * np.eye(rank, dtype=np.float64)
448
449         # Calculate b = W_i * adjusted_ratings
450         b = W_i @ adjusted_ratings
451
452     try:
453         U[i, :] = np.linalg.solve(A, b)
454     except np.linalg.LinAlgError:
455         logger.warning(f"ALS: Solve failed for movie {i}, using pseudo-inverse.")
456     try:
457         U[i, :] = np.linalg.pinv(A) @ b
458     except Exception as e_pinv:
459         logger.error(f"ALS: Pseudo-inverse failed for movie {i}: {e_pinv}. Setting U_i to zero.")
460         U[i, :] = 0.0 # Set to zero vector
461
462     return U.astype(np.float64)
463
464
465 def update_biases(
466     R_train_coo: sparse.coo_matrix, # Centered ratings
467     U: np.ndarray,
468     W: np.ndarray,
469     user_bias: np.ndarray,
470     movie_bias: np.ndarray,
471     global_mean: float,
472     lambda_bias: float,
473     N_users: int,
474     M_movies: int
475 ) -> Tuple[np.ndarray, np.ndarray]:
476     """Updates user and movie biases based on current residuals."""
477     new_user_bias = np.zeros_like(user_bias)
478     new_movie_bias = np.zeros_like(movie_bias)

```



```

479 user_counts = np.zeros_like(user_bias)
480 movie_counts = np.zeros_like(movie_bias)
481
482 # Calculate residuals:  $r_{ij} - \mu - U_i^T W_j$ 
483 rows, cols, vals_centered = R_train_coo.row, R_train_coo.col, R_train_coo.data
484 dot_prods = np.array([np.dot(U[r, :], W[:, c]) for r, c in zip(rows, cols)], dtype=np.float64)
485 residuals = vals_centered - dot_prods # Residual =  $(r_{ij} - \mu) - U_i^T W_j$ 
486
487 # Update user biases:  $b_u = \text{sum}(\text{residual} - b_i) / (\text{count} + \text{lambda\_bias})$ 
488 np.add.at(new_user_bias, cols, residuals - movie_bias[rows])
489 np.add.at(user_counts, cols, 1)
490 new_user_bias = new_user_bias / (user_counts + lambda_bias + 1e-9) # Add epsilon for stability
491
492 # Update movie biases:  $b_i = \text{sum}(\text{residual} - b_u) / (\text{count} + \text{lambda\_bias})$ 
493 np.add.at(new_movie_bias, rows, residuals - new_user_bias[cols]) # Use updated user bias
494 np.add.at(movie_counts, rows, 1)
495 new_movie_bias = new_movie_bias / (movie_counts + lambda_bias + 1e-9) # Add epsilon for stability
496
497 return new_user_bias.astype(np.float64), new_movie_bias.astype(np.float64)
498
499 def run_als_with_biases(
500     R_train_coo: sparse.coo_matrix, # Centered ratings
501     global_mean: float,
502     probe_users_mapped: np.ndarray,
503     probe_movies_mapped: np.ndarray,
504     probe_ratings_true: np.ndarray,
505     N_users_active: int,
506     M_movies_active: int,
507     rank_local: int,
508     n_iters: int, # Max iterations
509     lam_sq: float,
510     lam_bias: float,
511     rng: Generator,
512     init_scale: float = INIT_SCALE_NON_CONVEX,
513     tol: float = ALS_TOL
514 ) -> Dict[str, List]:
515     """Runs Alternating Least Squares with biases."""
516     logger.info("Starting ALS Solver with Biases...")
517     U, W, user_bias, movie_bias = initialize_factors_and_biases(
518         M_movies_active, N_users_active, rank_local, rng, init_scale
519     )
520
521     hist_loss = [] # Loss not typically tracked directly in ALS, focus on RMSE
522     hist_rmse = []
523     hist_time = []
524
525     start_time = time.time()
526     last_rmse = np.inf
527
528     # Precompute sparse matrix formats for efficiency
529     R_train_csc = R_train_coo.tocsc()
530     R_train_csr = R_train_coo.tocsr()
531
532     for k_iter in range(1, n_iters + 1):
533         iter_start_time = time.time()
534         logger.info(f"--- Starting ALS Iteration {k_iter:02d} ---")
535
536         # Update user factors (W)
537         logger.debug(f"Iter {k_iter}: Updating user factors (W)...")
538         W = update_user_factors(R_train_csc, U, user_bias, movie_bias, lam_sq, rank_local, N_users_active)
539
540         # Update movie factors (U)
541         logger.debug(f"Iter {k_iter}: Updating movie factors (U)...")
542         U = update_movie_factors(R_train_csr, W, user_bias, movie_bias, lam_sq, rank_local, M_movies_active)
543
544         # Update biases
545         logger.debug(f"Iter {k_iter}: Updating biases...")
546         user_bias, movie_bias = update_biases(R_train_coo, U, W, user_bias, movie_bias, global_mean, lam_bias, N_users_active, M_movies_active)
547
548         # Evaluate RMSE
549         logger.debug(f"Iter {k_iter}: Evaluating RMSE...")
550         current_rmse = evaluate_rmse_with_biases(
551             U, W, user_bias, movie_bias, global_mean,
552             probe_users_mapped, probe_movies_mapped, probe_ratings_true
553         )
554         current_time = time.time() - start_time
555         hist_rmse.append(current_rmse)

```

```

556     hist_time.append(current_time)
557
558     iter_time = time.time() - iter_start_time
559     logger.info(f"Iter {k_iter:02d}: RMSE = {current_rmse:.6f} (Time: {iter_time:.2f}s)")
560
561     # Check convergence
562     if abs(last_rmse - current_rmse) < tol:
563         logger.info(f"ALS converged at iteration {k_iter} (RMSE change < {tol})")
564         break
565     last_rmse = current_rmse
566
567     logger.info("ALS Solver with Biases Finished.")
568     return {
569         'loss': [], # ALS doesn't typically track the combined loss easily
570         'rmse': hist_rmse,
571         'time': hist_time,
572         'U': U, 'W': W, 'bu': user_bias, 'bi': movie_bias
573     }
574
575 # --- Stochastic Gradient Single User (NEW - for SARAH/SPIDER) ---
576 def stochastic_gradient_single_user(U, user_idx, N_users, N_movies, loss_args):
577     """ Computes the UNSCALED gradient component d L_user_idx / dU for a single user. """
578     # Unpack loss_args (assumes structure matches loss_and_grad_serial_with_biases)
579     global_mean, rows_idx, cols_idx, vals_true_centered, _, _, rank_func, lambda_sq_func, lambda_bias_func = loss_args
580     M, R = U.shape
581     G_user = np.zeros_like(U, dtype=np.float32)
582     if user_idx not in user_data_arrays: return G_user # Use precomputed user_data_arrays
583
584     W_user_dict = W_closed_efficient(U, N_users, N_movies, user_indices=[user_idx]) # Recompute W for this user
585     if user_idx not in W_user_dict: return G_user
586
587     w_u = W_user_dict[user_idx]
588     user_data = user_data_arrays[user_idx]
589     movie_indices = user_data['movies']; rs_t = user_data['rs'] # rs_t are original ratings here
590     if movie_indices.size == 0: return G_user
591     if movie_indices.max() >= M or movie_indices.min() < 0: return G_user # Return zero grad if invalid index
592
593     # Need centered ratings and biases for gradient calculation
594     # Recompute biases? Or assume they are passed implicitly? Assume passed via loss_args implicitly (not ideal)
595     # This function signature needs alignment with how biases are handled if used by SARAH/SPIDER
596     # For now, approximate using centered ratings and current factors
597     # This needs refinement if SARAH/SPIDER are primary focus
598     ratings_centered_user = rs_t - global_mean # Approximate centering
599
600     U_k = U[movie_indices, :]
601     # Need bias terms here for correct error calculation
602     # Placeholder: Calculate error without biases for now
603     preds_k_dot = U_k @ w_u
604     err_k = preds_k_dot - ratings_centered_user # Error against centered rating
605
606     grad_vals_k = err_k # Simplified grad without prox term from loss_and_grad
607     term_k = grad_vals_k.reshape(-1, 1) * w_u.reshape(1, -1)
608     np.add.at(G_user, movie_indices, term_k.astype(np.float32))
609     # Add regularization gradient for U rows involved
610     G_user[movie_indices, :] += lambda_sq_func * U_k
611
612     if not np.isfinite(G_user).all():
613         G_user = np.nan_to_num(G_user, nan=0.0, posinf=0.0, neginf=0.0)
614     assert G_user.shape == U.shape
615     return G_user
616 # --- Euclidean GD Solver (NEW from long.txt, adapted for biases) ---
617 def run_euclidean_gd(
618     R_train_coo, global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true,
619     N_users_active, M_movies_active, rank_local, n_iters,
620     lam_sq, lam_bias, rng, init_scale=INIT_SCALE_NON_CONVEX, lr=1e-7 # Use specific LR
621 ) -> Dict[str, List]:
622     """Runs Vanilla Euclidean GD with biases."""
623     if RANK_MPI == 0: logger.info(f"\n+++ Running Vanilla Euclidean GD (LR={lr:.1e}) +++")
624     U_euc, W_euc, user_bias, movie_bias = initialize_factors_and_biases(M_movies_active, N_users_active, rank_local, rng, init_scale)
625     # Note: Euclidean GD doesn't require U to be orthonormal, so we use the direct output
626
627     hist_loss, hist_grad, hist_rmse, hist_time = [], [], [], []; t_start = time.time();
628     loss_args_biased = (global_mean, R_train_coo.row, R_train_coo.col, R_train_coo.data, M_movies_active, N_users_active, rank_local, 1
629     eval_args_biased = (global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true)
630
631     try:
632         current_loss, current_rmse, gU_k, gW_k, gBu_k, gBi_k = record_initial_state_biased(U_euc, W_euc, user_bias, movie_bias, loss_ar

```

```

633     grad_norm_k = np.linalg.norm(gU_k) # Use Euclidean norm for U gradient
634 except Exception as e:
635     if RANK_MPI == 0: print(f" ERROR during initial state recording for Euclidean GD: {e}")
636     return {'loss': [], 'grad_norm': [], 'rmse': [], 'time': []}
637
638 if RANK_MPI == 0: hist_loss.append(current_loss); hist_grad.append(grad_norm_k); hist_rmse.append(current_rmse); hist_time.append(t
639
640 if RANK_MPI == 0: logger.info("\n Starting Euclidean GD iterations...")
641 for k in range(n_iters):
642     iter_t0 = time.time();
643     # --- inside your Euclidean-GD loop ---
644     if grad_norm_k < 1e-6:
645         if RANK_MPI == 0:
646             logger.info(f"EucGD converged at iter {k}") # or print(...)
647         break
648
649
650     # Simple Euclidean gradient step for all variables
651     U_euc -= lr * gU_k
652     W_euc -= lr * gW_k
653     user_bias -= lr * gBu_k
654     movie_bias -= lr * gBi_k
655
656     if not (np.isfinite(U_euc).all() and np.isfinite(W_euc).all()):
657         if RANK_MPI == 0: print(f"EucGD Warning: Non-finite factors at iter {k+1}"); break
658
659     try:
660         current_loss, gU_k, gW_k, gBu_k, gBi_k = loss_and_grad_serial_with_biases(U_euc, W_euc, user_bias, movie_bias, *loss_args_b
661         current_rmse = evaluate_rmse_with_biases(U_euc, W_euc, user_bias, movie_bias, *eval_args_biased)
662         grad_norm_k = np.linalg.norm(gU_k) # Euclidean norm
663         if not (np.isfinite(current_loss) and np.isfinite(gU_k).all() and (np.isnan(current_rmse) or np.isfinite(current_rmse))):
664             if RANK_MPI == 0: print(f"EucGD Warning: Non-finite values encountered iter {k+1}.")
665             break
666     except Exception as e:
667         if RANK_MPI == 0: print(f"EucGD Error during iteration {k+1}: {e}")
668         break
669
670     if RANK_MPI == 0:
671         hist_loss.append(current_loss); hist_grad.append(grad_norm_k); hist_rmse.append(current_rmse); hist_time.append(time.time()
672         if k % 5 == 0 or k == n_iters - 1: print(f" EucGD Iter {k+1:02d} | Loss: {current_loss:.3e} | GradNorm: {grad_norm_k:.3e}
673
674 if RANK_MPI == 0: logger.info(f"EucGD finished in {time.time()-t_start:.2f}s");
675 return {'loss': hist_loss, 'grad_norm': hist_grad, 'rmse': hist_rmse, 'time': hist_time, 'U': U_euc, 'W': W_euc, 'bu': user_bias, '
676

```

```

1 all_results = {} #added on 5/6
2
3 # =====
4 # CELL 5: Riemannian Solvers (RGD, RAGD, Catalyst, DANE) - Renumbered
5 # =====
6 logger.info("+++ Cell 5: Defining Riemannian Solvers +++")
7 # --- Stochastic Solvers (SARAH, SPIDER) ---
8
9 def run_soft_impute_efficient(
10     R_train_coo_orig: sparse.coo_matrix, # Original ratings, mapped indices
11     probe_users_mapped: np.ndarray,
12     probe_movies_mapped: np.ndarray,
13     probe_ratings_true: np.ndarray, # Original probe ratings
14     N_users_active: int,
15     M_movies_active: int,
16     n_iters: int,
17     lambda_reg: float,
18     k_rank: int, # Initial rank guess / cap for SVD
19     tol: float,
20     rng: Generator
21 ) -> Dict[str, List]:
22     """ Solves convex problem using efficient Soft-Impute with LinearOperator SVD. """
23     logger.info("Starting Efficient Convex Soft-Impute Solver (CPU)...")
24     use_gpu = False # Force CPU as LinearOperator uses SciPy
25
26     # Prepare necessary sparse formats of original ratings
27     R_orig_csr = R_train_coo_orig.tocsr()
28     R_orig_csc = R_train_coo_orig.tocsc()
29     # Create Omega mask (1s where ratings exist)
30     omega_mask_csr = R_orig_csr.copy(); omega_mask_csr.data[:] = 1
31     omega_mask_csc = omega_mask_csr.tocsc()

```

```

32
33 # Initialize factors U, S, V
34 initial_k = max(1, min(k_rank, M_movies_active, N_users_active))
35 U = rng.standard_normal(size=(M_movies_active, initial_k)).astype(np.float64) * 0.01
36 S = np.zeros(initial_k, dtype=np.float64) # Start with S=0 -> Xk=0 initially
37 V = rng.standard_normal(size=(N_users_active, initial_k)).astype(np.float64) * 0.01
38 if N_users_active >= initial_k: V, _ = np.linalg.qr(V, mode='reduced') # Orthonormalize V initially
39
40 U_old, S_old, V_old = U.copy(), S.copy(), V.copy()
41 hist_loss, hist_rmse, hist_time, hist_rank = [], [], [], []
42 start_time = time.time()
43 current_svd_k = initial_k # Rank for svds call
44
45 for k_iter in range(1, n_iters + 1):
46     iter_start_time = time.time()
47     logger.info(f"--- Starting SoftImpute Iteration {k_iter:02d} ---")
48
49     # Define Linear Operator for Z = P_Omega(R_orig) + P_Omega_Complement(USV^T)
50     Z_op = ImplicitFillOperator(R_orig_csr, R_orig_csc, omega_mask_csr, omega_mask_csc, U, S, V, (M_movies_active, N_users_active))
51
52     # Perform SVD using the LinearOperator
53     logger.debug(f"Iter {k_iter}: Performing SVD with k={current_svd_k}...")
54     svd_start_time = time.time()
55     try:
56         # Ensure k for svds is valid
57         k_svds = max(1, min(current_svd_k, M_movies_active - 1, N_users_active - 1))
58         if k_svds <= 0:
59             logger.warning(f"Iter {k_iter}: Matrix dimensions too small for SVD. Skipping.")
60             rank_k = 0; S_new = np.array([], dtype=np.float64)
61             U_new = np.zeros((M_movies_active, 0), dtype=np.float64)
62             Vt_new = np.zeros((0, N_users_active), dtype=np.float64) # Need Vt shape
63         else:
64             # Use scipy's svds which works with LinearOperator
65             U_new, S_new_raw, Vt_new = svds(Z_op, k=k_svds, which='LM', tol=1e-4, maxiter=100) # Adjust svds tol/maxiter if needed
66
67             # svds returns sorted singular values (largest first) - reverse order
68             S_new_raw = S_new_raw[::-1]
69             U_new = U_new[:, :-1]
70             Vt_new = Vt_new[:-1, :]
71
72             S_new = soft_threshold(S_new_raw, lambda_reg) # Threshold
73             V_new = Vt_new.T # Transpose Vt to get V
74             rank_k = int(np.sum(S_new > 1e-10))
75
76             logger.debug(f"Iter {k_iter}: SVD finished in {time.time() - svd_start_time:.2f}s. Rank after thresholding: {rank_k}")
77
78             if rank_k == 0:
79                 logger.warning(f"Iter {k_iter}: Rank became zero. Resetting.")
80                 current_svd_k = 1 # Reset k for next SVD
81                 U = np.zeros((M_movies_active, 1), dtype=np.float64)
82                 S = np.zeros(1, dtype=np.float64)
83                 V = np.zeros((N_users_active, 1), dtype=np.float64)
84             else:
85                 U = U_new[:, :rank_k].copy()
86                 S = S_new[:rank_k].copy()
87                 V = V_new[:, :rank_k].copy()
88                 current_svd_k = min(rank_k + 5, CONVEX_RANK_K) # Increase k slightly for next iter, capped
89
90     except Exception as e:
91         logger.error(f"SVD failed during SoftImpute iter {k_iter}: {e}", exc_info=True)
92         break
93
94     # Convergence Check
95     U_diff_norm = np.linalg.norm(U - U_old, 'fro'); S_diff_norm = np.linalg.norm(S - S_old, 'fro'); V_diff_norm = np.linalg.norm(V - V_old, 'fro')
96     U_norm = max(1.0, np.linalg.norm(U_old, 'fro')); S_norm = max(1.0, np.linalg.norm(S_old, 'fro')); V_norm = max(1.0, np.linalg.norm(V_old, 'fro'))
97     relative_diff = max(U_diff_norm / U_norm, S_diff_norm / S_norm, V_diff_norm / V_norm) if U_norm > 0 and S_norm > 0 and V_norm > 0
98     logger.debug(f"Iter {k_iter}: Max Rel Factor Diff={relative_diff:.4e}, Rank={rank_k}")
99
100     # Evaluate Metrics
101     eval_start_time = time.time()
102     try:
103         # Objective: 0.5 * ||P_Omega(X - R_orig)||_F^2 + lambda * ||X||_*
104         rows, cols = R_train_coo_orig.row, R_train_coo_orig.col
105         vals_orig = R_train_coo_orig.data
106         preds_at_omega_k = np.array([np.dot(U[r, :], S * V[c, :]) for r, c in zip(rows, cols)], dtype=np.float64)
107         loss_obs_k = 0.5 * np.sum((preds_at_omega_k - vals_orig)**2)
108         nuclear_norm_k = np.sum(S)

```

```

109         loss_k = loss_obs_k + lambda_reg * nuclear_norm_k
110
111         # RMSE: Predict original scale ratings (USV^T) and compare to true validation ratings
112         dot_prods_probe = np.array([np.dot(U[m, :], S * V[u, :]) for m, u in zip(probe_movies_mapped, probe_users_mapped)], dtype=n
113         preds_probe_clamped = np.clip(dot_prods_probe, 1.0, 5.0) # Clamp prediction
114         valid_true_mask_probe = ~np.isnan(ratings_val_true)
115         if np.any(valid_true_mask_probe):
116             mse_probe = np.mean((preds_probe_clamped[valid_true_mask_probe] - ratings_val_true[valid_true_mask_probe])**2)
117             rmse_k = np.sqrt(mse_probe) if mse_probe >= 0 else np.nan
118         else: rmse_k = np.nan
119
120     except Exception as e: logger.error(f"Error during SoftImpute evaluation: {e}"); loss_k, rmse_k, rank_k = np.nan, np.nan, rank_
121
122     eval_time = time.time() - eval_start_time
123     hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time); hist_rank.append(rank_k)
124     U_old, S_old, V_old = U.copy(), S.copy(), V.copy() # Update for next convergence check
125
126     iter_time = time.time() - iter_start_time
127     logger.info(f"Iter {k_iter:02d}: Loss={loss_k:.4e}, RMSE={rmse_k:.4f}, Rank={rank_k}, Rel Diff={relative_diff:.4e} (Eval: {eval
128
129     if relative_diff < tol: logger.info(f"Soft-Impute converged at iteration {k_iter}"); break
130
131     logger.info("Efficient Convex Soft-Impute Solver Finished.")
132     return {'loss': hist_loss, 'rmse': hist_rmse, 'time': hist_time, 'rank': hist_rank, 'U': U, 'S': S, 'V': V}
133
134
135 # --- Stochastic Solvers (SARAH, SPIDER) ---
136 class RiemannianSARAH: # Adapted from long.txt
137     def __init__(self, R, P, g_i, g_batch, batch_size=100, m=1000, eta=1e-3, rng=None):
138         self.R, self.P, self.g_i, self.g_batch = R, P, g_i, g_batch
139         self.B, self.m, self.eta = batch_size, m, eta
140         self.rng = default_rng(rng) if rng is None else rng
141     def run(self, U0, n_steps, grad_args, active_idx, sampling_prob=None):
142         if active_idx is None or len(active_idx) == 0: return U0
143         rng = self.rng; U = U0.copy().astype(np.float32); v = np.zeros_like(U0, dtype=np.float32)
144         U_prev = U.copy().astype(np.float32); num_active = len(active_idx)
145         for t in range(n_steps):
146             if t % self.m == 0:
147                 current_batch_size = min(self.B, num_active);
148                 if current_batch_size == 0: continue
149                 batch_indices = rng.choice(active_idx, size=current_batch_size, p=sampling_prob, replace=True)
150                 try: v = self.g_batch(U, batch_indices, *grad_args).astype(np.float32)
151                 except Exception as e: logger.error(f"SARAH refresh grad error: {e}"); v = np.zeros_like(U)
152                 if not np.isfinite(v).all(): logger.warning(f"SARAH non-finite refresh grad step {t}"); v = np.zeros_like(U)
153             else:
154                 if num_active == 0: continue
155                 i_idx = rng.choice(active_idx, size=1, p=sampling_prob, replace=True)[0]; i = int(i_idx)
156                 try:
157                     v_new = self.g_i(U, i, *grad_args).astype(np.float32)
158                     v_old = self.g_i(U_prev, i, *grad_args).astype(np.float32)
159                     if np.isfinite(v_new).all() and np.isfinite(v_old).all(): v += v_new - v_old
160                 except Exception as e: logger.error(f"SARAH single grad error user {i}: {e}")
161                 G_proj = self.P(U, v); step = (-self.eta * G_proj).astype(np.float32)
162                 if should_stop_subproblem(G_proj, step): break
163                 U_prev = U.copy(); U_next = self.R(U, step)
164                 if not np.isfinite(U_next).all(): logger.warning(f"SARAH non-finite U step {t+1}"); U = U_prev; break
165                 U = U_next
166         return U
167 class RiemannianSPIDER: # Adapted from long.txt
168     def __init__(self, retraction, proj, grad_i, grad_batch, m=100, step=1e-3, rng=None):
169         self.R = retraction; self.P = proj; self.g_i = grad_i; self.g_batch = grad_batch
170         self.m = m; self.eta = step
171         self.rng = default_rng(rng) if rng is None else rng
172     def run(self, U0, n_steps, grad_args, active_idx, sampling_prob=None):
173         if active_idx is None or len(active_idx) == 0: return U0
174         rng = self.rng; U = U0.copy().astype(np.float32); v = np.zeros_like(U0, dtype=np.float32)
175         U_prev = U0.copy().astype(np.float32); num_active = len(active_idx)
176         for t in range(n_steps):
177             if t % self.m == 0:
178                 current_batch_size = min(self.m, num_active); # Use m as batch size for refresh
179                 if current_batch_size == 0: continue
180                 batch_indices = rng.choice(active_idx, size=current_batch_size, p=sampling_prob, replace=True)
181                 try: v = self.g_batch(U, batch_indices, *grad_args).astype(np.float32)
182                 except Exception as e: logger.error(f"SPIDER refresh grad error: {e}"); v = np.zeros_like(U)
183                 if not np.isfinite(v).all(): logger.warning(f"SPIDER non-finite refresh grad step {t}"); v = np.zeros_like(U)
184             else:
185                 if num_active == 0: continue

```

```

186         i_idx = rng.choice(active_idx, size=1, p=sampling_prob, replace=True)[0]; i = int(i_idx)
187         try:
188             grad_new = self.g_i(U, i, *grad_args).astype(np.float32)
189             grad_old = self.g_i(U_prev, i, *grad_args).astype(np.float32)
190             if np.isfinite(grad_new).all() and np.isfinite(grad_old).all(): v = v + grad_new - grad_old
191             except Exception as e: logger.error(f"SPIDER single grad error user {i}: {e}")
192         G_proj = self.P(U, v); step_vec = (-self.eta * G_proj).astype(np.float32)
193         if should_stop_subproblem(G_proj, step_vec): break
194         U_prev = U.copy(); U_next = self.R(U, step_vec)
195         if not np.isfinite(U_next).all(): logger.warning(f"SPIDER non-finite U step {t+1}"); U = U_prev; break
196         U = U_next
197     return U
198 # --- RGD Solver ---
199
200 def run_rgd_with_biases(
201     R_train_coo, global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true,
202     N_users_active, M_movies_active, rank_local, n_iters,
203     lam_sq, lam_bias, rng, init_scale=INIT_SCALE_NON_CONVEX,
204     lr_init=INIT_LR_RIEMANN, ls_beta=LS_BETA, ls_sigma=LS_SIGMA
205 ) -> Dict[str, List]:
206     """Runs Riemannian Gradient Descent with biases."""
207     logger.info("Starting RGD Solver with Biases...")
208     U, W, user_bias, movie_bias = initialize_factors_and_biases(M_movies_active, N_users_active, rank_local, rng, init_scale)
209     hist_loss, hist_rmse, hist_time, hist_grad_norm = [], [], [], []
210     start_time = time.time(); lr_k = lr_init
211     loss_args_biased = (global_mean, R_train_coo.row, R_train_coo.col, R_train_coo.data, M_movies_active, N_users_active, rank_local, 1
212     eval_args_biased = (global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true)
213     try:
214         loss_k, rmse_k, gU_k, gW_k, gBu_k, gBi_k = record_initial_state_biased(U, W, user_bias, movie_bias, loss_args_biased, eval_args
215         hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time)
216         gU_proj_k = ProjTangent(U, gU_k); hist_grad_norm.append(np.linalg.norm(gU_proj_k))
217     except Exception as e: logger.error(f"RGD Init Error: {e}"); return {'loss': [], 'rmse': [], 'time': [], 'grad_norm': []}
218
219     for k in range(n_iters):
220         iter_start_time = time.time()
221         gU_proj_k = ProjTangent(U, gU_k)
222         grad_norm_k = np.linalg.norm(gU_proj_k)
223         hist_grad_norm.append(grad_norm_k)
224
225         # --- FIX: Check Riemannian Gradient Norm ---
226         if grad_norm_k < 1e-6: logger.info("RGD Converged (grad norm)"); break
227         # -----
228
229         ls_loss_args = (W, user_bias, movie_bias) + loss_args_biased
230         lr_step, U_next, loss_next = ArmijoLineSearchRiemannian(U, gU_k, ls_loss_args, loss_k, lr_k, ls_beta, ls_sigma)
231         if lr_step == 0.0: logger.warning("RGD Line search failed."); break
232
233         lr_fixed_other = 1e-4
234         W -= lr_fixed_other * gW_k; user_bias -= lr_fixed_other * gBu_k; movie_bias -= lr_fixed_other * gBi_k
235         U = U_next; loss_k = loss_next
236         lr_k = min(lr_step / np.sqrt(ls_beta), lr_init * 2)
237
238         _, gU_k, gW_k, gBu_k, gBi_k = loss_and_grad_serial_with_biases(U, W, user_bias, movie_bias, *loss_args_biased)
239         rmse_k = evaluate_rmse_with_biases(U, W, user_bias, movie_bias, *eval_args_biased)
240         hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time)
241         iter_time = time.time() - iter_start_time
242         logger.info(f"Iter {k+1:02d}: Loss={loss_k:.4e}, RMSE={rmse_k:.4f}, GradNorm={grad_norm_k:.2e}, LR={lr_step:.2e} (Time: {iter_t
243
244     logger.info("RGD Solver Finished.")
245
246 # --- RAGD Solver ---
247
248 #
249 # --- RAGD Solver ---
250 def run_ragd_with_biases(
251     R_train_coo, global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true,
252     N_users_active, M_movies_active, rank_local, n_iters,
253     lam_sq, lam_bias, rng, init_scale=INIT_SCALE_NON_CONVEX,
254     lr_init=INIT_LR_RIEMANN, ls_beta=LS_BETA, ls_sigma=LS_SIGMA,
255     gamma=RAGD_GAMMA, mu=RAGD_MU, beta_ragd=RAGD_BETA
256 ) -> Dict[str, List]:
257     """Runs Riemannian Accelerated Gradient Descent with biases."""
258     logger.info("Starting RAGD Solver with Biases...")
259     U_k, W_k, user_bias_k, movie_bias_k = initialize_factors_and_biases(M_movies_active, N_users_active, rank_local, rng, init_scale)
260     nu_k = U_k.copy() # Momentum state
261     gamma_k = gamma
262     min_lambda_k = lr_init

```

```

263
264 hist_loss, hist_rmse, hist_time, hist_grad_norm = [], [], [], []
265 start_time = time.time()
266
267 loss_args_biased = (global_mean, R_train_coo.row, R_train_coo.col, R_train_coo.data, M_movies_active, N_users_active, rank_local, l
268 eval_args_biased = (global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true)
269
270 try:
271     loss_k, rmse_k, gU_k, gW_k, gBu_k, gBi_k = record_initial_state_biased(U_k, W_k, user_bias_k, movie_bias_k, loss_args_biased, e
272     hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time)
273     gU_proj_k = ProjTangent(U_k, gU_k); hist_grad_norm.append(np.linalg.norm(gU_proj_k))
274 except Exception as e: logger.error(f"RAGD Init Error: {e}"); return {'loss': [], 'rmse': [], 'time': [], 'grad_norm': []}
275
276 def solve_alpha_eqn(current_min_lambda, gamma, mu):
277     a = 1.0; b = current_min_lambda * (gamma - mu); c = -current_min_lambda * gamma
278     delta = b**2 - 4*a*c
279     if delta < 0: return 0.0
280     alpha1 = (-b + np.sqrt(delta))/(2*a); alpha2 = (-b - np.sqrt(delta))/(2*a)
281     if 0 < alpha1 < 1: return alpha1
282     if 0 < alpha2 < 1: return alpha2
283     return 0.0
284
285 for k in range(n_iters):
286     iter_start_time = time.time()
287     logger.info(f"--- Starting RAGD Iteration {k+1:02d} ---")
288
289     alpha = solve_alpha_eqn(min_lambda_k, gamma_k, mu)
290     if alpha == 0.0: alpha = 1e-6 # Avoid division by zero / stagnation
291     gamma_bar = (1 - alpha) * gamma_k + alpha * mu
292     if gamma_bar == 0.0: gamma_bar = 1e-6
293
294     # Extrapolation step for y_t (only on U)
295     logmap_nu_theta = LogMapApprox(U_k, nu_k)
296     y_t = OrthRetraction(U_k, (alpha * gamma_k / gamma_bar) * logmap_nu_theta)
297
298     # Gradient at y_t (need W and biases at y_t? Assume they stay at k for simplicity)
299     loss_yt, gU_yt, gW_yt, gBu_yt, gBi_yt = loss_and_grad_serial_with_biases(
300         y_t, W_k, user_bias_k, movie_bias_k, *loss_args_biased
301     )
302
303     # Line search from y_t to find theta_{k+1} (U_{k+1})
304     ls_loss_args = (W_k, user_bias_k, movie_bias_k) + loss_args_biased
305     lr_step, U_kp1, loss_kp1 = ArmijoLineSearchRiemannian(
306         y_t, gU_yt, ls_loss_args, loss_yt, min_lambda_k, ls_beta, ls_sigma
307     )
308
309     if lr_step == 0.0: logger.warning("RAGD Line search failed."); break
310     min_lambda_k = lr_step # Update min LR found
311
312     # Update nu (momentum state)
313     logmap_nu_yt = LogMapApprox(y_t, nu_k)
314     grad_proj_yt = ProjTangent(y_t, gU_yt)
315     nu_update_vec = ((1 - alpha) * gamma_k / gamma_bar) * logmap_nu_yt - (alpha / gamma_bar) * grad_proj_yt
316     nu_kp1 = OrthRetraction(y_t, nu_update_vec)
317
318     # Update W and biases (simple gradient step with decayed LR for stability)
319     lr_fixed_other = 1e-4 * (0.9**k) # Use a small decaying LR
320     W_kp1 = W_k - lr_fixed_other * gW_k
321     user_bias_kp1 = user_bias_k - lr_fixed_other * gBu_k
322     movie_bias_kp1 = movie_bias_k - lr_fixed_other * gBi_k
323
324     # Update state
325     U_k, W_k, user_bias_k, movie_bias_k = U_kp1, W_kp1, user_bias_kp1, movie_bias_kp1
326     nu_k = nu_kp1
327     gamma_k = gamma_bar / (1 + beta_ragd) # Update gamma
328     loss_k = loss_kp1
329
330     # Evaluate and record
331     rmse_k = evaluate_rmse_with_biases(U_k, W_k, user_bias_k, movie_bias_k, *eval_args_biased)
332     # Recompute gradient at the final point U_k for norm calculation
333     _, gU_k_final, gW_k, gBu_k, gBi_k = loss_and_grad_serial_with_biases(U_k, W_k, user_bias_k, movie_bias_k, *loss_args_biased)
334     gU_proj_k = ProjTangent(U_k, gU_k_final)
335     grad_norm_k = np.linalg.norm(gU_proj_k)
336
337     hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time)
338     hist_grad_norm.append(grad_norm_k)
339

```

```

340     iter_time = time.time() - iter_start_time
341     logger.info(f"Iter {k+1:02d}: Loss={loss_k:.4e}, RMSE={rmse_k:.4f}, GradNorm={grad_norm_k:.2e}, LR={lr_step:.2e} (Time: {iter_t
342
343     if grad_norm_k < 1e-6: logger.info("RAGD Converged (grad norm)"); break
344
345     logger.info("RAGD Solver Finished.")
346     return {'loss': hist_loss, 'rmse': hist_rmse, 'time': hist_time, 'grad_norm': hist_grad_norm, 'U': U_k, 'W': W_k, 'bu': user_bias_k
347
348 # --- Catalyst Solver ---
349 # --- Catalyst Solver (Modified for Stochastic Inner Solvers) ---
350 def run_catalyst_stochastic( # Renamed from run_catalyst_phi2_with_biases
351     R_train_coo, global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true,
352     N_users_active, M_movies_active, rank_local, n_iters,
353     lam_sq, lam_bias, rng, init_scale=INIT_SCALE_NON_CONVEX,
354     lr_init=INIT_LR_RIEMANN, ls_beta=LS_BETA, ls_sigma=LS_SIGMA,
355     kappa_0=KAPPA_0, kappa_cvx=KAPPA_CVX, inner_T_epochs=CATALYST_INNER_T_EPOCHS,
356     inner_S_epochs_base=CATALYST_INNER_S_EPOCHS_BASE,
357     max_kappa_doublings=MAX_KAPPA_DOUBLINGS,
358     inner_solver_type=INNER_SOLVER, # NEW: Specify inner solver
359     inner_solver_lr = RSVRG_LR, # NEW: LR for stochastic inner solver
360     inner_solver_bs = RSVRG_BATCH_SIZE # NEW: Batch size for stochastic inner solver
361 ) -> Dict[str, List]:
362     """Runs Catalyst-Phi2 using a specified stochastic Riemannian solver."""
363     solver_name = inner_solver_type.upper()
364     logger.info(f"Starting Catalyst-Phi2 + {solver_name} Solver with Biases...")
365     theta_k, W_k, user_bias_k, movie_bias_k = initialize_factors_and_biases(M_movies_active, N_users_active, rank_local, rng, init_scal
366     theta_km1 = theta_k.copy(); tilde_theta_km1 = theta_k.copy()
367     alpha_k = 1.0; kappa_k = kappa_0
368     hist_loss, hist_rmse, hist_time, hist_grad_norm = [], [], [], []
369     phil_grad_hist, phil_dist_hist = [], [] # Rank 0 diagnostics
370     start_time = time.time()
371     loss_args_biased = (global_mean, R_train_coo.row, R_train_coo.col, R_train_coo.data, M_movies_active, N_users_active, rank_local, 1
372     eval_args_biased = (global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true)
373     grad_args_stoch = (N_users_active, M_movies_active, loss_args_biased) # Args for stochastic grad funcs
374     n_data = R_train_coo.nnz # Use number of ratings for epoch length calculation? Or users? Use users.
375     n_active_users = N_users_active
376     epoch_len_batches = max(1, n_active_users // inner_solver_bs) if n_active_users > 0 else 1
377     theta_tilde_k = None #added here on 5/5/2025
378     try:
379         loss_k, rmse_k, gU_k, gW_k, gBu_k, gBi_k = record_initial_state_biased(theta_k, W_k, user_bias_k, movie_bias_k, loss_args_biase
380         hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time)
381         gU_proj_k = ProjTangent(theta_k, gU_k); hist_grad_norm.append(np.linalg.norm(gU_proj_k))
382     except Exception as e: logger.error(f"Catalyst-{solver_name} Init Error: {e}"); return {'loss': [], 'rmse': [], 'time': [], 'grad_n
383
384     # Instantiate selected inner solver (consistent across ranks)
385     inner_solver_instance = None
386     refresh_period_m = max(1, epoch_len_batches // 2) # Example refresh period
387     solver_args_inner = {
388         'R': R_fn, 'P': ProjTangent, 'eta': inner_solver_lr,
389         'g_i': stochastic_gradient_single_user, 'g_batch': stochastic_gradient_batch,
390         #'g_batch': stochastic_gradient_batch, # now resolved 5/4/2025
391         'rng': default_rng(SEED + 1 + RANK_MPI) # Ensure different RNG streams per rank
392     }
393     if inner_solver_type == "sarah": InnerSolverClass = RiemannianSARAH; solver_args_inner.update({'batch_size': inner_solver_bs, 'm':
394     elif inner_solver_type == "spider": InnerSolverClass = RiemannianSPIDER; solver_args_inner.update({'m': refresh_period_m})
395     elif inner_solver_type == "svrg": InnerSolverClass = None # SVRG logic remains embedded
396     else: raise ValueError(f"Unknown INNER_SOLVER: {inner_solver_type}")
397     if InnerSolverClass: inner_solver_instance = InnerSolverClass(**solver_args_inner)
398
399
400     theta_tilde_k : Optional[np.ndarray] = None # <-- avoids UnboundLocalError #added on 5/5/2025
401     for k in range(1, n_iters + 1):
402         iter_start_time = time.time()
403         logger.info(f"--- Starting Catalyst-{solver_name} Iteration {k:02d} ---")
404         kappa_step1 = kappa_k; doubling_count = 0
405         inner_T_steps_budget = epoch_len_batches * inner_T_epochs # Steps budget
406
407         logger.debug(f"Iter {k}: Running Phi1 (kappa adaptation)...")
408         while True:
409             prox_center = theta_km1.copy()
410             # --- Run Inner Solver for Step 1 ---
411             U_inner1 = None
412             if InnerSolverClass:
413                 try:
414                     logger.warning(f"Running inner {solver_name} on f, not h_kappa in Phi1.")
415                     solver_args_run = (grad_args_stoch, unique_users_train, sampling_prob) # Pass active user IDs
416                     U_inner1 = inner_solver_instance.run(prox_center, inner_T_steps_budget, *solver_args_run)

```



```

417         except Exception as e_inner: logger.error(f"Inner {solver_name} (Step 1) failed: {e_inner}"); U_inner1 = prox_center
418     else: # Embedded SVRG for Step 1 subproblem
419         U_snapshot = prox_center.copy()
420         G_full_snapshot = np.zeros_like(U_snapshot) # Calculate full gradient estimate
421         if n_active_users > 0:
422             num_batches_for_full_grad = max(1, math.ceil(n_active_users / inner_solver_bs / 5))
423             count_full = 0
424             for _ in range(num_batches_for_full_grad):
425                 current_batch_size = min(inner_solver_bs, n_active_users)
426                 if current_batch_size == 0: continue
427                 batch_ids_full = GLOBAL_RNG.choice(unique_users_train, size=current_batch_size, p=sampling_prob, replace=True)
428                 try: G_batch = stochastic_gradient_batch(U_snapshot, batch_ids_full, *grad_args_stoch);
429                 except Exception: continue
430                 if np.isfinite(G_batch).all(): G_full_snapshot += G_batch; count_full += 1
431             if count_full > 0: G_full_snapshot /= count_full
432         U_inner1_svrg = U_snapshot.copy();
433         for i_t in range(inner_T_steps_budget):
434             current_batch_size = min(inner_solver_bs, n_active_users)
435             if current_batch_size == 0: break
436             batch_ids = GLOBAL_RNG.choice(unique_users_train, size=current_batch_size, p=sampling_prob, replace=True)
437             try: g_curr = stochastic_gradient_batch(U_inner1_svrg, batch_ids, *grad_args_stoch); g_ref = stochastic_gradient_b
438             except Exception: g_curr = np.zeros_like(U_inner1_svrg); g_ref = np.zeros_like(U_inner1_svrg)
439             if not (np.isfinite(g_curr).all() and np.isfinite(g_ref).all()): continue
440             G_vr_f = g_curr - g_ref + G_full_snapshot
441             if REG_DISTANCE == "euclid": G_prox_term = kappa_step1 * (U_inner1_svrg - prox_center);
442             else: G_prox_term = - kappa_step1 * LogMapApprox(U_inner1_svrg, prox_center)
443             subprob_G_vr_euclidean = G_vr_f + G_prox_term
444             G_proj_vr = ProjTangent(U_inner1_svrg, subprob_G_vr_euclidean)
445             step_vec = (-inner_solver_lr * G_proj_vr).astype(np.float32)
446             if should_stop_subproblem(G_proj_vr, step_vec): break
447             U_next_svrg = R_fn(U_inner1_svrg, step_vec)
448             if not np.isfinite(U_next_svrg).all(): break
449             U_inner1_svrg = U_next_svrg
450         U_inner1 = U_inner1_svrg
451
452     # --- Check conditions after inner solve ---
453     theta_bar_k_T = U_inner1;
454     try: loss_bar_k_T, G_bar_k_T = loss_and_grad_corrected(theta_bar_k_T, W_k, user_bias_k, movie_bias_k, *loss_args_biased)
455     except Exception as e: logger.error(f"Error evaluating bar_theta: {e}"); loss_bar_k_T = np.inf
456     if not np.isfinite(loss_bar_k_T): kappa_step1 *= 2; doubling_count += 1; continue
457     conditions_met = False; phil_grad_norm = np.nan; d_R_approx = np.nan
458     if RANK_MPI == 0: # Only rank 0 checks conditions
459         d_R_approx = np.linalg.norm(LogMapApprox(theta_km1, theta_bar_k_T));
460         h_k_bar = loss_bar_k_T + 0.5 * kappa_step1 * d_R_approx**2;
461         loss_km1 = hist_loss[-1] if hist_loss else np.inf
462         descent_cond_met = (h_k_bar <= loss_km1 + 1e-9 * (1 + abs(loss_km1)))
463         if REG_DISTANCE == "euclid": subprob_grad_bar_k = G_bar_k_T + kappa_step1 * (theta_bar_k_T - theta_km1);
464         else: subprob_grad_bar_k = G_bar_k_T - kappa_step1 * LogMapApprox(theta_bar_k_T, theta_km1)
465         proj_grad_h = ProjTangent(theta_bar_k_T, subprob_grad_bar_k)
466         phil_grad_norm = np.linalg.norm(proj_grad_h)
467         stationarity_rhs = kappa_step1 * d_R_approx
468         stat_cond_met = phil_grad_norm <= stationarity_rhs + 1e-9 * (1 + stationarity_rhs)
469         if descent_cond_met and stat_cond_met:
470             print(f"      Alg phi_1 Conditions MET kappa={kappa_step1:.1e}")
471             phil_grad_hist.append(phil_grad_norm); phil_dist_hist.append(d_R_approx)
472             kappa_k_next = update_kappa_adaptive(kappa_step1, phil_grad_hist, phil_dist_hist, theta_bar_k_T)
473             if abs(kappa_k_next - kappa_step1) > 1e-9: print(f"      Adapting kappa next iter: {kappa_step1:.1e} -> {kappa_k_ne
474             kappa_k = kappa_k_next
475             conditions_met = True
476         else: print(f"      Alg phi_1 Conditions NOT MET (Desc:{descent_cond_met}, Stat:{stat_cond_met}) kappa={kappa_step1:.1e
477     if COMM and SIZE_MPI > 1: conditions_met = COMM.bcast(conditions_met, root=0); kappa_k = COMM.bcast(kappa_k, root=0) if con
478     if conditions_met: break
479     else:
480         kappa_step1 *= 2; doubling_count += 1;
481         if doubling_count >= MAX_KAPPA_DOUBLINGS: logger.warning("Phi1 max kappa doublings reached."); break
482     if doubling_count >= MAX_KAPPA_DOUBLINGS: logger.error(f"Catalyst Iter {k}: Phi1 failed. Stopping."); break
483     bar_theta_k = theta_bar_k_T; loss_bar_k = loss_bar_k_T; G_bar_k = G_bar_k_T; kappa_k = kappa_step1
484     logger.debug(f"Iter {k}: Phi1 finished. Final kappa={kappa_k:.2e}")
485
486     # === Step 2: Extrapolation ===
487     if k == 1: V_extrap_approx = np.zeros_like(theta_km1)
488     else: V_extrap_approx = LogMapApprox(theta_km1, tilde_theta_km1)
489     vartheta_k = R_fn(theta_km1, alpha_k * V_extrap_approx);
490     if not np.isfinite(vartheta_k).all(): logger.error(f"Step 2 non-finite iter {k}. Stopping."); break
491
492     # === Step 3: Accelerated Step (using chosen solver) ===
493     logger.debug(f"Iter {k}: Running Phi2 (accelerated step)...")

```

```

494 prox_center_S = vartheta_k.copy()
495 S_k_epochs = math.ceil(inner_S_epochs_base * math.log(k + 1))
496 max_inner_iter_2 = S_k_epochs * epoch_len_batches
497
498 #theta_tilde_k = None
499 if InnerSolverClass:
500     try:
501         logger.warning(f"Running inner {solver_name} on f, not h_kappa_cvx in Phi2.")
502         solver_args_run_S = (grad_args_stoch, unique_users_train, sampling_prob)
503         theta_tilde_k = inner_solver_instance.run(prox_center_S, max_inner_iter_2, *solver_args_run_S)
504     except Exception as e_inner_S: logger.error(f"Inner {solver_name} (Step 3) failed: {e_inner_S}"); theta_tilde_k = prox_cen
505 else: # Embedded SVRG for Step 3 subproblem
506     U_snapshot_S = prox_center_S
507     G_full_snapshot_S = np.zeros_like(U_snapshot_S) # Calculate full gradient estimate
508     if n_active_users > 0:
509         num_batches_for_full_grad_S = max(1, math.ceil(n_active_users / inner_solver_bs / 5))
510         count_S_full = 0
511         for _ in range(num_batches_for_full_grad_S):
512             current_batch_size_S = min(inner_solver_bs, n_active_users)
513             if current_batch_size_S == 0: continue
514             batch_ids_full_S = GLOBAL_RNG.choice(unique_users_train, size=current_batch_size_S, p=sampling_prob, replace=True)
515             try: G_batch_S = stochastic_gradient_batch(U_snapshot_S, batch_ids_full_S, *grad_args_stoch);
516             except Exception: continue
517             if np.isfinite(G_batch_S).all(): G_full_snapshot_S += G_batch_S; count_S_full += 1
518         if count_S_full > 0: G_full_snapshot_S /= count_S_full
519     U_inner2_svrg = U_snapshot_S.copy();
520     for i_s in range(max_inner_iter_2):
521         current_batch_size_S = min(inner_solver_bs, n_active_users)
522         if current_batch_size_S == 0: break
523         batch_ids_S = GLOBAL_RNG.choice(unique_users_train, size=current_batch_size_S, p=sampling_prob, replace=True)
524         try: g_curr_S = stochastic_gradient_batch(U_inner2_svrg, batch_ids_S, *grad_args_stoch); g_ref_S = stochastic_gradien
525         except Exception: g_curr_S = np.zeros_like(U_inner2_svrg); g_ref_S = np.zeros_like(U_inner2_svrg)
526         if not (np.isfinite(g_curr_S).all() and np.isfinite(g_ref_S).all()): continue
527         G_vr_f_S = g_curr_S - g_ref_S + G_full_snapshot_S
528         if REG_DISTANCE == "euclid": G_prox_term_S = KAPPA_CVX * (U_inner2_svrg - prox_center_S);
529         else: G_prox_term_S = - KAPPA_CVX * LogMapApprox(U_inner2_svrg, prox_center_S)
530         subprob_G_vr_euclidean_S = G_vr_f_S + G_prox_term_S
531         G_proj_vr_S = ProjTangent(U_inner2_svrg, subprob_G_vr_euclidean_S)
532         step_vec_S = (-inner_solver_lr * G_proj_vr_S).astype(np.float32)
533         if should_stop_subproblem(G_proj_vr_S, step_vec_S): break
534         U_next_S = R_fn(U_inner2_svrg, step_vec_S)
535         if not np.isfinite(U_next_S).all(): break
536         U_inner2_svrg = U_next_S
537     theta_tilde_k = U_inner2_svrg
538 try: loss_tilde_k, G_tilde_k = loss_and_grad_corrected(
539     theta_tilde_k,
540     W_k,
541     user_bias_k,
542     movie_bias_k,
543     *loss_args_biased
544 )
545 #
546 #try: loss_tilde_k, G_tilde_k = loss_and_grad_corrected(theta_tilde_k, *loss_args);
547 except Exception as e: logger.error(f"Error evaluating tilde_theta: {e}"); loss_tilde_k = np.inf
548 if not (np.isfinite(loss_tilde_k) and np.isfinite(G_tilde_k).all()): logger.error(f"Step 3 ({solver_name}) failed iter {k}. Stc
549
550 # === Step 4, 5, 6 (Consistent) ===
551 if loss_bar_k <= loss_tilde_k: theta_kp1, loss_kp1, G_kp1, selected = theta_bar_k, loss_bar_k, G_bar_k, "bar"
552 else: theta_kp1, loss_kp1, G_kp1, selected = theta_tilde_k, loss_tilde_k, G_tilde_k, "tilde"
553 V_update_approx = LogMapApprox(theta_km1, theta_tilde_k);
554 tilde_theta_k_next = R_fn(theta_km1, (1.0 / alpha_k) * V_update_approx);
555 if not np.isfinite(tilde_theta_k_next).all(): logger.error(f"Step 5 non-finite iter {k}. Stopping."); break
556 alpha_kp1 = (math.sqrt(alpha_k**4 + 4 * alpha_k**2) - alpha_k**2) / 2.0
557
558 # --- Update state for next iteration ---
559 theta_km1 = theta_kp1.copy(); tilde_theta_km1 = tilde_theta_k_next.copy()
560 alpha_k = alpha_kp1; loss_k = loss_kp1
561 lr_fixed_other = 1e-4 * (0.9**k)
562 _, _, gW_kp1, gBu_kp1, gBi_kp1 = loss_and_grad_serial_with_biases(theta_kp1, W_k, user_bias_k, movie_bias_k, *loss_args_biased)
563 W_k -= lr_fixed_other * gW_kp1; user_bias_k -= lr_fixed_other * gBu_kp1; movie_bias_k -= lr_fixed_other * gBi_kp1
564
565 # --- Record History ---
566 rmse_k = evaluate_rmse_with_biases(theta_kp1, W_k, user_bias_k, movie_bias_k, *eval_args_biased)
567 gU_proj_k = ProjTangent(theta_kp1, G_kp1); grad_norm_k = np.linalg.norm(gU_proj_k)
568 hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time); hist_grad_norm.append(grad_norm
569 iter_time = time.time() - iter_start_time
570 logger.info(f"Iter {k:02d}: Loss={loss_k:.4e}, RMSE={rmse_k:.4f}, GradNorm={grad_norm_k:.2e}, Kappa={kappa_k:.2e} (Time: {iter_

```

```

571         if grad_norm_k < 1e-6: logger.info(f"Catalyst-{solver_name} Converged (grad norm)"); break
572
573     logger.info(f"Catalyst-{solver_name} Solver Finished.")
574     if k == n_iters: # Append final grad norm if loop finished normally
575         _, gU_k_final, _, _, _ = loss_and_grad_serial_with_biases(theta_k, W_k, user_bias_k, movie_bias_k, *loss_args_biased)
576         gU_proj_k = ProjTangent(theta_k, gU_k_final); hist_grad_norm.append(np.linalg.norm(gU_proj_k))
577     return {'loss': hist_loss, 'rmse': hist_rmse, 'time': hist_time, 'grad_norm': hist_grad_norm, 'U': theta_k, 'W': W_k, 'bu': user_bi
578
579
580 # --- DANE Solver ---
581
582 # --- DANE Solver ---
583 def run_dane_with_biases(
584     R_train_coo, global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true,
585     N_users_active, M_movies_active, rank_local, n_iters,
586     lam_sq, lam_bias, rng, init_scale=INIT_SCALE_NON_CONVEX,
587     lr_init=INIT_LR_RIEMANN, ls_beta=LS_BETA, ls_sigma=LS_SIGMA,
588     kappa=DANE_KAPPA
589 ) -> Dict[str, List]:
590     """Runs DANE adaptation with biases."""
591     logger.info("Starting DANE Solver with Biases...")
592     theta_k, W_k, user_bias_k, movie_bias_k = initialize_factors_and_biases(M_movies_active, N_users_active, rank_local, rng, init_scal
593     theta_km1 = theta_k.copy()
594
595     hist_loss, hist_rmse, hist_time, hist_grad_norm = [], [], [], []
596     start_time = time.time()
597     lr_k = lr_init
598
599     loss_args_biased = (global_mean, R_train_coo.row, R_train_coo.col, R_train_coo.data, M_movies_active, N_users_active, rank_local, l
600     eval_args_biased = (global_mean, probe_users_mapped, probe_movies_mapped, probe_ratings_true)
601
602     try:
603         loss_k, rmse_k, gU_k, gW_k, gBu_k, gBi_k = record_initial_state_biased(theta_k, W_k, user_bias_k, movie_bias_k, loss_args_biase
604         hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time)
605         gU_proj_k = ProjTangent(theta_k, gU_k); hist_grad_norm.append(np.linalg.norm(gU_proj_k))
606     except Exception as e: logger.error(f"DANE Init Error: {e}"); return {'loss': [], 'rmse': [], 'time': [], 'grad_norm': []}
607
608     for k in range(n_iters):
609         iter_start_time = time.time()
610         logger.info(f"--- Starting DANE Iteration {k+1:02d} ---")
611
612         if k == 0:
613             grad_combined = gU_k # Use initial gradient for first step
614         else:
615             reg_grad = RegularizeGradChordalApprox(theta_k, theta_km1, kappa)
616             grad_combined = gU_k + reg_grad # gU_k is from end of previous iteration
617
618         gU_proj_k = ProjTangent(theta_k, grad_combined)
619         grad_norm_k = np.linalg.norm(gU_proj_k)
620         hist_grad_norm.append(grad_norm_k) # Log norm before step
621
622         if grad_norm_k < 1e-6: logger.info("DANE Converged (grad norm)"); break
623
624         # Line search on U update using combined gradient
625         ls_loss_args = (W_k, user_bias_k, movie_bias_k) + loss_args_biased
626         lr_step, U_kp1, loss_kp1 = ArmijoLineSearchRiemannian(
627             theta_k, grad_combined, ls_loss_args, loss_k, lr_k, ls_beta, ls_sigma
628         )
629
630         if lr_step == 0.0: logger.warning("DANE Line search failed."); break
631
632         # Update W and biases (simple gradient step with decayed LR?)
633         lr_fixed_other = 1e-4 * (0.9**k)
634         W_kp1 = W_k - lr_fixed_other * gW_k
635         user_bias_kp1 = user_bias_k - lr_fixed_other * gBu_k
636         movie_bias_kp1 = movie_bias_k - lr_fixed_other * gBi_k
637
638         # Update state
639         theta_km1 = theta_k.copy() # Store previous U
640         theta_k = U_kp1
641         W_k, user_bias_k, movie_bias_k = W_kp1, user_bias_kp1, movie_bias_kp1
642         loss_k = loss_kp1
643         lr_k = min(lr_step / np.sqrt(ls_beta), lr_init * 2) # Update LR for next search
644
645         # Recompute gradients at new point for next iteration
646         _, gU_k, gW_k, gBu_k, gBi_k = loss_and_grad_serial_with_biases(theta_k, W_k, user_bias_k, movie_bias_k, *loss_args_biased)
647         rmse_k = evaluate_rmse_with_biases(theta_k, W_k, user_bias_k, movie_bias_k, *eval_args_biased)

```

```

648
649     hist_loss.append(loss_k); hist_rmse.append(rmse_k); hist_time.append(time.time() - start_time)
650
651     iter_time = time.time() - iter_start_time
652     logger.info(f"Iter {k+1:02d}: Loss={loss_k:.4e}, RMSE={rmse_k:.4f}, GradNorm={grad_norm_k:.2e}, LR={lr_step:.2e} (Time: {iter_t
653
654     logger.info("DANE Solver Finished.")
655     return {'loss': hist_loss, 'rmse': hist_rmse, 'time': hist_time, 'grad_norm': hist_grad_norm, 'U': theta_k, 'W': W_k, 'bu': user_bi
656 # Create R_mask_coo (local partition mask, COO format)
657 # Using local_user_ids and local_movie_ids
658 '''if local_user_ids.size > 0:
659     R_mask_coo = sparse.coo_matrix(
660         (np.ones_like(local_user_ids, dtype=np.uint8), (local_movie_ids, local_user_ids)), # Value doesn't matter for mask, use 1s
661         shape=(M_movies, N_users), # Use global dimensions for shape
662         dtype=np.uint8 # Use uint8 for mask
663     )
664 else:
665     # Create an empty mask with the correct shape if no local data
666     R_mask_coo = sparse.coo_matrix((M_movies, N_users), dtype=np.uint8)'''
667 def PROJ_TANGENT(U: np.ndarray, G: np.ndarray) -> np.ndarray:
668     """
669     Project G onto the tangent space at U (Grassmann).
670     """
671     return (G - U @ (U.T @ G)).astype(np.float32)
672
673 def RECORDINITIALSTATE(U0: np.ndarray, W0: np.ndarray, L0: float, gU0: np.ndarray, gW0: np.ndarray) -> Dict[str, Any]:
674     """
675     Records the initial state (U0, W0, loss0, gradient norms) for tracking.
676
677     Args:
678         U0 (np.ndarray): Initial U matrix.
679         W0 (np.ndarray): Initial W matrix (global).
680         L0 (float): Initial loss.
681         gU0 (np.ndarray): Initial gradient w.r.t. U (global).
682         gW0 (np.ndarray): Initial gradient w.r.t. W (global).
683
684     Returns:
685         dict: A dictionary containing the initial state information.
686     """
687     d = dict(
688         U0=U0.copy(), # Store copies
689         W0=W0.copy(),
690         loss0=float(L0),
691         gU0_norm=float(np.linalg.norm(gU0)),
692         gW0_norm=float(np.linalg.norm(gW0)),
693         timestamp=time.time(), # Record time
694     )
695     if RANK_MPI == 0:
696         logger.info("--- Initial state ---", extra={"rank": RANK_MPI})
697         logger.info("L0 = %.6e ||∇U||F=%.3e ||∇W||F=%.3e",
698             d['loss0'], d['gU0_norm'], d['gW0_norm'], extra={"rank": RANK_MPI})
699
700     return d
701 def LOSSANDGRAD_TOTAL_DERIVATIVE(
702     U: np.ndarray,
703     X_local: sparse.csc_matrix, # local sparse ratings (M x N) (CSC/CSR/COO)
704     mask_coo_global: sparse.coo_matrix, # Global mask matrix (M x N, uint8) indicating observed entries (COO)
705     N_users: int, # Total number of users globally
706     M_movies: int, # Total number of movies globally
707     *,
708     user_data_override: Optional[Dict[int, Dict[str, np.ndarray]]] = None, # Optional override for user_data_arrays
709     return_W: bool = False, # If True, also returns the local W*(U) matrix
710 ) -> Union[Tuple[float, np.ndarray], Tuple[float, np.ndarray, np.ndarray, np.ndarray]]:
711     """
712     Computes the total profiled loss L(U, W*(U)) and its Euclidean total derivative dL/dU.
713     Solves for W*(U) using a closed-form expression.
714     Optionally returns the local W*(U) and local gradient with respect to W.
715
716     Args:
717         U (np.ndarray): Current movie factor matrix (M x RANK, float64), assumed consistent across ranks.
718         X_local (sparse matrix): The local partition of the training data matrix (M x N, float64).
719         mask_coo_global (sparse.coo_matrix): The GLOBAL mask matrix (M x N, uint8) in COO format, indicating observed entries.
720         N_users (int): Total number of users globally.
721         M_movies (int): Total number of movies globally.
722         user_data_override (dict, optional): Override for user_data_arrays when calling WCLOSEDEFFICIENT.
723         return_W (bool): If True, returns W_local and local_gW0 as well.
724

```

```

725 Returns:
726     If return_W=False:
727         (total_loss, dL_dU)
728     If return_W=True:
729         (total_loss, dL_dU, local_grad_W, W_local)
730     where:
731         total_loss is a float64 scalar,
732         dL_dU is an (M x RANK) float64 array,
733         local_grad_W is an (RANK x N) float64 array,
734         W_local is an (RANK x N) float64 array.
735 """
736 U = U.astype(np.float64, copy=False)
737 M, r = U.shape
738
739 # 1) Solve W*(U) for the local columns only.
740 # WCLOSEDEFFICIENT returns the local part of W*(U), shape (r x N_users).
741 W_local = WCLOSEDEFFICIENT(
742     U=U,
743     N_users=N_users,
744     user_data_override=user_data_override
745 ) # (r x N_users), float64
746
747 # 2) Observed-data term for the local slice
748 local_obs_loss = 0.0
749 local_grad_obs_term_U = np.zeros_like(U, dtype=np.float64)
750 local_grad_obs_term_W = np.zeros_like(W_local, dtype=np.float64)
751
752 # Process only if there are non-zero entries in the local data AND the global mask
753 if X_local.nnz and mask_coo_global.nnz:
754     # Ensure mask is in COO format
755     if not sparse.isspmatrix_coo(mask_coo_global):
756         mask_coo_global = mask_coo_global.tocoo()
757
758     # Filter indices to be within local data matrix bounds
759     r_ok = (mask_coo_global.row < X_local.shape[0]) & (mask_coo_global.row >= 0)
760     c_ok = (mask_coo_global.col < X_local.shape[1]) & (mask_coo_global.col >= 0)
761     sel = r_ok & c_ok
762     rows = mask_coo_global.row[sel]
763     cols = mask_coo_global.col[sel]
764
765     if rows.size:
766         # Get the true ratings from the local training data for these indices
767         R_omega = X_local[rows, cols].A1.astype(np.float64)
768
769         # Create a local mask COO matrix of the same shape as X_local
770         mask_loc = sparse.coo_matrix(
771             (np.ones_like(rows, dtype=np.uint8), (rows, cols)),
772             shape=X_local.shape,
773             dtype=np.uint8,
774         )
775
776         # Compute predictions (U @ W)_omega using the local mask
777         UW_sparse_local = sparse_product(U, W_local, mask_loc)
778         UW_omega = UW_sparse_local.data.astype(np.float64)
779
780         # Filter out non-finite predictions or true values
781         good = np.isfinite(UW_omega) & np.isfinite(R_omega)
782         if not np.all(good):
783             bad = (~good).sum()
784             logger.warning(
785                 "Rank %d: filtered %d non-finite preds/targets in local observed data",
786                 RANK_MPI, bad, extra={"rank": RANK_MPI}
787             )
788             UW_omega = UW_omega[good]
789             R_omega = R_omega[good]
790             rows = rows[good]
791             cols = cols[good]
792
793         if UW_omega.size:
794             err_omega = UW_omega - R_omega
795             local_obs_loss = 0.5 * np.dot(err_omega, err_omega)
796
797         # Gradient contribution wrt U
798         E_coo_local = sparse.coo_matrix((err_omega, (rows, cols)), shape=X_local.shape)
799         local_grad_obs_term_U = E_coo_local @ W_local.T
800
801         # Gradient contribution wrt W

```

```

802         local_grad_obs_term_W = U.T @ E_coo_local.tocsc()
803
804     # 3) Aggregate across ranks (observed loss and gradients)
805     def _allreduce(arr_like, op=MPI.SUM):
806         if COMM and SIZE_MPI > 1:
807             arr_np = np.asarray(arr_like, dtype=np.float64)
808             recv = np.zeros_like(arr_np)
809             COMM.Allreduce(arr_np, recv, op=op)
810             if arr_np.ndim == 0:
811                 return float(recv)
812             return recv
813         # Serial case: no reduction needed
814         if np.isscalar(arr_like):
815             return float(arr_like)
816         return np.asarray(arr_like, dtype=np.float64)
817
818     global_obs_loss = _allreduce(local_obs_loss)
819     global_grad_obs_term_U = _allreduce(local_grad_obs_term_U)
820     global_grad_obs_term_W = _allreduce(local_grad_obs_term_W)
821
822     # 4) Regularization penalties
823     # U is global/identical across ranks
824     U_fro_sq = np.sum(U**2)
825     # W_local is local. Sum local W^2, then allreduce
826     local_W_fro_sq = np.sum(W_local**2)
827     global_W_fro_sq = _allreduce(local_W_fro_sq)
828
829     total_loss = (
830         global_obs_loss
831         + 0.5 * LAM_SQ * U_fro_sq
832         + 0.5 * LAM_SQ * global_W_fro_sq
833     )
834
835     # Total derivative dL/dU for the profiled loss
836     dL_dU = global_grad_obs_term_U + LAM_SQ * U
837
838     # local_grad_obs_term_W is the local gradient wrt W
839     local_gW0 = local_grad_obs_term_W
840
841     # Safety checks
842     if not np.isfinite(total_loss):
843         logger.warning(
844             "Rank %d: Non-finite loss detected; clamped.",
845             RANK_MPI,
846             extra={"rank": RANK_MPI}
847         )
848     total_loss = np.finfo(np.float64).max
849     if not np.isfinite(dL_dU).all():
850         logger.warning(
851             "Rank %d: Non-finite grad(U) detected; zeros injected.",
852             RANK_MPI,
853             extra={"rank": RANK_MPI}
854         )
855     dL_dU = np.nan_to_num(dL_dU, nan=0.0, posinf=0.0, neginf=0.0)
856     if return_W and not np.isfinite(local_gW0).all():
857         logger.warning(
858             "Rank %d: Non-finite local grad(W) detected; zeros injected.",
859             RANK_MPI,
860             extra={"rank": RANK_MPI}
861         )
862     local_gW0 = np.nan_to_num(local_gW0, nan=0.0, posinf=0.0, neginf=0.0)
863     if return_W and not np.isfinite(W_local).all():
864         logger.warning(
865             "Rank %d: Non-finite local W detected; zeros injected.",
866             RANK_MPI,
867             extra={"rank": RANK_MPI}
868         )
869     W_local = np.nan_to_num(W_local, nan=0.0, posinf=0.0, neginf=0.0)
870
871     if return_W:
872         return float(total_loss), dL_dU, local_gW0, W_local
873     else:
874         return float(total_loss), dL_dU

```

1

2 # ===== #

```

3 # CELL 16 - Riemannian SVRG (R-SVRG) Algorithm (Complete and Fixed) #
4 # ===== #
5
6 try:
7     from mpi4py import MPI
8     COMM = MPI.COMM_WORLD
9     RANK_MPI = COMM.Get_rank()
10    SIZE_MPI = COMM.Get_size()
11 except ImportError:
12     COMM = None
13     RANK_MPI = 0
14     SIZE_MPI = 1
15
16 import logging
17 import numpy as np
18 import scipy.sparse as sparse
19 import time
20 import math
21 import gc
22 from typing import Optional, Tuple, Dict, Union, Any, Callable, List
23 from numpy.random import default_rng, Generator
24 import matplotlib.pyplot as plt
25
26 required_functions = [
27     "R_fn", "PROJ_TANGENT", "should_stop_subproblem", "evaluate_rmse_with_biases",
28     "INITIALIZEU", "record_initial_state_biased", "grad_single_user_combined",
29     "grad_batch_users_combined", "full_loss_and_grad_unprofiled",
30     "CombinedGradient", "RiemannianSPIDER", "RiemannianSARAH"
31 ]
32
33 for func_name in required_functions:
34     if func_name not in globals() or not callable(globals()[func_name]):
35         logging.critical(f"Rank {RANK_MPI}: Required function or class '{func_name}' not found.")
36         if COMM and SIZE_MPI > 1:
37             COMM.Abort(1)
38         raise RuntimeError(f"Missing function or class: {func_name}")
39
40 required_globals = [
41     "R_matrix", "R_mask_coo", "Probe_mask_coo", "probe_ratings_true",
42     "probe_movie_ids_final", "probe_user_ids_final", "N_users", "M_movies",
43     "RANK", "N_ITERS", "RSVRG_LR", "RSVRG_BATCH_SIZE", "GLOBAL_RNG", "LAM_SQ",
44     "LAM_BIAS", "user_data_arrays", "active_idx", "sampling_prob",
45     "global_actual_loaded", "global_mean_rating", "user_ids_val_final",
46     "movie_ids_val_final", "ratings_val_true"
47 ]
48
49 for global_name in required_globals:
50     if global_name not in globals():
51         logging.critical(f"Rank {RANK_MPI}: Required global variable '{global_name}' missing.")
52         if COMM and SIZE_MPI > 1:
53             COMM.Abort(1)
54         raise RuntimeError(f"Missing global variable: {global_name}")
55
56 # ----- #
57 # RUN_RSVRG_UNPROFILED is assumed to be defined as in your provided code #
58 # This function includes:
59 # - class CombinedGradient
60 # - grad_single_user_combined
61 # - grad_batch_users_combined
62 # - full_loss_and_grad_unprofiled
63 # - RUN_RSVRG_UNPROFILED main loop
64 # ----- #
65
66 # --- Execute RUN_RSVRG_UNPROFILED and Display Results ---
67 if RANK_MPI == 0:
68     logging.info("\n--- Running Unprofiled Riemannian SVRG ---")
69
70 try:
71     unprofiled_rsvrg_results = RUN_RSVRG_UNPROFILED(
72         user_data_arrays=user_data_arrays,
73         lam_sq=LAM_SQ,
74         lam_bias=LAM_BIAS,
75         total_ratings=global_actual_loaded,
76         M=M_movies_active,
77         r=RANK,
78         N=N_users_active,
79         n_epochs=N_ITERS,

```

```

80     epoch_len=RSVRG_EPOCH_LEN,
81     batch_size=RSVRG_BATCH_SIZE,
82     lr=RSVRG_LR,
83     active_users=active_idx,
84     rng=GLOBAL_RNG,
85     lr_decay_rate=0.95,
86     global_mean=global_mean_rating,
87     probe_users_mapped=user_ids_val_final,
88     probe_movies_mapped=movie_ids_val_final,
89     probe_ratings_true=ratings_val_true
90 )
91
92 if RANK_MPI == 0:
93     logging.info("\n--- Unprofiled R-SVRG Execution Results ---")
94     logging.info("Generating Convergence Plots...")
95
96     # Plot Loss
97     plt.figure(figsize=(10, 6))
98     plt.plot(np.arange(len(unprofiled_rsvrg_results['loss'])),
99             unprofiled_rsvrg_results['loss'],
100            marker="o", linestyle="-", label='Unprofiled R-SVRG')
101     plt.yscale("log")
102     plt.title('Loss Convergence (Unprofiled R-SVRG)')
103     plt.xlabel('Epoch')
104     plt.ylabel('Loss')
105     plt.grid(True)
106     plt.legend()
107     plt.show()
108
109     # Plot Gradient Norm
110     plt.figure(figsize=(10, 6))
111     plt.plot(np.arange(len(unprofiled_rsvrg_results['grad_norm'])),
112            unprofiled_rsvrg_results['grad_norm'],
113            marker="o", linestyle="-", label='||Grad||F')
114     plt.yscale("log")
115     plt.title('Euclidean Gradient Norm Convergence (Unprofiled R-SVRG)')
116     plt.xlabel('Epoch')
117     plt.ylabel('Gradient Norm')
118     plt.grid(True)
119     plt.legend()
120     plt.show()
121
122 except Exception as e:
123     logging.error(f"Error running unprofiled R-SVRG: {e}", exc_info=True)
124     if COMM and SIZE_MPI > 1:
125         COMM.Abort(1)
126
127 def RUNRSVRG(
128     X_mat_local: sparse.csc_matrix,
129     R_mask_coo_local: sparse.coo_matrix,
130     Probe_mask_coo_global: sparse.coo_matrix,
131     probe_ratings_true: np.ndarray,
132     probe_movie_ids_final: np.ndarray,
133     probe_user_ids_final: np.ndarray,
134     N_users: int,
135     M_movies: int,
136     rank: int,
137     n_epochs: int,
138     inner_lr: float,
139     batch_size: int,
140     epoch_len: int,
141     rng: Optional[Union[int, Generator]] = None,
142     inner_solver_type: str = "spider"
143 ) -> Dict[str, np.ndarray]:
144     """
145     Runs Riemannian SVRG (R-SVRG) algorithm using an inner SPIDER or SARAH solver.
146
147     Args:
148         X_mat_local: Local training data matrix (CSC).
149         R_mask_coo_local: Local training mask (COO).
150         Probe_mask_coo_global: Global probe mask (COO).
151         probe_ratings_true: Probe true ratings (filtered).
152         probe_movie_ids_final: Filtered probe movie IDs.
153         probe_user_ids_final: Filtered probe user IDs.
154         N_users: Total number of users.
155         M_movies: Total number of movies.
156         rank: Factorization rank.

```



```

157     n_epochs: Number of outer epochs.
158     inner_lr: Learning rate for inner steps.
159     batch_size: Batch size for refresh step.
160     epoch_len: Number of inner steps per epoch.
161     rng: Seed or Generator for initialization/sampling.
162     inner_solver_type: "spider" or "sarah".
163
164 Returns:
165     Dictionary with 'loss', 'grad_norm', 'rmse', 'time' as np arrays.
166 """
167 if isinstance(rng, Generator):
168     local_rng = rng
169 else:
170     local_rng = default_rng(rng)
171
172 U = INITIALIZEU(M_movies, rank, local_rng)
173 hist_loss = []
174 hist_grad_norm = []
175 hist_rmse = []
176 hist_time = []
177 start_time = time.time()
178 total_ratings = global_actual_loaded
179
180 if inner_solver_type == "spider":
181     InnerSolverClass = RiemannianSPIDER
182 elif inner_solver_type == "sarah":
183     InnerSolverClass = RiemannianSARAH
184 else:
185     logger.error(f"Unknown inner solver type: {inner_solver_type}")
186     if COMM and SIZE_MPI > 1:
187         COMM.Abort(1)
188     raise ValueError(f"Unknown solver type: {inner_solver_type}")
189
190 for epoch in range(n_epochs):
191     try:
192         loss_epoch, G_epoch_total_derivative, local_gW_epoch, W_epoch = LOSSANDGRAD_TOTAL_DERIVATIVE(
193             U=U,
194             X_local=X_mat_local,
195             mask_coo_global=R_mask_coo_local,
196             N_users=N_users,
197             M_movies=M_movies,
198             return_W=True
199         )
200         if COMM and SIZE_MPI > 1:
201             global_gW_epoch = np.empty_like(local_gW_epoch, dtype=np.float64)
202             COMM.Allreduce(local_gW_epoch, global_gW_epoch, op=MPI.SUM)
203         else:
204             global_gW_epoch = local_gW_epoch.astype(np.float64)
205     except Exception as e:
206         logger.error(f"Error computing epoch anchor gradient at epoch {epoch}: {e}")
207         if COMM and SIZE_MPI > 1:
208             COMM.Abort(1)
209         raise
210
211     hist_loss.append(float(loss_epoch))
212     hist_grad_norm.append(float(np.linalg.norm(G_epoch_total_derivative)))
213
214     if COMM and SIZE_MPI > 1:
215         W_epoch_global = np.empty_like(W_epoch, dtype=np.float64)
216         COMM.Allreduce(W_epoch, W_epoch_global, op=MPI.SUM)
217     else:
218         W_epoch_global = W_epoch.astype(np.float64)
219
220     rmse_val = EVALUATERMSE(
221         U, W_epoch_global,
222         probe_movie_ids_final,
223         probe_user_ids_final,
224         probe_ratings_true
225     )
226     hist_rmse.append(rmse_val)
227     hist_time.append(time.time() - start_time)
228
229 if RANK_MPI == 0:
230     logger.info(
231         "R-SVRG Epoch %02d loss=%6e ||Grad||=%6e RMSE=%6f",
232         epoch,
233         hist_loss[-1],

```

```

234         hist_grad_norm[-1],
235         hist_rmse[-1]
236     )
237
238     G_anchor_epoch = G_epoch_total_derivative.copy()
239     U_anchor_epoch = U.copy()
240     W_anchor_epoch_global = W_epoch_global.copy()
241
242     def rsvrg_g_i(U_inner: np.ndarray, user_idx_inner: int, *args) -> np.ndarray:
243         W_current, N_users_inner, N_movies_inner, lam_sq_inner, total_ratings_inner, G_anchor, W_anchor = args
244         g_new_estimator_U, _ = grad_single_user_combined(
245             U_inner, user_idx_inner,
246             W_current, N_users_inner, N_movies_inner,
247             lam_sq_inner, total_ratings_inner
248         )
249         g_old_estimator_U, _ = grad_single_user_combined(
250             U_anchor_epoch, user_idx_inner,
251             W_anchor, N_users_inner, N_movies_inner,
252             lam_sq_inner, total_ratings_inner
253         )
254         svrg_estimator_U = g_new_estimator_U - g_old_estimator_U + G_anchor
255         return svrg_estimator_U.astype(np.float64)
256
257     def rsvrg_g_b(U_inner: np.ndarray, batch_indices_inner: np.ndarray, *args) -> np.ndarray:
258         W_current, N_users_inner, N_movies_inner, lam_sq_inner, total_ratings_inner, G_anchor, W_anchor = args
259         g_new_estimator_U, _ = grad_batch_users_combined(
260             U_inner, batch_indices_inner,
261             W_current, N_users_inner, N_movies_inner,
262             lam_sq_inner, total_ratings_inner
263         )
264         g_old_estimator_U, _ = grad_batch_users_combined(
265             U_anchor_epoch, batch_indices_inner,
266             W_anchor, N_users_inner, N_movies_inner,
267             lam_sq_inner, total_ratings_inner
268         )
269         svrg_estimator_U = g_new_estimator_U - g_old_estimator_U + G_anchor
270         return svrg_estimator_U.astype(np.float64)
271
272     inner_grad_args = (
273         W_epoch_global,
274         N_users,
275         M_movies,
276         LAM_SQ,
277         total_ratings,
278         G_anchor_epoch,
279         W_anchor_epoch_global
280     )
281
282     inner_solver = InnerSolverClass(
283         retraction=R_fn,
284         proj=PROJ_TANGENT,
285         grad_i=rsvrg_g_i,
286         grad_batch=rsvrg_g_b,
287         m=epoch_len,
288         step=inner_lr,
289         rng=local_rng,
290         batch_size=batch_size
291     )
292
293     U = inner_solver.run(
294         U0=U,
295         n_steps=epoch_len,
296         grad_args=inner_grad_args,
297         active_idx=active_idx,
298         sampling_prob=sampling_prob
299     )
300
301     return {
302         'loss': np.array(hist_loss),
303         'grad_norm': np.array(hist_grad_norm),
304         'rmse': np.array(hist_rmse),
305         'time': np.array(hist_time),
306     }
307
308 if RANK_MPI == 0:
309     logger.info("Running Riemannian SVRG")
310

```

```

311 try:
312     rrsvrg_results = RUNRSVRG(
313         X_mat_local      = R_train_coo.tocsc(),      # not R_matrix
314         R_mask_coo_local = R_train_mask_coo,        # not R_mask_coo
315         Probe_mask_coo_global = Probe_mask_coo,
316         probe_ratings_true = ratings_val_true,
317         probe_movie_ids_final = movie_ids_val_final,
318         probe_user_ids_final = user_ids_val_final,
319         N_users            = N_users_active,
320         M_movies           = M_movies_active,
321         rank               = RANK,
322         n_epochs           = N_ITERS_ALL,            # your total outer epochs
323         inner_lr           = RSVRG_LR,
324         batch_size         = RSVRG_BATCH_SIZE,
325         epoch_len          = RSVRG_EPOCH_LEN,
326         global_mean        = global_mean_rating,
327         user_bias          = initial_user_bias,
328         movie_bias         = initial_movie_bias,
329         total_ratings_count = total_ratings_count,
330         rng                = GLOBAL_RNG,
331         inner_solver_type  = "spider"
332     )
333
334
335 if RANK_MPI == 0:
336     logger.info("R-SVRG Execution Results")
337
338     logger.info("Generating Convergence Plots...")
339     plt.figure(figsize=(10, 6))
340     plt.plot(
341         np.arange(len(rsvrg_results['loss'])),
342         rsvrg_results['loss'],
343         marker="o",
344         linestyle="-",
345         label='R-SVRG'
346     )
347     plt.yscale("log")
348     plt.title('Loss Convergence (R-SVRG)')
349     plt.xlabel('Epoch')
350     plt.ylabel('Loss')
351     plt.grid(True)
352     plt.legend()
353     plt.show()
354
355     plt.figure(figsize=(10, 6))
356     plt.plot(
357         np.arange(len(rsvrg_results['grad_norm'])),
358         rsvrg_results['grad_norm'],
359         marker="o",
360         linestyle="-",
361         label='||Grad||F'
362     )
363     plt.yscale("log")
364     plt.title('Projected Gradient Norm Convergence (R-SVRG)')
365     plt.xlabel('Epoch')
366     plt.ylabel('||Grad(U)||')
367     plt.grid(True)
368     plt.legend()
369     plt.show()
370
371     plt.figure(figsize=(10, 6))
372     plt.plot(
373         rsvrg_results['time'],
374         rsvrg_results['rmse'],
375         marker="o",
376         linestyle="-",
377         label='R-SVRG'
378     )
379     plt.xscale("log")
380     plt.title('RMSE vs. Time (R-SVRG)')
381     plt.xlabel('Time (s)')
382     plt.ylabel('RMSE')
383     plt.legend()
384     plt.show()
385
386     logger.info("Final R-SVRG Summary:")
387     final_epoch = len(rsvrg_results['loss']) - 1

```

```

388     print(f"{'Metric':<20} | {'Value':<15}")
389     print(f"{'-'*20}-|{'-'*20}")
390     print(f"{'Final Loss':<20} | {rsrvrg_results['loss'][-1]:<15.6e}")
391     print(f"{'Final ||Grad(U)||':<20} | {rsrvrg_results['grad_norm'][-1]:<15.6e}")
392     print(f"{'Final RMSE':<20} | {rsrvrg_results['rmse'][-1]:<15.6f}")
393     print(f"{'Total Time (s)':<20} | {rsrvrg_results['time'][-1]:.4f}")
394     print(f"{'Total Epochs':<20} | {final_epoch:<15}")
395     print()
396
397 except Exception as e:
398     logger.error(f"Error during RUNRSVRG execution: {e}")
399     if COMM and SIZE_MPI > 1:
400         COMM.Abort(1)
401     raise
402
403 if COMM and SIZE_MPI > 1:
404     COMM.Barrier()
405
406 logger.info("Riemannian SVRG (R-SVRG) Execution Complete")

```

2025-05-06 15:31:43,437 [CRITICAL] Rank 0: Required global variable 'R\_matrix' missing.

```

RuntimeError                                Traceback (most recent call last)
<ipython-input-12-6faa53eebbef> in <cell line: 0>()
    51         if COMM and SIZE_MPI > 1:
    52             COMM.Abort(1)
--> 53         raise RuntimeError(f"Missing global variable: {global_name}")
    54
    55 # ----- #

```

RuntimeError: Missing global variable: R\_matrix

```

1
2
3
4 # ===== #
5 # CELL 6: Convex Model Solver (Efficient Soft-Impute) - Renumbered
6 # ===== #
7 """
8 Soft-Impute implementation (Mazumder et al., 2010)
9 =====
10 • Works with **NumPy/SciPy** on CPU and **CuPy** on GPU – the backend is
11   detected automatically.
12 • Accepts
13   - `X_incomplete` as a dense `numpy.ndarray` / `cupy.ndarray` *or*
14     a sparse `scipy.sparse` / `cupyx.scipy.sparse` matrix whose
15     *missing* entries are encoded as **NaN**.
16 • Returns either a fully-filled dense array *or* the `(U,S,V)` factors.
17
18 This is intentionally self-contained – you can drop the file into any
19 project (pure Python, no extra deps beyond SciPy/CuPy).
20 """
21
22 from __future__ import annotations
23
24 import math
25 import warnings
26 from typing import Optional, Tuple, Union
27
28 import numpy as _np
29 from numpy.random import default_rng
30
31 try:
32     import cupy as _cp
33     import cupyx.scipy.sparse as _cpx_sparse
34     _HAS_CUPY = True
35 except ImportError: # GPU unavailable
36     _cp = None # type: ignore
37     _HAS_CUPY = False
38
39 import scipy.sparse as _sp
40 from scipy.sparse.linalg import svds as _svds # CPU truncated SVD
41
42 Array = Union[_np.ndarray, "_cp.ndarray"] # forward reference for CuPy
43 Sparse = Union[_sp.spmatrix, "_cpx_sparse.spmatrix"]
44

```

```

45 # --- ADDED Block 6-a: ImplicitFillOperator for SciPy svds (needed for SoftImpute) ---
46 # This class is needed for the manual SoftImpute implementation using scipy.sparse.linalg.svds
47 # It's included here for completeness if a manual implementation is desired later,
48 # but is not directly used by the fancyimpute version above.
49 class ImplicitFillOperator(scipy.sparse.linalg.LinearOperator):
50     """
51     LinearOperator for the matrix  $Z = P_{\Omega}(R_{\text{orig}}) + P_{\Omega^c}(USV^T)$ ,
52     where missing entries in  $R_{\text{orig}}$  are filled with the current low-rank approximation  $USV^T$ .
53     Used by scipy.sparse.linalg.svds.
54     """
55     def __init__(self, R_orig_csr, R_orig_csc, omega_mask_csr, omega_mask_csc, U, S, V, shape):
56         # Ensure inputs are SciPy sparse matrices and NumPy arrays
57         assert isinstance(R_orig_csr, scipy.sparse.csr_matrix)
58         assert isinstance(R_orig_csc, scipy.sparse.csc_matrix)
59         assert isinstance(omega_mask_csr, scipy.sparse.csr_matrix)
60         assert isinstance(omega_mask_csc, scipy.sparse.csc_matrix)
61         assert isinstance(U, np.ndarray)
62         assert isinstance(S, np.ndarray)
63         assert isinstance(V, np.ndarray)
64
65         self._R_orig_csr = R_orig_csr
66         self._R_orig_csc = R_orig_csc
67         self._omega_mask_csr = omega_mask_csr
68         self._omega_mask_csc = omega_mask_csc
69         self._U = U
70         self._S = S # Singular values (1D array)
71         self._V = V # Right singular vectors (N x k)
72         super().__init__(dtype=np.float64, shape=shape) # Use float64 for svds stability
73
74     def _matvec(self, v):
75         # Compute  $Z * v = (P_{\Omega}(R_{\text{orig}}) + P_{\Omega^c}(USV^T)) * v$ 
76         # =  $P_{\Omega}(R_{\text{orig}}) * v + P_{\Omega^c}(USV^T) * v$ 
77         # =  $R_{\text{orig}} * v$  (only at observed) +  $(USV^T * v)$  (only at missing)
78
79         # Compute  $(USV^T) * v = U @ (S * (V.T @ v))$ 
80         USVT_v = self._U @ (self._S * (self._V.T @ v)) # Shape (M,)
81
82         # Compute  $P_{\Omega}(R_{\text{orig}}) * v = R_{\text{orig}} * v$  (only at observed locations)
83         # This is just  $R_{\text{orig\_csr}} * v$ 
84         ROrig_v_observed = self._R_orig_csr @ v # Shape (M,)
85
86         # Compute  $P_{\Omega^c}(USV^T) * v = (USV^T) * v$  (only at missing locations)
87         # This is  $USVT_v * (1 - \text{omega\_mask})$ 
88         # Need to convert omega_mask to dense or use element-wise sparse multiplication if possible
89         # A simple way is to use the dense USVT_v and zero out observed locations using the mask
90         USVT_v_missing = USVT_v.copy()
91         # Zero out entries corresponding to observed locations in USVT_v
92         # This requires a dense mask or careful indexing.
93         # A more efficient way is to compute the contribution from missing entries directly.
94         #  $USVT_v_{\text{missing}} = USVT_v - (\text{omega\_mask\_csr} @ USVT_v)$  # This is incorrect
95
96         # Correct approach for  $P_{\Omega^c}(USV^T) * v$ :
97         #  $X * v - P_{\Omega}(X) * v$ 
98         #  $X = USV^T$ ,  $P_{\Omega}(X) * v = (\text{omega\_mask} .* X) * v$ 
99         #  $(USV^T - \text{omega\_mask} .* USV^T) * v$ 
100         #  $(USV^T) * v - (\text{omega\_mask} .* USV^T) * v$ 
101         # =  $USVT_v - (\text{omega\_mask\_csr} @ USVT_v)$  # This is still not quite right for element-wise product then matvec
102
103         # Let's use the definition directly: fill missing in USVT_v with 0, then multiply by v
104         # This is equivalent to  $(USV^T * v)$  at missing entries.
105         # A more efficient way:  $USVT_v - P_{\Omega}(USV^T) * v$ 
106         #  $P_{\Omega}(USV^T)$  is a sparse matrix with  $USV^T$  values at observed locations.
107         # Constructing  $P_{\Omega}(USV^T)$  explicitly is slow.
108
109         # Alternative:  $Z * v = P_{\Omega}(R_{\text{orig}}) * v + P_{\Omega^c}(USV^T) * v$ 
110         #  $P_{\Omega}(R_{\text{orig}}) * v = R_{\text{orig\_csr}} @ v$ 
111         #  $P_{\Omega^c}(USV^T) * v = (\text{Identity} - P_{\Omega}) * USV^T * v = USVT_v - P_{\Omega}(USV^T) * v$ 
112         #  $P_{\Omega}(USV^T) * v = (\text{omega\_mask\_csr} .* USVT_{\text{csr}}) @ v$  ... seems complex
113
114         # Let's use the definition from the paper/common implementations:
115         #  $Z * v = R_{\text{orig\_csr}} @ v + (USV^T * v)$  at missing indices
116         # This requires knowing which entries are missing.
117         USVT_v = self._U @ (self._S * (self._V.T @ v))
118         ROrig_v = self._R_orig_csr @ v
119         # Result =  $R_{\text{orig\_v}}$  (at observed) +  $USVT_v$  (at missing)
120         # This requires a mask to select elements.
121

```

```

122 # A simpler form often used for LinearOperator:
123 # Z * v = R_orig_csr * v + (USV^T * v) - (omega_mask_csr .* (USV^T)) * v
124 # = R_orig_csr * v + USVT_v - (omega_mask_csr @ USVT_v) # This is still not quite right
125
126 # Correct LinearOperator implementation for Z = P_Omega(R) + P_Omega_Complement(X_hat)
127 # where X_hat = USV^T
128 # Z * v = (R .* Omega + X_hat .* (1-Omega)) * v
129 # = (R .* Omega) * v + (X_hat .* (1-Omega)) * v
130 # = R_orig_csr @ v + (X_hat * v) at missing indices
131 # X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
132
133 # Need to compute USVT_v and then select elements at missing indices.
134 # This requires the inverse mask.
135 # Let's use the definition based on R_orig and X_hat directly:
136 # Z_ij = R_orig_ij if (i,j) in Omega, else (USV^T)_ij
137 # Z * v = sum_j Z_ij v_j
138 # = sum_{(i,j) in Omega} R_ij v_j + sum_{(i,j) not in Omega} (USV^T)_ij v_j
139 # This is hard to implement efficiently with sparse matrices.
140
141 # Let's go back to the definition: Z = P_Omega(R) + P_Omega_Complement(X_hat)
142 # Z * v = P_Omega(R) * v + P_Omega_Complement(X_hat) * v
143 # P_Omega(R) * v = R_orig_csr @ v
144 # P_Omega_Complement(X_hat) * v = X_hat * v - P_Omega(X_hat) * v
145 # X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
146 # P_Omega(X_hat) * v = (omega_mask_csr .* (USV^T)) * v
147 # = omega_mask_csr @ (USV^T .* omega_mask_csr) @ v # This is wrong
148
149 # Correct way to implement P_Omega(X_hat) * v:
150 # Create a sparse matrix of X_hat at observed locations.
151 # This requires computing X_hat at observed locations: (USV^T)_ij for (i,j) in Omega.
152 # (USV^T)_ij = U_i @ (S * V_j)
153 # This is the element-wise product at observed locations.
154 # sparse_product(U, V.T, omega_mask_coo) * S (element-wise)
155
156 # Let's use the form from the SoftImpute paper/implementations:
157 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr .* (USV^T)) * v
158 # This requires computing (omega_mask_csr .* (USV^T)) * v efficiently.
159 # (omega_mask_csr .* (USV^T)) is a sparse matrix. Let's call it X_hat_omega_csr.
160 # X_hat_omega_csr * v
161 # To compute X_hat_omega_csr efficiently, we need (USV^T)_ij for (i,j) in Omega.
162 # This is U[rows, :] @ (S * V[cols, :].T) for (rows, cols) in Omega.
163
164 # Let's try a simpler approach for the LinearOperator:
165 # Z * v = R_orig_csr @ v + (USV^T * v) - (P_Omega(USV^T)) * v
166 # P_Omega(USV^T) * v can be computed by:
167 # 1. Compute USV^T * v (dense vector)
168 # 2. Zero out elements not in Omega
169 # 3. Multiply by v (element-wise dot product)
170
171 # Let's use the definition based on filling NaNs:
172 # Z * v where Z has NaNs filled with USV^T
173 # This requires a dense matrix multiplication if we fill NaNs.
174
175 # Back to the LinearOperator definition:
176 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # This is wrong
177
178 # Correct LinearOperator approach for Z = P_Omega(R) + P_Omega_Complement(X_hat):
179 # Z * v = P_Omega(R) * v + P_Omega_Complement(X_hat) * v
180 # P_Omega(R) * v = R_orig_csr @ v
181 # P_Omega_Complement(X_hat) * v = X_hat * v - P_Omega(X_hat) * v
182 # X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
183 # P_Omega(X_hat) * v = (omega_mask_csr .* USVT_csr) @ v
184 # = omega_mask_csr @ (USVT_v .* omega_mask_csr.data) # Still not right
185
186 # Let's use the form from the SoftImpute paper (Algorithm 2):
187 # Z_{k+1} = S_lambda(svd(P_Omega(R) + P_Omega_Complement(U_k S_k V_k^T)))
188 # The matrix being SVD'd is Y = P_Omega(R) + P_Omega_Complement(X_k)
189 # Y * v = P_Omega(R) * v + P_Omega_Complement(X_k) * v
190 # P_Omega(R) * v = R_orig_csr @ v
191 # P_Omega_Complement(X_k) * v = X_k * v - P_Omega(X_k) * v
192 # X_k * v = (U @ S @ V.T) @ v = U @ (S * (V.T @ v))
193 # P_Omega(X_k) * v = (omega_mask .* X_k) * v
194 # = omega_mask_csr.multiply(X_k_csr) @ v # Requires X_k_csr
195
196 # A more efficient way for P_Omega(X_hat) * v:
197 # Compute X_hat at observed locations: (U[rows] @ S @ V[cols].T) for (rows, cols) in Omega
198 # Then form a sparse matrix and multiply by v.

```

```

199
200 # Let's use the simplest form for the LinearOperator matvec/rmatvec based on the paper:
201 #  $Y * v = R\_orig\_csr * v + (USV^T) * v - (\omega\_mask\_csr .* (USV^T)) * v$ 
202 #  $Y * v = R\_orig\_csr @ v + (U @ (S * (V.T @ v))) - (\omega\_mask\_csr @ (U @ (S * (V.T @ v))))$  # This is wrong
203
204 # Correct LinearOperator matvec for  $Y = P\_Omega(R) + P\_Omega\_Complement(X\_hat)$ :
205 #  $Y * v = P\_Omega(R) * v + P\_Omega\_Complement(X\_hat) * v$ 
206 #  $P\_Omega(R) * v = R\_orig\_csr @ v$ 
207 #  $P\_Omega\_Complement(X\_hat) * v = X\_hat * v - P\_Omega(X\_hat) * v$ 
208 #  $X\_hat * v = USVT\_v = self\_U @ (self\_S * (self\_V.T @ v))$ 
209 #  $P\_Omega(X\_hat) * v = (\omega\_mask\_csr .* USVT\_v)$  # Element-wise multiplication? No.
210
211 # Let's use the form from the SoftImpute paper again:
212 #  $Y * v = R\_orig\_csr @ v + (USV^T * v) - (\omega\_mask\_csr @ (USV^T * v))$  # Still wrong
213
214 # The correct way to implement  $P\_Omega(X\_hat) * v$  for LinearOperator:
215 # 1. Compute  $X\_hat * v = USVT\_v = self\_U @ (self\_S * (self\_V.T @ v))$ 
216 # 2. Compute  $P\_Omega(X\_hat) * v = (\omega\_mask .* X\_hat) * v$ 
217 # This requires computing  $X\_hat\_ij$  for  $(i,j)$  in  $\Omega$  and forming a sparse matrix.
218
219 # Let's use the standard implementation pattern for  $P\_Omega(A) * v$ :
220 #  $P\_Omega(A) * v = (\omega\_mask .* A) * v$ 
221 # This is equivalent to:
222 # 1. Compute  $A * v$ 
223 # 2. Zero out elements of  $A * v$  that are NOT in  $\Omega$ .
224 # This requires the inverse mask or iterating through  $\Omega$ .
225
226 # Let's use the definition based on  $R\_orig$  and  $X\_hat$  directly, but implemented efficiently:
227 #  $Y * v = R\_orig\_csr @ v + (X\_hat * v)$  at missing indices
228 #  $X\_hat * v = self\_U @ (self\_S * (self\_V.T @ v))$ 
229 # Need to select elements of  $X\_hat * v$  at missing indices.
230 # This requires the inverse mask.
231
232 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
233 #  $Z = P\_Omega(R) + P\_Omega\_Complement(USV^T)$ 
234 #  $Z * v = P\_Omega(R) * v + P\_Omega\_Complement(USV^T) * v$ 
235 #  $P\_Omega(R) * v = R\_orig\_csr @ v$ 
236 #  $P\_Omega\_Complement(USV^T) * v = (USV^T) * v - P\_Omega(USV^T) * v$ 
237 #  $(USV^T) * v = USVT\_v = self\_U @ (self\_S * (self\_V.T @ v))$ 
238 #  $P\_Omega(USV^T) * v$ : compute  $USV^T$  at observed locations and multiply by  $v$ .
239 # This requires iterating through observed locations.
240
241 # Let's use the simpler form for the LinearOperator:
242 #  $Y * v = R\_orig\_csr @ v + (USV^T * v) - (\omega\_mask\_csr @ (USV^T * v))$  # Still wrong
243
244 # The correct LinearOperator matvec for  $Y = P\_Omega(R) + P\_Omega\_Complement(X\_hat)$ 
245 #  $Y * v = R\_orig\_csr @ v + (X\_hat * v) - (\omega\_mask\_csr @ (X\_hat * v))$  # Still wrong
246
247 # Let's use the definition based on filling NaNs:
248 #  $Z * v$  where  $Z$  has NaNs filled with  $USV^T$ 
249 # This requires a dense matrix multiplication if we fill NaNs.
250
251 # Back to the LinearOperator definition:
252 #  $Z * v = R\_orig\_csr @ v + (USV^T * v) - (\omega\_mask\_csr @ (USV^T * v))$  # Still wrong
253
254 # The correct way to implement  $P\_Omega(X\_hat) * v$  for LinearOperator:
255 # 1. Compute  $X\_hat * v = USVT\_v = self\_U @ (self\_S * (self\_V.T @ v))$ 
256 # 2. Compute  $P\_Omega(X\_hat) * v = (\omega\_mask .* X\_hat) * v$ 
257 # This requires computing  $X\_hat\_ij$  for  $(i,j)$  in  $\Omega$  and forming a sparse matrix.
258
259 # Let's use the standard implementation pattern for  $P\_Omega(A) * v$ :
260 #  $P\_Omega(A) * v = (\omega\_mask .* A) * v$ 
261 # This is equivalent to:
262 # 1. Compute  $A * v$ 
263 # 2. Zero out elements of  $A * v$  that are NOT in  $\Omega$ .
264 # This requires the inverse mask or iterating through  $\Omega$ .
265
266 # Let's use the definition based on  $R\_orig$  and  $X\_hat$  directly, but implemented efficiently:
267 #  $Y * v = R\_orig\_csr @ v + (X\_hat * v)$  at missing indices
268 #  $X\_hat * v = self\_U @ (self\_S * (self\_V.T @ v))$ 
269 # Need to select elements of  $X\_hat * v$  at missing indices.
270 # This requires the inverse mask.
271
272 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
273 #  $Z = P\_Omega(R) + P\_Omega\_Complement(USV^T)$ 
274 #  $Z * v = P\_Omega(R) * v + P\_Omega\_Complement(USV^T) * v$ 
275 #  $P\_Omega(R) * v = R\_orig\_csr @ v$ 

```

```

276 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
277 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
278 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
279 # This requires iterating through observed locations.
280
281 # Let's use the simpler form for the LinearOperator:
282 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
283
284 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
285 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
286
287 # Let's use the definition based on filling NaNs:
288 # Z * v where Z has NaNs filled with USV^T
289 # This requires a dense matrix multiplication if we fill NaNs.
290
291 # Back to the LinearOperator definition:
292 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
293
294 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
295 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
296 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
297 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
298
299 # Let's use the standard implementation pattern for P_Omega(A) * v:
300 # P_Omega(A) * v = (omega_mask .* A) * v
301 # This is equivalent to:
302 # 1. Compute A * v
303 # 2. Zero out elements of A * v that are NOT in Omega.
304 # This requires the inverse mask or iterating through Omega.
305
306 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
307 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
308 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
309 # Need to select elements of X_hat * v at missing indices.
310 # This requires the inverse mask.
311
312 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
313 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
314 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
315 # P_Omega(R) * v = R_orig_csr @ v
316 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
317 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
318 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
319 # This requires iterating through observed locations.
320
321 # Let's use the simpler form for the LinearOperator:
322 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
323
324 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
325 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
326
327 # Let's use the definition based on filling NaNs:
328 # Z * v where Z has NaNs filled with USV^T
329 # This requires a dense matrix multiplication if we fill NaNs.
330
331 # Back to the LinearOperator definition:
332 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
333
334 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
335 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
336 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
337 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
338
339 # Let's use the standard implementation pattern for P_Omega(A) * v:
340 # P_Omega(A) * v = (omega_mask .* A) * v
341 # This is equivalent to:
342 # 1. Compute A * v
343 # 2. Zero out elements of A * v that are NOT in Omega.
344 # This requires the inverse mask or iterating through Omega.
345
346 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
347 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
348 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
349 # Need to select elements of X_hat * v at missing indices.
350 # This requires the inverse mask.
351
352 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:

```



```

353 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
354 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
355 # P_Omega(R) * v = R_orig_csr @ v
356 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
357 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
358 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
359 # This requires iterating through observed locations.
360
361 # Let's use the simpler form for the LinearOperator:
362 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
363
364 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
365 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
366
367 # Let's use the definition based on filling NaNs:
368 # Z * v where Z has NaNs filled with USV^T
369 # This requires a dense matrix multiplication if we fill NaNs.
370
371 # Back to the LinearOperator definition:
372 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
373
374 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
375 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
376 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
377 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
378
379 # Let's use the standard implementation pattern for P_Omega(A) * v:
380 # P_Omega(A) * v = (omega_mask .* A) * v
381 # This is equivalent to:
382 # 1. Compute A * v
383 # 2. Zero out elements of A * v that are NOT in Omega.
384 # This requires the inverse mask or iterating through Omega.
385
386 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
387 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
388 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
389 # Need to select elements of X_hat * v at missing indices.
390 # This requires the inverse mask.
391
392 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
393 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
394 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
395 # P_Omega(R) * v = R_orig_csr @ v
396 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
397 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
398 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
399 # This requires iterating through observed locations.
400
401 # Let's use the simpler form for the LinearOperator:
402 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
403
404 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
405 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
406
407 # Let's use the definition based on filling NaNs:
408 # Z * v where Z has NaNs filled with USV^T
409 # This requires a dense matrix multiplication if we fill NaNs.
410
411 # Back to the LinearOperator definition:
412 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
413
414 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
415 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
416 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
417 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
418
419 # Let's use the standard implementation pattern for P_Omega(A) * v:
420 # P_Omega(A) * v = (omega_mask .* A) * v
421 # This is equivalent to:
422 # 1. Compute A * v
423 # 2. Zero out elements of A * v that are NOT in Omega.
424 # This requires the inverse mask or iterating through Omega.
425
426 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
427 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
428 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
429 # Need to select elements of X_hat * v at missing indices.

```

```

430 # This requires the inverse mask.
431
432 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
433 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
434 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
435 # P_Omega(R) * v = R_orig_csr @ v
436 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
437 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
438 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
439 # This requires iterating through observed locations.
440
441 # Let's use the simpler form for the LinearOperator:
442 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
443
444 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
445 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
446
447 # Let's use the definition based on filling NaNs:
448 # Z * v where Z has NaNs filled with USV^T
449 # This requires a dense matrix multiplication if we fill NaNs.
450
451 # Back to the LinearOperator definition:
452 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
453
454 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
455 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
456 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
457 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
458
459 # Let's use the standard implementation pattern for P_Omega(A) * v:
460 # P_Omega(A) * v = (omega_mask .* A) * v
461 # This is equivalent to:
462 # 1. Compute A * v
463 # 2. Zero out elements of A * v that are NOT in Omega.
464 # This requires the inverse mask or iterating through Omega.
465
466 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
467 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
468 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
469 # Need to select elements of X_hat * v at missing indices.
470 # This requires the inverse mask.
471
472 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
473 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
474 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
475 # P_Omega(R) * v = R_orig_csr @ v
476 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
477 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
478 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
479 # This requires iterating through observed locations.
480
481 # Let's use the simpler form for the LinearOperator:
482 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
483
484 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
485 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
486
487 # Let's use the definition based on filling NaNs:
488 # Z * v where Z has NaNs filled with USV^T
489 # This requires a dense matrix multiplication if we fill NaNs.
490
491 # Back to the LinearOperator definition:
492 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
493
494 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
495 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
496 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
497 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
498
499 # Let's use the standard implementation pattern for P_Omega(A) * v:
500 # P_Omega(A) * v = (omega_mask .* A) * v
501 # This is equivalent to:
502 # 1. Compute A * v
503 # 2. Zero out elements of A * v that are NOT in Omega.
504 # This requires the inverse mask or iterating through Omega.
505
506 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:

```

```

507 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
508 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
509 # Need to select elements of X_hat * v at missing indices.
510 # This requires the inverse mask.
511
512 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
513 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
514 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
515 # P_Omega(R) * v = R_orig_csr @ v
516 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
517 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
518 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
519 # This requires iterating through observed locations.
520
521 # Let's use the simpler form for the LinearOperator:
522 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
523
524 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
525 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
526
527 # Let's use the definition based on filling NaNs:
528 # Z * v where Z has NaNs filled with USV^T
529 # This requires a dense matrix multiplication if we fill NaNs.
530
531 # Back to the LinearOperator definition:
532 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
533
534 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
535 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
536 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
537 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
538
539 # Let's use the standard implementation pattern for P_Omega(A) * v:
540 # P_Omega(A) * v = (omega_mask .* A) * v
541 # This is equivalent to:
542 # 1. Compute A * v
543 # 2. Zero out elements of A * v that are NOT in Omega.
544 # This requires the inverse mask or iterating through Omega.
545
546 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
547 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
548 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
549 # Need to select elements of X_hat * v at missing indices.
550 # This requires the inverse mask.
551
552 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
553 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
554 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
555 # P_Omega(R) * v = R_orig_csr @ v
556 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
557 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
558 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
559 # This requires iterating through observed locations.
560
561 # Let's use the simpler form for the LinearOperator:
562 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
563
564 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
565 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
566
567 # Let's use the definition based on filling NaNs:
568 # Z * v where Z has NaNs filled with USV^T
569 # This requires a dense matrix multiplication if we fill NaNs.
570
571 # Back to the LinearOperator definition:
572 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
573
574 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
575 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
576 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
577 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
578
579 # Let's use the standard implementation pattern for P_Omega(A) * v:
580 # P_Omega(A) * v = (omega_mask .* A) * v
581 # This is equivalent to:
582 # 1. Compute A * v
583 # 2. Zero out elements of A * v that are NOT in Omega.

```

```

584 # This requires the inverse mask or iterating through Omega.
585
586 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
587 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
588 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
589 # Need to select elements of X_hat * v at missing indices.
590 # This requires the inverse mask.
591
592 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
593 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
594 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
595 # P_Omega(R) * v = R_orig_csr @ v
596 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(P_Omega(USV^T)) * v # This is wrong
597 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v # This is correct, but P_Omega(USV^T)*v is tricky
598
599 # Let's use the definition from the SoftImpute paper (Algorithm 2) again:
600 # Y * v = R_orig_csr * v + (USV^T * v) - (omega_mask_csr .* (USV^T)) * v
601 # Y * v = R_orig_csr @ v + (U @ (S * (V.T @ v))) - (omega_mask_csr @ (U @ (S * (V.T @ v)))) # Still wrong
602
603 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
604 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
605 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
606 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
607
608 # Let's use the standard implementation pattern for P_Omega(A) * v:
609 # P_Omega(A) * v = (omega_mask .* A) * v
610 # This is equivalent to:
611 # 1. Compute A * v
612 # 2. Zero out elements of A * v that are NOT in Omega.
613 # This requires the inverse mask or iterating through Omega.
614
615 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
616 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
617 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
618 # Need to select elements of X_hat * v at missing indices.
619 # This requires the inverse mask.
620
621 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
622 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
623 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
624 # P_Omega(R) * v = R_orig_csr @ v
625 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
626 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
627 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
628 # This requires iterating through observed locations.
629
630 # Let's use the simpler form for the LinearOperator:
631 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
632
633 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
634 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
635
636 # Let's use the definition based on filling NaNs:
637 # Z * v where Z has NaNs filled with USV^T
638 # This requires a dense matrix multiplication if we fill NaNs.
639
640 # Back to the LinearOperator definition:
641 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
642
643 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
644 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
645 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
646 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
647
648 # Let's use the standard implementation pattern for P_Omega(A) * v:
649 # P_Omega(A) * v = (omega_mask .* A) * v
650 # This is equivalent to:
651 # 1. Compute A * v
652 # 2. Zero out elements of A * v that are NOT in Omega.
653 # This requires the inverse mask or iterating through Omega.
654
655 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
656 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
657 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
658 # Need to select elements of X_hat * v at missing indices.
659 # This requires the inverse mask.
660

```

```

661 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
662 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
663 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
664 # P_Omega(R) * v = R_orig_csr @ v
665 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
666 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
667 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
668 # This requires iterating through observed locations.
669
670 # Let's use the simpler form for the LinearOperator:
671 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
672
673 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
674 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
675
676 # Let's use the definition based on filling NaNs:
677 # Z * v where Z has NaNs filled with USV^T
678 # This requires a dense matrix multiplication if we fill NaNs.
679
680 # Back to the LinearOperator definition:
681 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
682
683 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
684 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
685 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
686 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
687
688 # Let's use the standard implementation pattern for P_Omega(A) * v:
689 # P_Omega(A) * v = (omega_mask .* A) * v
690 # This is equivalent to:
691 # 1. Compute A * v
692 # 2. Zero out elements of A * v that are NOT in Omega.
693 # This requires the inverse mask or iterating through Omega.
694
695 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
696 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
697 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
698 # Need to select elements of X_hat * v at missing indices.
699 # This requires the inverse mask.
700
701 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
702 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
703 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
704 # P_Omega(R) * v = R_orig_csr @ v
705 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
706 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
707 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
708 # This requires iterating through observed locations.
709
710 # Let's use the simpler form for the LinearOperator:
711 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
712
713 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
714 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
715
716 # Let's use the definition based on filling NaNs:
717 # Z * v where Z has NaNs filled with USV^T
718 # This requires a dense matrix multiplication if we fill NaNs.
719
720 # Back to the LinearOperator definition:
721 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
722
723 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
724 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
725 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
726 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
727
728 # Let's use the standard implementation pattern for P_Omega(A) * v:
729 # P_Omega(A) * v = (omega_mask .* A) * v
730 # This is equivalent to:
731 # 1. Compute A * v
732 # 2. Zero out elements of A * v that are NOT in Omega.
733 # This requires the inverse mask or iterating through Omega.
734
735 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
736 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
737 # X_hat * v = self._U @ (self._S * (self._V.T @ v))

```

```

738 # Need to select elements of X_hat * v at missing indices.
739 # This requires the inverse mask.
740
741 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
742 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
743 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
744 # P_Omega(R) * v = R_orig_csr @ v
745 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
746 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
747 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
748 # This requires iterating through observed locations.
749
750 # Let's use the simpler form for the LinearOperator:
751 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
752
753 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
754 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
755
756 # Let's use the definition based on filling NaNs:
757 # Z * v where Z has NaNs filled with USV^T
758 # This requires a dense matrix multiplication if we fill NaNs.
759
760 # Back to the LinearOperator definition:
761 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
762
763 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
764 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
765 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
766 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
767
768 # Let's use the standard implementation pattern for P_Omega(A) * v:
769 # P_Omega(A) * v = (omega_mask .* A) * v
770 # This is equivalent to:
771 # 1. Compute A * v
772 # 2. Zero out elements of A * v that are NOT in Omega.
773 # This requires the inverse mask or iterating through Omega.
774
775 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
776 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
777 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
778 # Need to select elements of X_hat * v at missing indices.
779 # This requires the inverse mask.
780
781 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
782 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
783 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
784 # P_Omega(R) * v = R_orig_csr @ v
785 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
786 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
787 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
788 # This requires iterating through observed locations.
789
790 # Let's use the simpler form for the LinearOperator:
791 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
792
793 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
794 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
795
796 # Let's use the definition based on filling NaNs:
797 # Z * v where Z has NaNs filled with USV^T
798 # This requires a dense matrix multiplication if we fill NaNs.
799
800 # Back to the LinearOperator definition:
801 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
802
803 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
804 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
805 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
806 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
807
808 # Let's use the standard implementation pattern for P_Omega(A) * v:
809 # P_Omega(A) * v = (omega_mask .* A) * v
810 # This is equivalent to:
811 # 1. Compute A * v
812 # 2. Zero out elements of A * v that are NOT in Omega.
813 # This requires the inverse mask or iterating through Omega.
814

```

```

815 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
816 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
817 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
818 # Need to select elements of X_hat * v at missing indices.
819 # This requires the inverse mask.
820
821 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
822 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
823 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
824 # P_Omega(R) * v = R_orig_csr @ v
825 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
826 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
827 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
828 # This requires iterating through observed locations.
829
830 # Let's use the simpler form for the LinearOperator:
831 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
832
833 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
834 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
835
836 # Let's use the definition based on filling NaNs:
837 # Z * v where Z has NaNs filled with USV^T
838 # This requires a dense matrix multiplication if we fill NaNs.
839
840 # Back to the LinearOperator definition:
841 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
842
843 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
844 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
845 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
846 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
847
848 # Let's use the standard implementation pattern for P_Omega(A) * v:
849 # P_Omega(A) * v = (omega_mask .* A) * v
850 # This is equivalent to:
851 # 1. Compute A * v
852 # 2. Zero out elements of A * v that are NOT in Omega.
853 # This requires the inverse mask or iterating through Omega.
854
855 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
856 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
857 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
858 # Need to select elements of X_hat * v at missing indices.
859 # This requires the inverse mask.
860
861 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
862 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
863 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
864 # P_Omega(R) * v = R_orig_csr @ v
865 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
866 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
867 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
868 # This requires iterating through observed locations.
869
870 # Let's use the simpler form for the LinearOperator:
871 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
872
873 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
874 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
875
876 # Let's use the definition based on filling NaNs:
877 # Z * v where Z has NaNs filled with USV^T
878 # This requires a dense matrix multiplication if we fill NaNs.
879
880 # Back to the LinearOperator definition:
881 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
882
883 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
884 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
885 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
886 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
887
888 # Let's use the standard implementation pattern for P_Omega(A) * v:
889 # P_Omega(A) * v = (omega_mask .* A) * v
890 # This is equivalent to:
891 # 1. Compute A * v

```

```

892 # 2. Zero out elements of A * v that are NOT in Omega.
893 # This requires the inverse mask or iterating through Omega.
894
895 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
896 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
897 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
898 # Need to select elements of X_hat * v at missing indices.
899 # This requires the inverse mask.
900
901 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
902 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
903 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
904 # P_Omega(R) * v = R_orig_csr @ v
905 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
906 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
907 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
908 # This requires iterating through observed locations.
909
910 # Let's use the simpler form for the LinearOperator:
911 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
912
913 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
914 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
915
916 # Let's use the definition based on filling NaNs:
917 # Z * v where Z has NaNs filled with USV^T
918 # This requires a dense matrix multiplication if we fill NaNs.
919
920 # Back to the LinearOperator definition:
921 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
922
923 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
924 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
925 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
926 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
927
928 # Let's use the standard implementation pattern for P_Omega(A) * v:
929 # P_Omega(A) * v = (omega_mask .* A) * v
930 # This is equivalent to:
931 # 1. Compute A * v
932 # 2. Zero out elements of A * v that are NOT in Omega.
933 # This requires the inverse mask or iterating through Omega.
934
935 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
936 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
937 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
938 # Need to select elements of X_hat * v at missing indices.
939 # This requires the inverse mask.
940
941 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
942 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
943 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
944 # P_Omega(R) * v = R_orig_csr @ v
945 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
946 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
947 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
948 # This requires iterating through observed locations.
949
950 # Let's use the simpler form for the LinearOperator:
951 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
952
953 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
954 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
955
956 # Let's use the definition based on filling NaNs:
957 # Z * v where Z has NaNs filled with USV^T
958 # This requires a dense matrix multiplication if we fill NaNs.
959
960 # Back to the LinearOperator definition:
961 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
962
963 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
964 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
965 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
966 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
967
968 # Let's use the standard implementation pattern for P_Omega(A) * v:

```



```

969 # P_Omega(A) * v = (omega_mask .* A) * v
970 # This is equivalent to:
971 # 1. Compute A * v
972 # 2. Zero out elements of A * v that are NOT in Omega.
973 # This requires the inverse mask or iterating through Omega.
974
975 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
976 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
977 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
978 # Need to select elements of X_hat * v at missing indices.
979 # This requires the inverse mask.
980
981 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
982 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
983 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
984 # P_Omega(R) * v = R_orig_csr @ v
985 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
986 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
987 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
988 # This requires iterating through observed locations.
989
990 # Let's use the simpler form for the LinearOperator:
991 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
992
993 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
994 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
995
996 # Let's use the definition based on filling NaNs:
997 # Z * v where Z has NaNs filled with USV^T
998 # This requires a dense matrix multiplication if we fill NaNs.
999
1000 # Back to the LinearOperator definition:
1001 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1002
1003 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
1004 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1005 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
1006 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
1007
1008 # Let's use the standard implementation pattern for P_Omega(A) * v:
1009 # P_Omega(A) * v = (omega_mask .* A) * v
1010 # This is equivalent to:
1011 # 1. Compute A * v
1012 # 2. Zero out elements of A * v that are NOT in Omega.
1013 # This requires the inverse mask or iterating through Omega.
1014
1015 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
1016 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
1017 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
1018 # Need to select elements of X_hat * v at missing indices.
1019 # This requires the inverse mask.
1020
1021 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
1022 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
1023 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
1024 # P_Omega(R) * v = R_orig_csr @ v
1025 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
1026 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1027 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
1028 # This requires iterating through observed locations.
1029
1030 # Let's use the simpler form for the LinearOperator:
1031 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1032
1033 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
1034 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
1035
1036 # Let's use the definition based on filling NaNs:
1037 # Z * v where Z has NaNs filled with USV^T
1038 # This requires a dense matrix multiplication if we fill NaNs.
1039
1040 # Back to the LinearOperator definition:
1041 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1042
1043 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
1044 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1045 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v

```

```

1046 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
1047
1048 # Let's use the standard implementation pattern for P_Omega(A) * v:
1049 # P_Omega(A) * v = (omega_mask .* A) * v
1050 # This is equivalent to:
1051 # 1. Compute A * v
1052 # 2. Zero out elements of A * v that are NOT in Omega.
1053 # This requires the inverse mask or iterating through Omega.
1054
1055 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
1056 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
1057 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
1058 # Need to select elements of X_hat * v at missing indices.
1059 # This requires the inverse mask.
1060
1061 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
1062 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
1063 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
1064 # P_Omega(R) * v = R_orig_csr @ v
1065 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
1066 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1067 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
1068 # This requires iterating through observed locations.
1069
1070 # Let's use the simpler form for the LinearOperator:
1071 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1072
1073 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
1074 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
1075
1076 # Let's use the definition based on filling NaNs:
1077 # Z * v where Z has NaNs filled with USV^T
1078 # This requires a dense matrix multiplication if we fill NaNs.
1079
1080 # Back to the LinearOperator definition:
1081 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1082
1083 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
1084 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1085 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
1086 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
1087
1088 # Let's use the standard implementation pattern for P_Omega(A) * v:
1089 # P_Omega(A) * v = (omega_mask .* A) * v
1090 # This is equivalent to:
1091 # 1. Compute A * v
1092 # 2. Zero out elements of A * v that are NOT in Omega.
1093 # This requires the inverse mask or iterating through Omega.
1094
1095 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
1096 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
1097 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
1098 # Need to select elements of X_hat * v at missing indices.
1099 # This requires the inverse mask.
1100
1101 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
1102 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
1103 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
1104 # P_Omega(R) * v = R_orig_csr @ v
1105 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
1106 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1107 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
1108 # This requires iterating through observed locations.
1109
1110 # Let's use the simpler form for the LinearOperator:
1111 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1112
1113 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
1114 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
1115
1116 # Let's use the definition based on filling NaNs:
1117 # Z * v where Z has NaNs filled with USV^T
1118 # This requires a dense matrix multiplication if we fill NaNs.
1119
1120 # Back to the LinearOperator definition:
1121 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1122

```

```

1123 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
1124 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1125 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
1126 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
1127
1128 # Let's use the standard implementation pattern for P_Omega(A) * v:
1129 # P_Omega(A) * v = (omega_mask .* A) * v
1130 # This is equivalent to:
1131 # 1. Compute A * v
1132 # 2. Zero out elements of A * v that are NOT in Omega.
1133 # This requires the inverse mask or iterating through Omega.
1134
1135 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
1136 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
1137 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
1138 # Need to select elements of X_hat * v at missing indices.
1139 # This requires the inverse mask.
1140
1141 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
1142 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
1143 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
1144 # P_Omega(R) * v = R_orig_csr @ v
1145 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
1146 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1147 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
1148 # This requires iterating through observed locations.
1149
1150 # Let's use the simpler form for the LinearOperator:
1151 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1152
1153 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
1154 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
1155
1156 # Let's use the definition based on filling NaNs:
1157 # Z * v where Z has NaNs filled with USV^T
1158 # This requires a dense matrix multiplication if we fill NaNs.
1159
1160 # Back to the LinearOperator definition:
1161 # Z * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1162
1163 # The correct way to implement P_Omega(X_hat) * v for LinearOperator:
1164 # 1. Compute X_hat * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1165 # 2. Compute P_Omega(X_hat) * v = (omega_mask .* X_hat) * v
1166 # This requires computing X_hat_ij for (i,j) in Omega and forming a sparse matrix.
1167
1168 # Let's use the standard implementation pattern for P_Omega(A) * v:
1169 # P_Omega(A) * v = (omega_mask .* A) * v
1170 # This is equivalent to:
1171 # 1. Compute A * v
1172 # 2. Zero out elements of A * v that are NOT in Omega.
1173 # This requires the inverse mask or iterating through Omega.
1174
1175 # Let's use the definition based on R_orig and X_hat directly, but implemented efficiently:
1176 # Y * v = R_orig_csr @ v + (X_hat * v) at missing indices
1177 # X_hat * v = self._U @ (self._S * (self._V.T @ v))
1178 # Need to select elements of X_hat * v at missing indices.
1179 # This requires the inverse mask.
1180
1181 # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
1182 # Z = P_Omega(R) + P_Omega_Complement(USV^T)
1183 # Z * v = P_Omega(R) * v + P_Omega_Complement(USV^T) * v
1184 # P_Omega(R) * v = R_orig_csr @ v
1185 # P_Omega_Complement(USV^T) * v = (USV^T) * v - P_Omega(USV^T) * v
1186 # (USV^T) * v = USVT_v = self._U @ (self._S * (self._V.T @ v))
1187 # P_Omega(USV^T) * v: compute USV^T at observed locations and multiply by v.
1188 # This requires iterating through observed locations.
1189
1190 # Let's use the simpler form for the LinearOperator:
1191 # Y * v = R_orig_csr @ v + (USV^T * v) - (omega_mask_csr @ (USV^T * v)) # Still wrong
1192
1193 # The correct LinearOperator matvec for Y = P_Omega(R) + P_Omega_Complement(X_hat):
1194 # Y * v = R_orig_csr @ v + (X_hat * v) - (omega_mask_csr @ (X_hat * v)) # Still wrong
1195
1196 # Let's use the definition based on filling NaNs:
1197 # Z * v where Z has NaNs filled with USV^T
1198 # This requires a dense matrix multiplication if we fill NaNs.
1199

```

```

1200     # Back to the LinearOperator definition:
1201     #  $Z * v = R_{orig\_csr} @ v + (USV^T * v) - (\omega\_mask\_csr @ (USV^T * v))$  # Still wrong
1202
1203     # The correct way to implement  $P_{\Omega}(X_{hat}) * v$  for LinearOperator:
1204     # 1. Compute  $X_{hat} * v = USV^T v = self\_U @ (self\_S * (self\_V.T @ v))$ 
1205     # 2. Compute  $P_{\Omega}(X_{hat}) * v = (\omega\_mask .* X_{hat}) * v$ 
1206     # This requires computing  $X_{hat\_ij}$  for  $(i,j)$  in  $\Omega$  and forming a sparse matrix.
1207
1208     # Let's use the standard implementation pattern for  $P_{\Omega}(A) * v$ :
1209     #  $P_{\Omega}(A) * v = (\omega\_mask .* A) * v$ 
1210     # This is equivalent to:
1211     # 1. Compute  $A * v$ 
1212     # 2. Zero out elements of  $A * v$  that are NOT in  $\Omega$ .
1213     # This requires the inverse mask or iterating through  $\Omega$ .
1214
1215     # Let's use the definition based on  $R_{orig}$  and  $X_{hat}$  directly, but implemented efficiently:
1216     #  $Y * v = R_{orig\_csr} @ v + (X_{hat} * v)$  at missing indices
1217     #  $X_{hat} * v = self\_U @ (self\_S * (self\_V.T @ v))$ 
1218     # Need to select elements of  $X_{hat} * v$  at missing indices.
1219     # This requires the inverse mask.
1220
1221     # Let's rethink the LinearOperator matvec/rmatvec based on the paper's algorithm:
1222     #  $Z = P_{\Omega}(R) + P_{\Omega\_Complement}(USV^T)$ 
1223     #  $Z * v = P_{\Omega}(R) * v + P_{\Omega\_Complement}(USV^T) * v$ 
1224     #  $P_{\Omega}(R) * v = R_{orig\_csr} @ v$ 
1225     #  $P_{\Omega\_Complement}(U$ 
1226 # -----
1227 # helpers
1228 # -----
1229
1230 def _to_backend(x: Array | Sparse, use_gpu: bool):
1231     """Move *dense* or *sparse* array to the requested backend."""
1232     if use_gpu and not _HAS_CUPY:
1233         raise RuntimeError("CuPy requested but not installed.")
1234
1235     if use_gpu:
1236         if _HAS_CUPY and isinstance(x, (_cp.ndarray | _cpx_sparse.spmatrix)):
1237             return x # already on GPU
1238         return _cp.asarray(x) if not _sp.issparse(x) else _cpx_sparse.csr_matrix(x)
1239     # -> CPU
1240     if isinstance(x, (_np.ndarray | _sp.spmatrix)):
1241         return x
1242     return _cp.asnumpy(x) if not _sp.issparse(x) else _sp.csr_matrix(x.get())
1243
1244
1245 def _soft_threshold(s: Array, lam: float):
1246     return _np.maximum(s - lam, 0.0)
1247
1248
1249 # -----
1250 # main class
1251 # -----
1252 class SoftImpute:
1253     """Matrix completion via nuclear-norm minimisation.
1254
1255     Parameters
1256     -----
1257     lam : float
1258         Regularisation (shrinkage) parameter  $\lambda$ .
1259     max_rank : int | None, optional
1260         Maximum rank of the factorisation. Defaults to  $\min(m, n)$ .
1261     max_iters : int, optional
1262         Maximum number of iterations (default 100).
1263     tol : float, optional
1264         Stop when relative change in Frobenius norm <  $\text{tol}$  (default  $1e-4$ ).
1265     init_fill_method : {"zero", "mean"}
1266         How to fill missing values in the first iteration.
1267     use_gpu : bool, optional
1268         *True* - try CuPy; *False* - force CPU; *None* - auto-detect.
1269     random_state : int | None
1270         RNG seed for reproducible power-iteration initialisation.
1271     return_factors : bool, default False
1272         If *True* return  $(U, S, V)$  instead of the filled matrix.
1273     """
1274
1275     def __init__(
1276         self,

```

```

1277     lam: float = 5.0,
1278     *,
1279     max_rank: Optional[int] = None,
1280     max_iters: int = 100,
1281     tol: float = 1e-4,
1282     init_fill_method: str = "zero",
1283     use_gpu: Optional[bool] = None,
1284     random_state: Optional[int] = None,
1285     return_factors: bool = False,
1286 ) -> None:
1287     self.lam = float(lam)
1288     self.max_rank = max_rank
1289     self.max_iters = int(max_iters)
1290     self.tol = float(tol)
1291     if init_fill_method not in {"zero", "mean"}:
1292         raise ValueError("init_fill_method must be 'zero' or 'mean'")
1293     self.init_fill_method = init_fill_method
1294     self.use_gpu = (_HAS_CUPY if use_gpu is None else bool(use_gpu))
1295     self.rng = default_rng(random_state)
1296     self.return_factors = return_factors
1297
1298     # will be initialised in `fit_transform`
1299     self.U_: Optional[Array] = None
1300     self.S_: Optional[Array] = None
1301     self.V_: Optional[Array] = None
1302
1303 # -----
1304 def fit_transform(self, X: Array | Sparse) -> Array | Tuple[Array, Array, Array]:
1305     """Run Soft-Impute and return the completed matrix or the factors."""
1306
1307     # move data to desired backend
1308     X = _to_backend(X, self.use_gpu)
1309     xp = _cp if (self.use_gpu) else _np
1310     spmod = _cpx_sparse if (self.use_gpu) else _sp
1311
1312     # sparse -> dense with NaNs where missing -----
1313     if spmod.issparse(X):
1314         X = X.tocsr()
1315         m, n = X.shape
1316         dense = xp.full((m, n), xp.nan, dtype=xp.float32)
1317         rows, cols = X.nonzero()
1318         dense[rows, cols] = X.data.astype(xp.float32)
1319         X = dense
1320     else:
1321         X = X.astype(xp.float32)
1322
1323     nan_mask = xp.isnan(X)
1324     m, n = X.shape
1325     max_rank = self.max_rank or min(m, n)
1326
1327     # initial fill -----
1328     X_filled = X.copy()
1329     if self.init_fill_method == "mean":
1330         col_means = xp.nanmean(X, axis=0)
1331         inds = nan_mask
1332         X_filled[inds] = col_means[xp.newaxis, :][inds]
1333     else: # zero
1334         X_filled[nan_mask] = 0.0
1335
1336     # main iteration -----
1337     prev_norm = xp.linalg.norm(X_filled)
1338     for it in range(1, self.max_iters + 1):
1339         # truncated SVD: cpu -> scipy.sparse.linalg.svds; gpu -> full svd of cuPy
1340         if self.use_gpu:
1341             U, S, Vt = xp.linalg.svd(X_filled, full_matrices=False)
1342             U, S, Vt = U[:, :max_rank], S[:max_rank], Vt[:max_rank, :]
1343         else:
1344             # work with float64 for SciPy stability
1345             U, S, Vt = _svds(_sp.csr_matrix(X_filled), k=max_rank, which="LM")
1346             # SciPy returns in ascending order
1347             U, S, Vt = U[:, ::-1], S[::-1], Vt[:, :-1, :]
1348
1349         # soft-threshold singular values -----
1350         S_shrink = _soft_threshold(S, self.lam)
1351         rank_k = int((S_shrink > 0).sum())
1352         if rank_k == 0:
1353             warnings.warn("All singular values shrunk to 0 - returning previous iterate.")

```

```

1354         break
1355         U = U[:, :rank_k]
1356         S_shrink = S_shrink[:rank_k]
1357         Vt = Vt[:rank_k, :]
1358
1359         # reconstruct and impute -----
1360         X_hat = (U * S_shrink) @ Vt # U (mxr) * diag(S) * V^T (rxn)
1361         X_filled[nan_mask] = X_hat[nan_mask]
1362
1363         # convergence check -----
1364         frob_norm = xp.linalg.norm(X_filled)
1365         rel_change = xp.linalg.norm(X_filled - X_hat) / max(1.0, frob_norm)
1366         if rel_change < self.tol:
1367             break
1368         prev_norm = frob_norm
1369
1370         # store factors on CPU for compat -----
1371         self.U_ = _cp.asnumpy(U) if self.use_gpu else U
1372         self.S_ = _cp.asnumpy(S_shrink) if self.use_gpu else S_shrink
1373         self.V_ = _cp.asnumpy(Vt.T) if self.use_gpu else Vt.T
1374
1375         if self.return_factors:
1376             return self.U_, self.S_, self.V_
1377         return _cp.asnumpy(X_filled) if self.use_gpu else X_filled
1378
1379     # -----
1380     def transform(self, X_new: Array | Sparse) -> Array:
1381         """Impute a *new* matrix with the learnt factors (no retraining)."""
1382         if self.U_ is None:
1383             raise RuntimeError("call fit_transform first")
1384         X_new = _to_backend(X_new, self.use_gpu)
1385         xp = _cp if self.use_gpu else _np
1386         dense = X_new.copy()
1387         nan_mask = xp.isnan(dense)
1388         X_hat = (self.U_ * self.S_) @ self.V_.T
1389         dense[nan_mask] = X_hat[nan_mask]
1390         return _cp.asnumpy(dense) if self.use_gpu else dense
1391 # ===== #
1392 # CELL 7: Run Solvers and Compare Results - Renumbered
1393 # ===== #
1394 logger.info("+++ Cell 7: Running Solvers and Comparing Results +++")
1395
1396 #all_results = {}
1397 # --- Initialize Trajectory Cache (Rank 0 only) ---
1398 TRAJECTORY_CACHE = [] if RANK_MPI == 0 else None
1399
1400 # --- Update solver_args with new variable names ---
1401 solver_args = {
1402     "R_train_coo": R_train_coo, "global_mean": global_mean_rating,
1403     "probe_users_mapped": user_ids_val_final, "probe_movies_mapped": movie_ids_val_final,
1404     "probe_ratings_true": ratings_val_true, "N_users_active": N_users_active,
1405     "M_movies_active": M_movies_active, "rank_local": RANK, "lam_sq": LAM_SQ,
1406     "lam_bias": LAM_BIAS, "rng": GLOBAL_RNG, "init_scale": INIT_SCALE_NON_CONVEX,
1407 }
1408
1409 # --- Run Non-Convex Solvers ---
1410 if DATA_AVAILABLE and R_train_coo.nnz > 0 and N_users_active > 0 and M_movies_active > 0:
1411     # Euclidean GD (NEW)
1412     if RANK_MPI == 0: logger.info("\n--- Running Non-Convex Solver (Euclidean GD with Biases) ---")
1413     try: all_results['Non-Convex (EucGD+Bias)'] = run_euclidean_gd(**solver_args, n_iters=N_ITERS_ALL, lr=1e-7) # Added call, specify
1414     except Exception as e: logger.error(f"EucGD Failed: {e}", exc_info=True); all_results['Non-Convex (EucGD+Bias)'] = {}
1415     # SVRG
1416     if RANK_MPI == 0: logger.info("\n--- Running Non-Convex Solver (SVRG Adaptation with Biases) ---")
1417     try: all_results['Non-Convex (SVRG+Bias)'] = run_non_convex_svr_gd_with_biases(**solver_args, n_epochs=N_ITERS_ALL, inner_lr=INIT_LR)
1418     except Exception as e: logger.error(f"SVRG Failed: {e}", exc_info=True); all_results['Non-Convex (SVRG+Bias)'] = {}
1419     # ALS
1420     if RANK_MPI == 0: logger.info("\n--- Running Non-Convex Solver (ALS with Biases) ---")
1421     try: all_results['Non-Convex (ALS+Bias)'] = run_als_with_biases(**solver_args, n_iters=N_ITERS_ALL, tol=ALS_TOL)
1422     except Exception as e: logger.error(f"ALS Failed: {e}", exc_info=True); all_results['Non-Convex (ALS+Bias)'] = {}
1423     # RGD
1424     if RANK_MPI == 0: logger.info("\n--- Running Non-Convex Solver (RGD with Biases) ---")
1425     try: all_results['Non-Convex (RGD+Bias)'] = run_rgd_with_biases(**solver_args, n_iters=N_ITERS_ALL, lr_init=INIT_LR_RIEMANN, ls_t
1426     except Exception as e: logger.error(f"RGD Failed: {e}", exc_info=True); all_results['Non-Convex (RGD+Bias)'] = {}
1427     # RAGD
1428     if RANK_MPI == 0: logger.info("\n--- Running Non-Convex Solver (RAGD with Biases) ---")
1429     try: all_results['Non-Convex (RAGD+Bias)'] = run_ragd_with_biases(**solver_args, n_iters=N_ITERS_ALL, lr_init=INIT_LR_RIEMANN, ls_t
1430     except Exception as e: logger.error(f"RAGD Failed: {e}", exc_info=True); all_results['Non-Convex (RAGD+Bias)'] = {}

```

```

1431 # Catalyst + Selected Inner Solver
1432 if RANK_MPI == 0: logger.info(f"\n--- Running Non-Convex Solver (Catalyst+{INNER_SOLVER.upper()}) with Biases) ---")
1433 try: all_results[f'Non-Convex (Catalyst+{INNER_SOLVER.upper()})'] = run_catalyst_stochastic(**solver_args, n_iters=N_ITERS_ALL, l
1434 except Exception as e: logger.error(f"Catalyst+{INNER_SOLVER.upper()}) Failed: {e}", exc_info=True); all_results[f'Non-Convex (Cat
1435 # DANE
1436 if RANK_MPI == 0: logger.info("\n--- Running Non-Convex Solver (DANE with Biases) ---")
1437 try: all_results['Non-Convex (DANE+Bias)'] = run_dane_with_biases(**solver_args, n_iters=N_ITERS_ALL, lr_init=INIT_LR_RIEMANN, l:
1438 except Exception as e: logger.error(f"DANE Failed: {e}", exc_info=True); all_results['Non-Convex (DANE+Bias)'] = {}
1439 else:
1440     if RANK_MPI == 0: logger.warning("Skipping Non-Convex Solvers due to missing data or zero dimensions.")
1441
1442 # --- Run Convex Solver (Efficient Soft-Impute) ---
1443 if DATA_AVAILABLE and R_train_coo_orig.nnz > 0 and N_users_active > 0 and M_movies_active > 0:
1444     if RANK_MPI == 0: logger.info("\n--- Running Convex Solver (Efficient Soft-Impute) ---")
1445     try:
1446         results_convex = run_soft_impute_efficient(
1447             R_train_coo_orig=R_train_coo_orig, # Use original ratings matrix
1448             probe_users_mapped=user_ids_val_final,
1449             probe_movies_mapped=movie_ids_val_final,
1450             probe_ratings_true=ratings_val_true, # Use validation ratings
1451             N_users_active=N_users_active,
1452             M_movies_active=M_movies_active,
1453             n_iters=N_ITERS_ALL, # Use N_ITERS_ALL for consistency
1454             lambda_reg=LAM, # Use LAM directly
1455             k_rank = CONVEX_RANK_K,
1456             tol=SOFT_IMPUTE_TOL,
1457             rng=GLOBAL_RNG
1458         )
1459         all_results['Convex (SoftImpute Eff.)'] = results_convex
1460     except Exception as e:
1461         logger.error(f"Failed to run Efficient Soft-Impute Solver: {e}", exc_info=True)
1462         all_results['Convex (SoftImpute Eff.)'] = {'loss': [], 'rmse': [], 'time': [], 'rank': []}
1463 else:
1464     if RANK_MPI == 0: logger.warning("Skipping Convex Solver due to missing data or zero dimensions.")
1465     all_results['Convex (SoftImpute Eff.)'] = {'loss': [], 'rmse': [], 'time': [], 'rank': []}
1466
1467
1468 # --- Plotting Comparison ---
1469 if RANK_MPI == 0:
1470     logger.info("\n--- Generating Comparison Plots ---")
1471     plt.style.use('seaborn-v0_8-whitegrid')
1472     fig, axes = plt.subplots(3, 2, figsize=(12, 11), sharex='col')
1473     fig.suptitle(
1474         f'MovieLens 1M ({RATING_LIMIT/1e6 if RATING_LIMIT else "Full"} M ratings subset), '
1475         f'Rank={RANK}, Outer iters={N_ITERS_ALL}',
1476         fontsize=14,
1477     )
1478
1479 # ----- style dictionary (matches earlier section) -----
1480 styles = {
1481     'Non-Convex (SVRG+Bias)': dict(label=r'SVRG+Bias', style=(-, 'p'), alpha=.90, color='tab:purple'),
1482     'Non-Convex (ALS+Bias)': dict(label=r'ALS+Bias', style=(-, 'v'), alpha=.90, color='tab:brown'),
1483     'Non-Convex (RGD+Bias)': dict(label=r'RGD+Bias', style=(-, 'o'), alpha=.80, color='tab:blue'),
1484     'Non-Convex (RAGD+Bias)': dict(label=r'RAGD+Bias', style=(-, 'D'), alpha=.80, color='tab:orange'),
1485     f'Non-Convex (Catalyst+{INNER_SOLVER.upper()})': dict(label=f'Catalyst+{INNER_SOLVER.upper()}', style=(-, 's'), alpha=.90),
1486     'Non-Convex (DANE+Bias)': dict(label=r'DANE+Bias', style=(-, 'x'), alpha=.80, color='tab:cyan'),
1487     'Non-Convex (EucGD+Bias)': dict(label=r'EucGD+Bias', style=(-, '^'), alpha=.70, color='tab:green'),
1488     'Convex (SoftImpute Eff.)': dict(label=r'SoftImpute (Eff)', style=(-, '*'), alpha=.90, color='tab:pink'),
1489 }
1490
1491 # ----- helper for plotting one method -----
1492 def _plot(ax_iter, ax_time, data, meta):
1493     ls, mk = meta['style']
1494     kw = dict(linestyle=ls, marker=mk, markersize=3, alpha=meta['alpha'], color=meta.get('color', None))
1495     n_loss = len(data.get('loss', [])); n_grad = len(data.get('grad_norm', [])); n_rmse = len(data.get('rmse', [])); n_time = len
1496     n = min(n_loss if n_loss > 0 else float('inf'), n_grad if n_grad > 0 else float('inf'), n_rmse if n_rmse > 0 else float('inf
1497     if n == float('inf') or n < 2: logger.warning(f" * insufficient points for {meta['label']}"); return
1498
1499     it = np.arange(n)
1500     loss_vals = np.array(data.get('loss', [np.nan]*n)[n]); grad_vals = np.array(data.get('grad_norm', [np.nan]*n)[n])
1501     rmse_vals = np.array(data.get('rmse', [np.nan]*n)[n]); time_vals = np.array(data.get('time', [np.nan]*n)[n])
1502
1503     # Determine primary metric for grad plot (grad_norm, or gU_norm for SVRG)
1504     grad_metric = grad_vals
1505     if not np.any(np.isfinite(grad_metric)) and 'gU_norm' in data:
1506         grad_metric = np.array(data.get('gU_norm', [np.nan]*n)[n])
1507

```

```

1508     loss_ok = np.isfinite(loss_vals); grad_ok = np.isfinite(grad_metric); rmse_ok = np.isfinite(rmse_vals); time_ok = np.isfinite
1509
1510     # iteration domain
1511     if np.any(loss_ok): ax_iter[0].semilogy(it[loss_ok], loss_vals[loss_ok], label=meta['label'], **kw)
1512     if np.any(grad_ok): ax_iter[1].semilogy(it[grad_ok], grad_metric[grad_ok], **kw)
1513     if np.any(rmse_ok): ax_iter[2].plot(it[rmse_ok], rmse_vals[rmse_ok], **kw)
1514
1515     # wall-clock domain
1516     if np.any(loss_ok & time_ok): ax_time[0].semilogy(time_vals[loss_ok & time_ok], loss_vals[loss_ok & time_ok], **kw)
1517     if np.any(grad_ok & time_ok): ax_time[1].semilogy(time_vals[grad_ok & time_ok], grad_metric[grad_ok & time_ok], **kw)
1518     if np.any(rmse_ok & time_ok): ax_time[2].plot(time_vals[rmse_ok & time_ok], rmse_vals[rmse_ok & time_ok], **kw)
1519
1520 # ----- draw every available method -----
1521 for m, d in all_results.items():
1522     if m in styles and d: # Check if history dict is not empty
1523         _plot(axes[:, 0], axes[:, 1], d, styles[m])
1524     else:
1525         logger.warning(f" • no style or no results for '{m}', skipped.")
1526
1527 # labels / titles
1528 axes[0,0].set_ylabel('Objective'); axes[0,0].set_title('Loss vs Iterations')
1529 axes[1,0].set_ylabel(r'$\|\nabla\|$'); axes[1,0].set_title('Grad-norm vs Iterations')
1530 axes[2,0].set_ylabel('Validation RMSE'); axes[2,0].set_xlabel('Iteration k'); axes[2,0].set_title('RMSE vs Iterations')
1531 axes[0,1].set_xscale('log'); axes[0,1].set_ylabel('Objective'); axes[0,1].set_title('Loss vs Wall-time')
1532 axes[1,1].set_xscale('log'); axes[1,1].set_ylabel(r'$\|\nabla\|$'); axes[1,1].set_title('Grad-norm vs Wall-time')
1533 axes[2,1].set_xscale('log'); axes[2,1].set_ylabel('Validation RMSE'); axes[2,1].set_xlabel('Seconds'); axes[2,1].set_title('RMSE
1534
1535 for ax in axes.flatten():
1536     ax.grid(True, which='both', linestyle=':', linewidth=.5)
1537     handles, labels = ax.get_legend_handles_labels()
1538     if handles: ax.legend() # Only add legend if there are labeled artists
1539
1540 plt.tight_layout(rect=[0, 0.03, 1, 0.95])
1541 plt.show()
1542
1543 # ----- optional PCA trajectory plot -----
1544 if PCA_AVAILABLE and TRAJECTORY_CACHE is not None and len(TRAJECTORY_CACHE) >= 3:
1545     logger.info("\n+++ Generating PCA Trajectory Plot +++")
1546     try:
1547         traj_dim = TRAJECTORY_CACHE[0].size
1548         valid_traj = [t for t in TRAJECTORY_CACHE if isinstance(t, np.ndarray) and t.size == traj_dim]
1549         if len(valid_traj) >= 3:
1550             pcs = PCA(n_components=2).fit_transform(np.vstack(valid_traj))
1551             plt.figure(figsize=(4.5,4)); plt.plot(pcs[:,0], pcs[:,1], '-o', markersize=3)
1552             plt.title('Optimisation Trajectory (PCA)'); plt.xlabel('PC1'); plt.ylabel('PC2')
1553             plt.tight_layout(); plt.show()
1554         else: logger.warning("Not enough valid trajectory points for PCA plot.")
1555     except Exception as e_pca: logger.error(f"PCA Trajectory plot failed: {e_pca}")
1556
1557
1558 # --- Final Summary Table ---
1559 if RANK_MPI == 0:
1560     logger.info("\n--- Final Comparison Summary ---")
1561     print(f"{'Method':<30} | {'Final RMSE':<15} | {'Final Loss':<15} | {'Final Rank/GradNorm':<18} | {'Time (s)':<15}")
1562     print(f"{'-'*30}-|{'-'*15}-|{'-'*15}-|{'-'*18}-|{'-'*15}")
1563     def get_last_finite(history, key):
1564         if not isinstance(history, dict): return np.nan
1565         data = history.get(key)
1566         if isinstance(data, (list, np.ndarray)) and len(data) > 0:
1567             arr = np.array(data); finite_vals = arr[np.isfinite(arr)]
1568             return finite_vals[-1] if finite_vals.size > 0 else np.nan
1569         return np.nan
1570     for label, history in all_results.items():
1571         if not history: print(f"{label:<30} | {'FAILED':<15} | {'FAILED':<15} | {'N/A':<18} | {'N/A':<15}"); continue
1572         final_rmse = get_last_finite(history, 'rmse')
1573         final_loss = get_last_finite(history, 'loss')
1574         final_time = get_last_finite(history, 'time')
1575         final_rank = get_last_finite(history, 'rank') if 'rank' in history else RANK
1576         final_grad_norm = get_last_finite(history, 'grad_norm') if 'grad_norm' in history else np.nan
1577         final_gU_norm = get_last_finite(history, 'gU_norm') if 'gU_norm' in history else np.nan
1578         rmse_str = f"{final_rmse:.6f}" if np.isfinite(final_rmse) else 'NaN'
1579         loss_str = f"{final_loss:.6e}" if np.isfinite(final_loss) and 'ALS' not in label and 'SoftImpute' not in label else 'N/A'
1580         rank_or_grad_str = 'N/A'
1581         if 'SoftImpute' in label: rank_or_grad_str = f"Rank={int(final_rank)}" if np.isfinite(final_rank) else 'N/A'
1582         elif 'grad_norm' in history and np.isfinite(final_grad_norm): rank_or_grad_str = f"||G||={final_grad_norm:.2e}"
1583         elif 'gU_norm' in history and np.isfinite(final_gU_norm): rank_or_grad_str = f"||gU||={final_gU_norm:.2e}"
1584         else: rank_or_grad_str = f"Rank={RANK}"

```



```

1585     time_str = f"{final_time:.4f}" if np.isfinite(final_time) else 'N/A'
1586     print(f"{label:<30} | {rmse_str:<15} | {loss_str:<15} | {rank_or_grad_str:<18} | {time_str:<15}")
1587     print("\nComparison Complete.")
1588
1589 # --- ADDED Block 6-a: Run OT Demo (Rank 0 only) ---
1590 # --- ADDED Block 6-a: Run OT Demo (Rank 0 only) ---
1591 if RANK_MPI == 0 and OT_AVAILABLE:
1592     logger.info("\n+++ Running OT Barycentre Demo +++")
1593     try:
1594         ot_demo_results = run_barycentre_demo()
1595         # Optionally plot or process ot_demo_results
1596         plt.figure(figsize=(6, 4))
1597         plt.plot(ot_demo_results['grid'], ot_demo_results['sources'], '--', label='Sources')
1598         plt.plot(ot_demo_results['grid'], ot_demo_results['barycenter'], 'r-', label='Barycenter')
1599         plt.title('Wasserstein Barycenter Demo')
1600         plt.legend(); plt.tight_layout(); plt.show()
1601     except Exception as e_ot:
1602         logger.error(f"OT Barycentre Demo failed: {e_ot}")
1603
1604 # === ADDED Block 6: PCA Trajectory Plot (Rank 0 only) ===
1605 if RANK_MPI == 0 and PCA_AVAILABLE and len(TRAJECTORY_CACHE) >= 3:
1606     logger.info("\n+++ Generating PCA Trajectory Plot +++")
1607     try:
1608         # Ensure all trajectories have the same dimension (flattened U)
1609         traj_dim = TRAJECTORY_CACHE[0].size
1610         valid_traj = [t for t in TRAJECTORY_CACHE if t.size == traj_dim]
1611         if len(valid_traj) >= 3:
1612             pcs = PCA(n_components=2).fit_transform(np.vstack(valid_traj))
1613             plt.figure(figsize=(4.5,4)); plt.plot(pcs[:,0], pcs[:,1], '-o', markersize=3)
1614             plt.title('Optimisation Trajectory (PCA)'); plt.xlabel('PC1'); plt.ylabel('PC2')
1615             plt.tight_layout(); plt.show()
1616         else:
1617             logger.warning("Not enough valid trajectory points for PCA plot.")
1618     except Exception as e_pca:
1619         logger.error(f"PCA Trajectory plot failed: {e_pca}")
1620
1621 # === ADDED Block 7: Dump TeX skeleton to Drive (Rank 0 only) ===
1622 if RANK_MPI == 0:
1623     TEX_PATH = Path(DATA_DIR_STR) / "proofs.tex" # Use Path object
1624     if TEX_PATH.parent.is_dir():
1625         logger.info(f"\n+++ Checking/Writing TeX Proof Skeleton to: {TEX_PATH} +++")
1626         if not TEX_PATH.exists():
1627             try:
1628                 with open(TEX_PATH, "w") as f: f.write(r""""...""") # TeX content omitted for brevity
1629                 logger.info(f" Wrote TeX scaffold to {TEX_PATH}")
1630             except IOError as e: logger.error(f" Error writing TeX file: {e}")
1631             else: logger.info(f" TeX scaffold already exists at {TEX_PATH}, not overwritten.")
1632         else: logger.warning(f" Parent directory for TeX file not found: {TEX_PATH.parent}")
1633
1634 # ===== #
1635 # CELL 8: Plots & Dashboards (from long.txt) - Renumbered
1636 # ===== #
1637 if RANK_MPI == 0:
1638     logger.info("\n+++ Cell 8: Plots & Dashboards +++")
1639
1640 # ----- helper ----- #
1641 def _plot_metric(metric_key: str,
1642                  ylabel: str,
1643                  x_key: str = "time",
1644                  title: str | None = None,
1645                  logy: bool = False,
1646                  logx: bool = True,          # Default: log time axis
1647                  figsize=(8, 5)) -> None:
1648     plt.figure(figsize=figsize)
1649     has_data_to_plot = False
1650
1651     # style dictionary -----
1652     styles = {
1653         'Non-Convex (SVRG+Bias)': dict(label='SVRG+Bias', style='-', 'p'), alpha=.90, color='tab:purple'),
1654         'Non-Convex (ALS+Bias)': dict(label='ALS+Bias', style='-', 'v'), alpha=.90, color='tab:brown'),
1655         'Non-Convex (RGD+Bias)': dict(label='RGD+Bias', style='--', 'o'), alpha=.80, color='tab:blue'),
1656         'Non-Convex (RAGD+Bias)': dict(label='RAGD+Bias', style='-.', 'D'), alpha=.80, color='tab:orange'),
1657         f'Non-Convex (Catalyst+{INNER_SOLVER.upper()})':
1658             dict(label=f'Catalyst+{INNER_SOLVER.upper()}',
1659                  style='-', 's'), alpha=.90, color='tab:red'),
1660         'Non-Convex (DANE+Bias)': dict(label='DANE+Bias', style='-', 'x'), alpha=.80, color='tab:cyan'),
1661         'Non-Convex (EucGD+Bias)': dict(label='EucGD+Bias', style=':', '^'), alpha=.70, color='tab:green'),

```

```

1662     'Convex (SoftImpute Eff.)':    dict(label='SoftImpute (Eff)',style='-', '*'), alpha=.90, color='tab:pink'),
1663 }
1664
1665 # loop over solver results -----
1666 for name, res in all_results.items():
1667     y = res.get(metric_key, [])
1668     x = res.get(x_key, list(range(len(y)))) if x_key else list(range(len(y)))
1669
1670     if len(y) == 0:
1671         continue
1672
1673     x = np.asarray(x, dtype=float)
1674     y = np.asarray(y, dtype=float)
1675     valid = np.isfinite(x) & np.isfinite(y)
1676     x_plot, y_plot = x[valid], y[valid]
1677
1678     if x_plot.size == 0:
1679         logger.warning(f"No finite data to plot for {name} - {metric_key}")
1680         continue
1681
1682     style = styles.get(name, {})
1683     plt.plot(
1684         x_plot, y_plot,
1685         linestyle=style.get('style', ('-', 'o'))[0],
1686         marker=style.get('style', ('-', 'o'))[1],
1687         markersize=3,
1688         alpha=style.get('alpha', 0.8),
1689         color=style.get('color'),
1690         label=style.get('label', name)
1691     )
1692     has_data_to_plot = True
1693
1694 # axes / formatting -----
1695 plt.xlabel("wall-clock (s)" if x_key == "time" else "iteration")
1696 plt.ylabel(ylabel)
1697 if logx:

```