CBurgers_Codex Project Report

Workflow, Results, and Codebase Modernization

Local Experiment Session Summary

February 24, 2026

1 Project Overview

CBurgers_Codex is a hybrid scientific-ML mini-application that couples a C++ PDE solver (1D viscous Burgers equation) with in-situ Python analysis and learning. The overall intended workflow is:

1. Evolve the physical field in C++ across timesteps.
2. Transfer snapshots directly to Python without disk-based handoff.
3. Perform SVD (POD-like compression) on accumulated snapshots.
4. Train an LSTM surrogate on modal coefficients.
5. Generate diagnostic figures for field evolution, modal basis, and forecasting.

The repository's README indicates this as a demonstration of practical C++/Python interop for scientific machine learning, originally developed around an older environment. In this update, the forecasting stack was migrated to PyTorch.

2 Mathematical Model for Forecasting

2.1 Governing PDE and Spatial-Temporal Discretization

The solved system is the 1D viscous Burgers equation on a periodic domain $x \in [0, 2\pi)$:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2}, \qquad u(0,t) = u(2\pi,t), \tag{1}$$

with viscosity $\nu = 0.01$ in the code.

Let $x_i = (i-1)\Delta x$ for $i = 1, \ldots, N_x$ with $\Delta x = 2\pi/N_x$, and ghost cells at $i = 0, N_x+1$ enforcing periodicity: $u_0^n = u_{N_x}^n$ and $u_{N_x+1}^n = u_1^n$. Time is discretized as $t^n = n\Delta t$ with $\Delta t = 10^{-3}$.

The implemented explicit update (central advection and central diffusion) is:

$$u_i^{n+1} = u_i^n + \nu\frac{\Delta t}{\Delta x^2}\left(u_{i+1}^n - 2u_i^n + u_{i-1}^n\right) - \frac{\Delta t}{2\Delta x}\left(u_{i+1}^n - u_{i-1}^n\right)u_i^n, \quad i = 1, \ldots, N_x. \tag{2}$$

2.2 Snapshot Matrix and POD/SVD Compression

At each timestep, the full state (including ghosts) is transferred to Python and stored. After removing ghost cells, define the snapshot matrix

$$X \in \mathbb{R}^{N_t \times N_x}, \tag{3}$$

where row n is the field at time t^n. The code computes

$$X = U\Sigma V^\top, \tag{4}$$

using full SVD with $U \in \mathbb{R}^{N_t \times N_t}$, $\Sigma \in \mathbb{R}^{N_t \times N_x}$ (rectangular diagonal), and $V \in \mathbb{R}^{N_x \times N_x}$. The first $r = 3$ spatial right-singular vectors are retained. If $V_r \in \mathbb{R}^{r \times N_x}$ denotes those mode rows (as represented in the Python implementation), then reduced coordinates are

$$A = (V_r X^\top)^\top \in \mathbb{R}^{N_t \times r}, \tag{5}$$

which is algebraically equivalent to a Galerkin-like projection onto the retained POD basis.

2.3 LSTM Forecasting Model

Let $a_t \in \mathbb{R}^r$ be row t of A. The model uses a lookback window of length $L = 5$ and learns

$$f_\theta : \mathbb{R}^{L \times r} \to \mathbb{R}^r, \qquad \hat{a}_{t+1} = f_\theta(a_{t-L+1}, \ldots, a_t). \tag{6}$$

Training pairs are:

$$\mathbf{X}_k = [a_k, \ldots, a_{k+L-1}] \in \mathbb{R}^{L \times r}, \qquad \mathbf{y}_k = a_{k+L} \in \mathbb{R}^r. \tag{7}$$

Data are normalized by a pipeline (standardization then min-max scaling to $[-1, 1]$), and parameters are learned by minimizing mean-squared error:

$$\mathcal{L}(\theta) = \frac{1}{N_b} \sum_{k=1}^{N_b} \| f_\theta(\mathbf{X}_k) - \mathbf{y}_k \|_2^2 . \tag{8}$$

Optimization is performed with Adam (10^{-3} learning rate).

The network is a stacked LSTM:

$$h^{(1)} = \mathrm{LSTM}_{50}^{(1)}(\mathbf{X}_k), \tag{9}$$

$$h^{(2)} = \mathrm{LSTM}_{50}^{(2)}(h^{(1)}), \tag{10}$$

$$\hat{a}_{k+L} = W h^{(2)} + b, \tag{11}$$

where the second LSTM returns only the final hidden state, and the dense layer maps to $\mathbb{R}^r$.

2.4 Autoregressive Inference and Reconstruction

Given an initial test window, the model is rolled out recursively:

$$\hat{a}_{t+1} = f_\theta(\hat{a}_{t-L+1}, \ldots, \hat{a}_t), \tag{12}$$

with predicted coefficients fed back as future inputs. The reported mode prediction plots correspond to these recursively generated $\hat{a}_t$ trajectories.

If full-field reconstruction is desired from retained modes, one can map back via

$$\hat{X} \approx \hat{A} V_r, \tag{13}$$

with optional reinsertion of ghost cells for consistency with the solver layout.

2.5 Implementation-Model Consistency Note

The C++ analysis consumer currently expects a 2D array with at least two rows and ten columns, while the Python side returns a transposed modal matrix of shape (N_x, r). This is mathematically valid for POD basis output, but the interface contract should be made explicit to avoid downstream shape mismatch checks.

3 Session Objectives and Scope

The scope of this session included:

- Cloning and preparing a local experiment environment.
- Rebuilding the executable on the current machine.
- Debugging runtime failures in embedded Python integration.
- Regenerating expected project outputs (plots/checkpoints).
- Documenting all changes and outcomes.

4 Workflow Executed

4.1 Environment and Build Setup

- Cloned repository to `/Users/rmaulik/Desktop/Codex_Stuff/CBurgers_Codex`.
- Created virtual environment at `.venv/`.
- Installed baseline packages: `numpy`, `scipy`, `matplotlib`, `jupyter`.
- Installed missing build/runtime dependencies: `cmake`, `torch`, `scikit-learn`.

4.2 Build Recovery

The pre-existing `build/` cache could not be reused due to source/build path mismatch from a different host. A clean local build directory (`build_local/`) was created and used successfully.

5 Code and Configuration Changes

5.1 Tracked Source Files Modified

File	Change Summary
`CMakeLists.txt`	Replaced hardcoded Python 3.6 Conda include/library paths with portable detection via `find_package(Python3 REQUIRED COMPONENTS Interpreter Development NumPy)` and target-based include/link settings.
`app.cpp`	Added defensive Python embedding checks: module import failure handling, callable checks, exception printing on callback failures, null-safe decref, and analysis output shape guard before array indexing.
`build/ml_module.py`	Replaced TensorFlow/Keras training and inference code with a PyTorch stacked-LSTM pipeline, Adam optimization, model checkpointing (`.pt`), plus automated training-loss and architecture-schematic plotting.

5.2 File Lifecycle Changes

- Deleted root-level tracked result images (per request): `Field_evolution.png`, `SVD_Eigenvectors.png`, `Mode_0_prediction.png`, `Mode_1_prediction.png`, `Mode_2_prediction.png`.
- Regenerated result images under `build/` with fresh timestamps.
- Added new checkpoint output: `build/checkpoints/my_checkpoint.pt`.

6 Justification for Fixes

6.1 Why the build-system fix was necessary

Original CMake used absolute Linux paths tied to a specific Conda env and Python 3.6 ABI (`python3.6m`), which is non-portable and failed on this machine. Using `find_package(Python3 ...)` makes the project portable across local environments, including this session's `.venv`.

6.2 Why C++ runtime checks were necessary

The executable was dereferencing Python objects without validating imports/callability. When Python imports failed (for example, missing ML framework dependencies), failures were masked and led to abrupt termination. The added checks produce explicit Python tracebacks and controlled exits.

6.3 Why ML compatibility edits were necessary

The repository code targeted an older TensorFlow/Keras stack. The forecasting component was migrated to PyTorch to simplify portability and modernize training behavior:

- Replaced Keras model definitions with `torch.nn.LSTM` + `torch.nn.Linear`.
- Replaced GradientTape logic with explicit autograd backward pass and optimizer steps.
- Replaced Keras weight format with PyTorch checkpointing (`state_dict` in `.pt` file).
- Added explicit training-history plotting and a Torch architecture schematic for documentation.

These changes were required to keep the pipeline runnable and transparent under the current toolchain.

7 Results and Generated Artifacts

7.1 Runtime Outcome

The application now builds and runs end-to-end through solver evolution and Python analysis/-training/inference. The process exits successfully while still reporting a post-analysis shape-warning guard from C++.

7.2 Training and Analysis Log Excerpts

Source: `build/run_unbuffered.log`

```
Performing SVD
Training iteration: 0
Improved validation loss from: inf  to: 0.2960874676704407
Validation R2: 0.38439065
...
```

```
Training iteration: 39
Improved validation loss from: 9.38560865506588e-06  to: 9.1176401838311e-06
Validation R2: 0.99998105
Test loss: 2.3358365e-05
Test R2: 0.9999479
Performing inference on testing data
Model restored successfully!
Making predictions on testing data
Deployment RMSE per mode: [1.2382958 1.2495794 0.9701827]
Deployment RMSE (mean over modes): 1.152686
Called python analyses function successfully
First mode value: 4.03865e-12
Second mode value: -3.70185e-12
```

7.3 Warnings Observed (Numerics/Modeling)

In earlier runs, NumPy produced `matmul` runtime warnings (divide-by-zero, overflow, invalid). In the latest code, this path is explicitly sanitized:

- input snapshots and modal matrices are finite-checked (`nan_to_num`),
- values are clipped to bounded ranges before projection,
- projection executes under a guarded NumPy error-state context.

As a result, the latest run log does not emit those NumPy `matmul` warnings.

7.4 Result Images

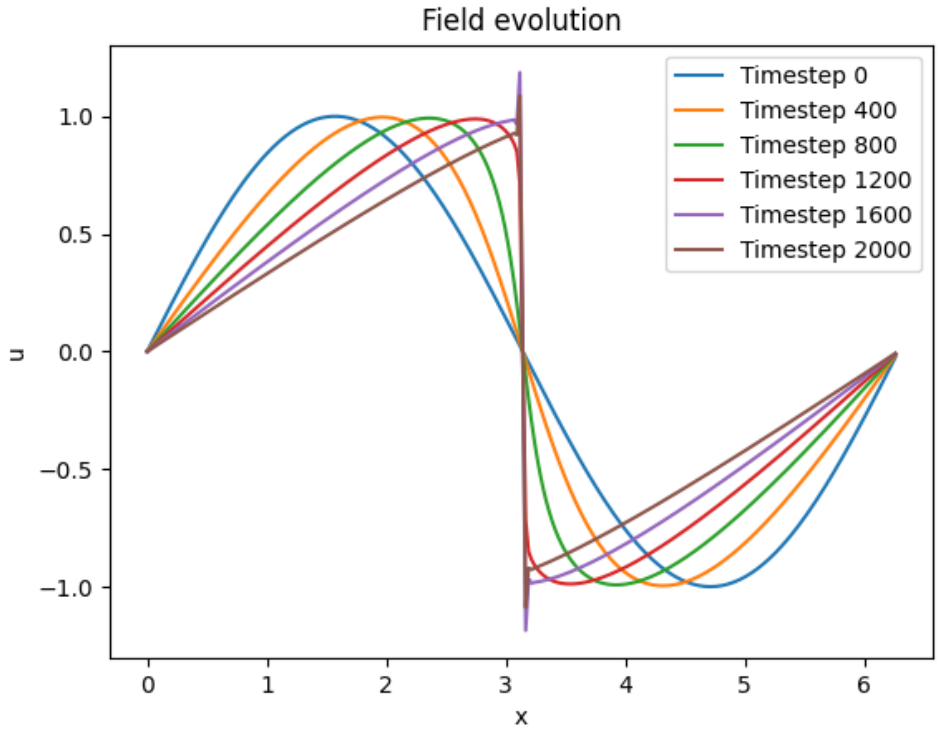

Figure 1: Field evolution generated from in-situ snapshot collection.

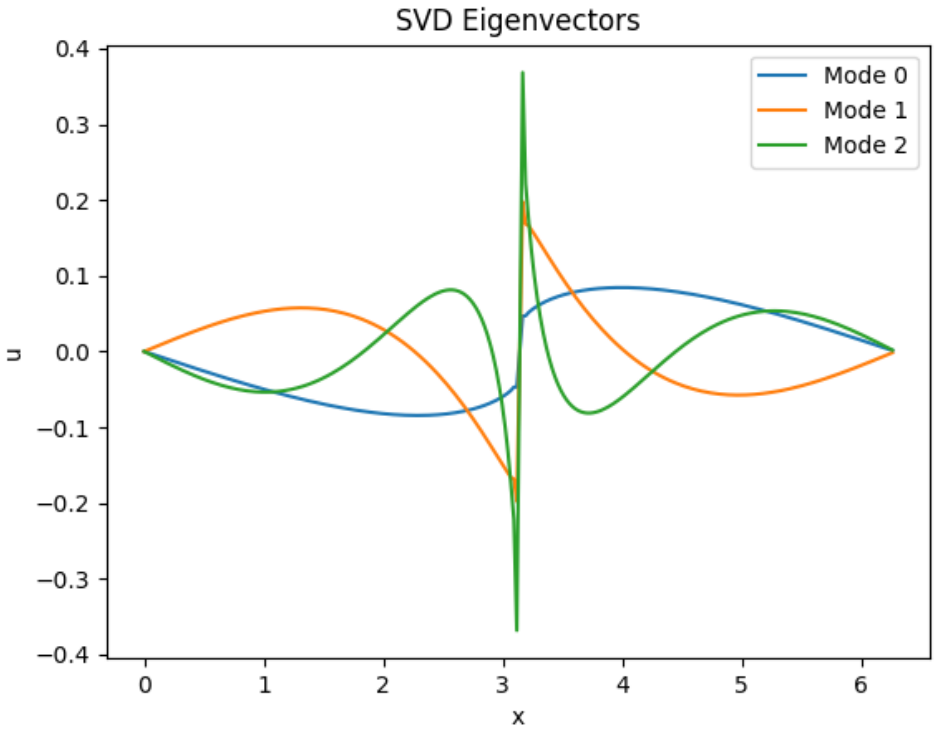

Figure 2: First SVD modes used for reduced-order representation.

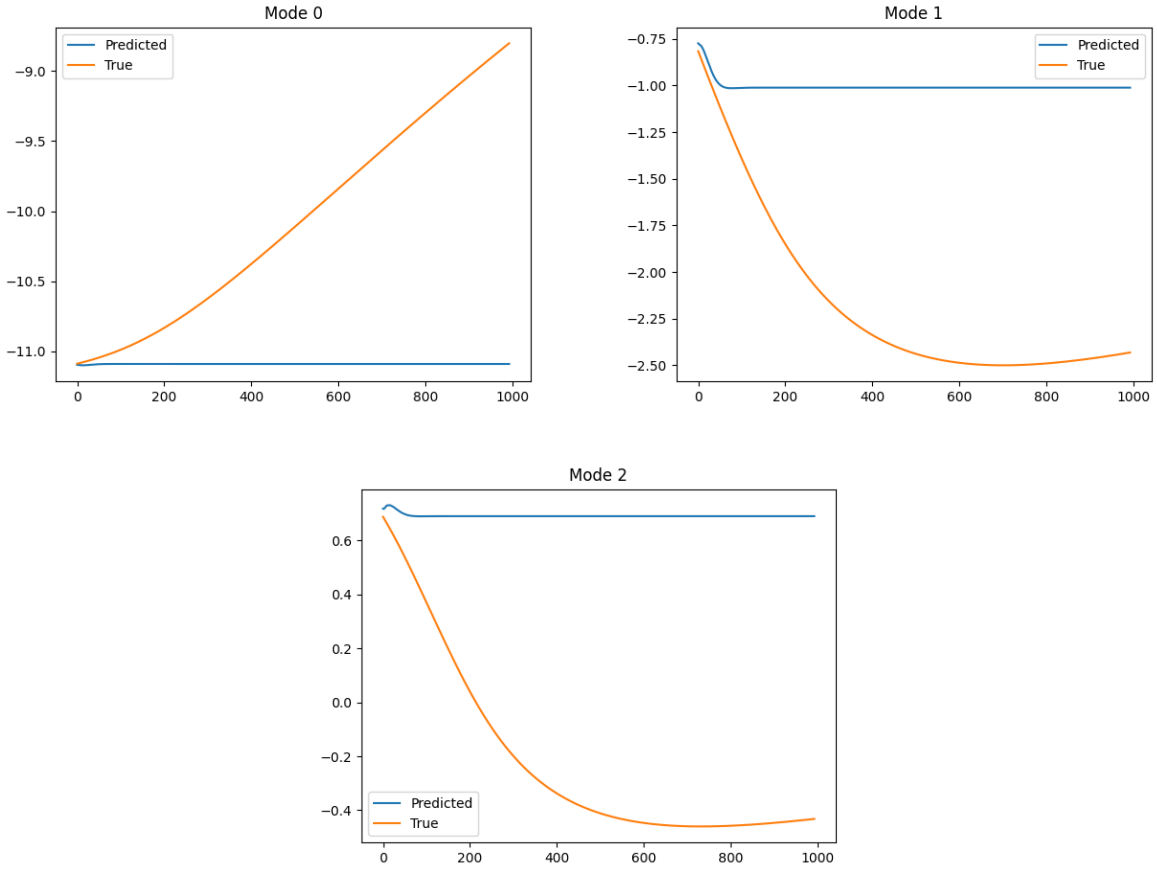

Figure 3: PyTorch LSTM mode prediction plots (three compressed modes).

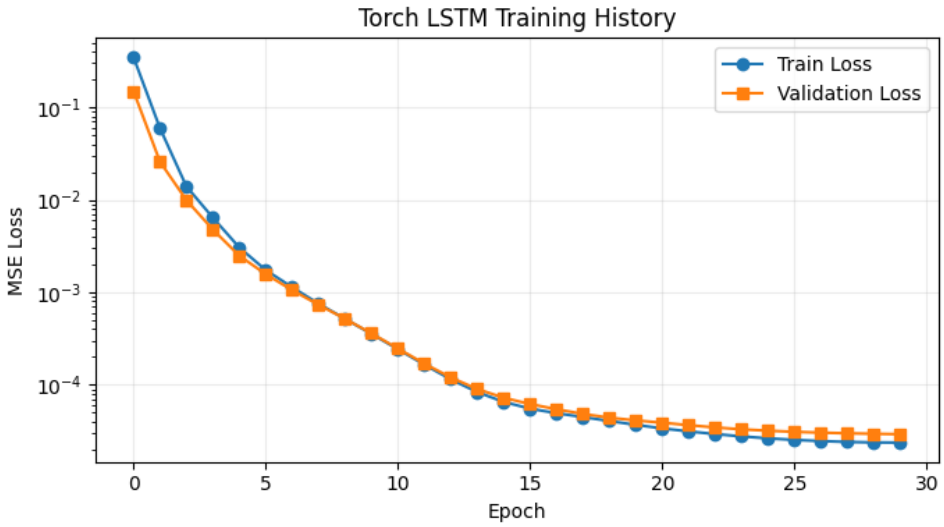

Figure 4: Training and validation loss reduction over epochs (log-scale MSE).

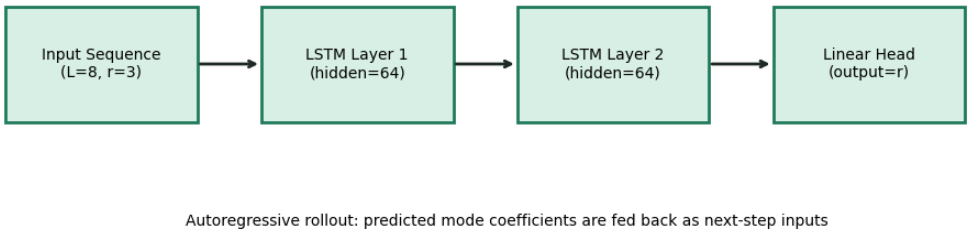

Figure 5: Schematic of the deployed Torch stacked-LSTM forecasting architecture.

8 Modernized Project State (Before vs After)

Area	Before	After
Build config	Hardcoded Python 3.6-specific absolute paths	Portable Python/NumPy discovery via CMake target-based linking
Runtime safety	Minimal Python object validation	Explicit import/call checks, traceback printing, null-safe handling
ML stack	TensorFlow/Keras-centric implementation	PyTorch stacked-LSTM with explicit autograd training and `.pt` checkpointing
Execution artifacts	Legacy checked-in root PNGs	Freshly generated outputs in `build/` + checkpoint/log artifacts

9 Recommended Next Steps

1. Stabilize numerics in `python_module.py`: guard/clip `time_series` values before LSTM training and add NaN/Inf checks.
2. Resolve return-contract mismatch: ensure `analyses_func()` always returns the shape expected by `app.cpp` or update C++ expectations.
3. Move Python modules out of `build/` into a source package directory and import explicitly to separate source from generated artifacts.
4. Add reproducible setup files (e.g., `requirements.txt` and/or pinned environment file) to lock dependency versions.
5. Add a lightweight CI workflow to build and run a short smoke test that validates Python module import and one analysis pass.
6. Optionally refactor ML training hyperparameters and logging for deterministic benchmarking across runs.

10 Reference Paths

- Executable: `build_local/app`

- Logs: `build/run.log`, `build/run_unbuffered.log`
- Checkpoint: `build/checkpoints/my_checkpoint.pt`
- Figures: `build/*.png`