Project Report

a. Motivation

I am working on the research project to solve inverse problem using plug and play algorithms with diffusion model prior. Many plug-and-play diffusion (PnPDP) methods are developed and released primarily in the context of high-dimensional image reconstruction, with implementations tightly coupled to image-specific architectures and GPU-intensive computation. This creates a practical barrier for researchers who do not work directly with image data, lack large-scale GPU resources, or wish to study the theoretical behavior of these methods in a more controlled setting. In particular, statisticians and researchers working on structured or tabular inverse problems—such as missing-data imputation or linear inverse problems with known priors—often find it difficult to experiment with or even meaningfully interpret existing PnPDP frameworks.

If these challenges can be addressed, this line of work has the potential to serve as a bridge between the statistics and AI communities. Our framework lowers the barrier for statisticians to adopt diffusion-based methods in settings with known or partially specified priors, such as missing-data imputation, inverse problems in scientific computing, or structured latent-variable models. At the same time, it exposes fundamental differences between natural-image priors, medical-image priors, and tabular or numerical data viewed as pseudo-images, highlighting when and why assumptions embedded in modern diffusion models may fail. Ultimately, this work aims to make diffusion-based inverse solvers easier to plug in, analyze, and trust for uncertainty quantification—particularly in applications beyond imaging, where calibrated uncertainty is often more critical than peak reconstruction accuracy.

b. Project Description

To address this gap, I designed a modular interface that maps low dimensional vectors to image-like tensors and back, allowing existing diffusion-based solvers to be plugged in without modification to their core implementations. This design directly reflects several concepts emphasized in the statistical computing course.

First, the framework follows a *modular programming* philosophy: the forward operator, prior, sampler, and evaluation metrics are implemented as separable components with clean interfaces. This modularity made it possible to swap algorithms, forward models, and hyperparameters systematically, which is essential for reproducible benchmarking.

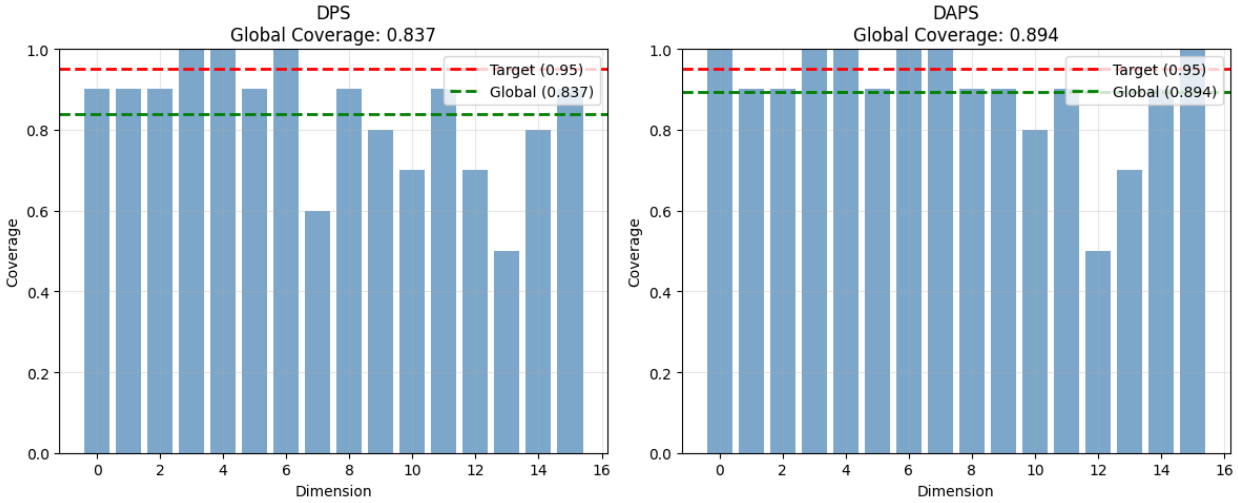
Second, I applied the principle of *reducing problem dimensionality for rapid experimentation*. By constructing low-dimensional, analytically tractable testbeds (e.g., mixture-of-Gaussians priors with known posteriors), I was able to stress-test complex sampling algorithms while keeping computation fast and interpretable. This mirrors the course emphasis on building minimal examples before scaling to large systems.

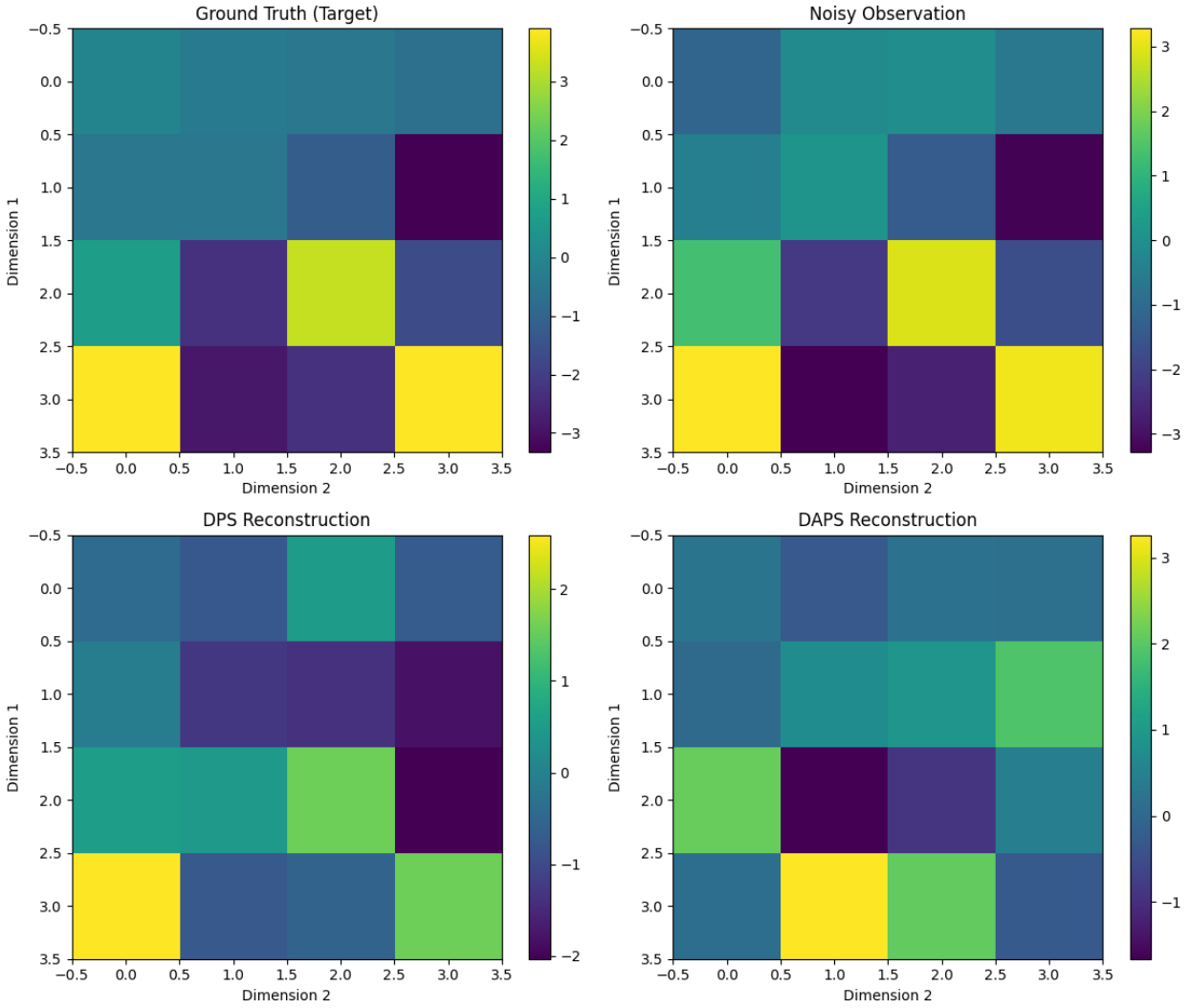
Third, the project relies heavily on *simulation-based validation and diagnostic testing*. I implemented repeated Monte Carlo runs, unit tests for variance and coverage, and step-by-step diagnostic checks (e.g., null-space variance tracking) to verify correctness and isolate failure modes. These testing strategies reflect the course focus on using simulation as a primary tool for understanding stochastic algorithms.

Overall, I used the modular design, efficient computation, and systematic testing learned from this course.

c. Results Demonstration

I evaluated the uncertainty quantification capabilities of several diffusion-based inverse algorithms (DPS, DAPS) on a toy 16-dimensional inverse problem with A=I (identity matrix). For each algorithm, I generated N=200 test samples from the known mixture-of-Gaussians prior, ran K=100 posterior samples per test case, and computed 95% credible intervals. The coverage analysis (left figure) shows the per-dimension coverage rates, measuring how often the true values fall within the estimated credible intervals. Ideally, these should be close to 95% for well-calibrated uncertainty estimates. The reconstruction results (right figure) demonstrate the point estimates from DPS and DAPS algorithms, showing their ability to recover the true 8-dimensional signal from noisy observations.





d. Lessons Learned

This project involved both methodological investigation and substantial implementation, which led to several important lessons.

1. Modular design is critical for research code. Early in the project, rapid prototyping and experimentation led to tightly coupled code, which made debugging and interpretation difficult. As the project evolved, I refactored the implementation into clearly separated modules (forward operators, diffusion priors, samplers, uncertainty metrics, and evaluation pipelines). This modular structure made it possible to systematically isolate components, swap methods, and design targeted diagnostic tests. In hindsight, investing in a clean top level project design significantly improved both development efficiency and scientific clarity.

2. AI-assisted coding is powerful, but architectural decisions must be human-driven. Modern AI coding tools were extremely effective at filling in implementation details, writing boilerplate, and translating mathematical ideas into code. However, "vibe coding" without a clear conceptual plan often resulted in code that ran but did not reflect the intended algorithmic behavior. The most effective workflow I think now was to first explicitly define what each

module should do and how it should interact with others, and then use AI assistance to implement each component. This separation of conceptual design and code generation proved far more efficient and robust.

3. Running code successfully is not the same as being correct.

One of the main challenges was that the code often executed successfully but produced visually impossible reconstructions. When results did not match expectations, the solution was rarely a single bug fix. Instead, we designed a sequence of increasingly simplified tests to localize the source of error. This process—reducing the problem to analytically tractable cases and checking each assumption—was essential for resolving issues such as variance collapse or explosion. The experience reinforced that debugging research code is fundamentally an experimental and scientific process, not just a software engineering task.