Final Project, STAT 243

Aparimit Kasliwal, Treves Li, Yuyang Wu

GitHub repo for this project https://github.berkeley.edu/ap-kasliwal/ars-dev.

1. Functional Programming Implementation of `ARS`

Our implementation of Adaptive Rejection Sampling, based on the `Gilkis et.al.` is based on the following modular functions;

(a) `construct_envelope` function: This function is used to construct an upper bound to the function which we want to sample from. A check is included to ensure that there are at least 3 points, which are required to create an envelope. Note that this function is called only once when we initiate an envelope for the first time, following which we only update the envelope from that point on.

(b) `update_envelope` function: Once an envelope is already created, this function performs the task of adding an additional point into the existing envelope by figuring out where it would lie, and then recalculating the slopes and the intercepts after that.

(c) `construct_squeezing` function: This function creates a lower bound to the given function which we want to sample from - similar to the `construct_envelope` function, but bounding it from the other side.

(d) `update_squeezing` function: This function updates the squeezing function in a similar fashion that the envelope is updated, by calculating the slopes and intercepts that get changed and thereby incorporating the new point.

(e) `calculate_piecewise_linear` function: Once there is a linear envelope, the sampling process requires us to calculate what the functional value of either the envelope, or the squeezing function would be at a given point on the x-axis. To do so, we implement this function called the `calculate_piecewise_linear` which basically figures out the linear part of the envelope / squeezing function corresponding to a given point, and calculates the functional value using the slope and the intercept of that linear part.

(f) `sample_piecewise_linear` function: Through this function, we are able to sample from the exponential transformation of the given linear part, the requirement of which majorly stems from working with exponential values instead of pure numerical values in order to avoid any numerical underflow.

(g) `adaptive_search_domain` function: This function figures out the ideal sampling domain by starting from a given point, and taking steps of a specified length to come up with the sampling domain.

(h) `init_points` function: This function calculates the intial points to be used during the sampling process which are offset by a parameter called the `threshold` which is to ensure avoiding any numerical issues when dealing with the computations for the given function.

(i) `check_overflow_underflow` function: Here, we specifcally check to see if there is any potential cause for an overflow or underflow to occur by comparing the numerical values that arise in the computation with the machine epsilon on one end and relative upper bounds for avoiding overflow on the other.

(j) `h_log` function: This function is used to implement caching for the log of the relevant function, and also implements underflow protection to prevent numerical issues during the sampling process.

(k) `h_cached` function: This function caches h values to avoid recomputing for the same x values, and thus helps reduce computational costs in case some calculations tend to be repeated.

(l) `is_log_concave` function: This function is specifically built to check if a given function is log-concave in some particular domain. If not, this throws errors that help the user debug at any relevant point during the sampling process.

(m) `compare_samples_to_distribution` function: As a part of this function, we implement a method for comparing the distribution of the generated samples with the target function. Note that this is not done for all the cases that the user might implement, but only for a few test cases on the backend. This function uses the KS Test, which relies on a closed form CDF, for calculating the information loss when approximating the given target distribution with a different approxmiate distribution, which in this case is the distribution of the generated samples.

(n) `ars` function: This is the key sampling function, whihc pretty much calls the other function within itself to perform modular tasks - which made for the whole sampling process pretty smooth to debug in case of any intermediate errors. Samples that are generated are either accpeted or rejected - where the former are appended to a list that is returned to the user, while the latter are used to consecutively update the envelope so theat the envelope forms a tighter approximation to the target function. We tried implement batching, which generated multiple samples in one single iteration, but found ourselves out of time to implement the changes required in other functions to incorporate the presence of multiple accepted and multiple rejected samples in one single iteration of the algorithm. Overall, we discovered that each fo the samples accepted according to the criterion was correct, and the rejection rate of the sampling algorithm went down as the number of iterations of sampling iterations increased, since the envelope became a tighter and a better approximation for the target function.

2. File Structure and Modules

We propose the following file structure, into which we collapse all the functions defined above. As visualized below, our main functional code is present in the **ars** directory within the **gradients.py**, **sampler.py**, **utils.py** and **validation.py** files. All the files are available in this GitHub repo.

On the other hand, we have a directory called **tests** which contains different files focusing on various aspects of testing our code - all of which can be executed using **pytest** with the following command:

```
pytest tests/.
```

This project is organized as follows; some irrelevant subdirectories or files are omitted for organizational clarity:

```
  README.md                       # Project overview and instructions
  ars                             # Main ARS package
      __init__.py                # Package initializer
      sampler.py                 # ARS sampling implementation
      sampler_jax.py             #TODO EXPLAIN <-------------------------
      utils.py                   # General utility functions
      validation.py              # Validation functions for ARS
  ars.ipynb                       #TODO EXPLAIN (can also omit) <--------------------------
  ars_new.ipynb                   #TODO EXPLAIN (can also omit) <--------------------------
  debug_and_compare.ipynb         # Jupyter notebook for debugging and comparison
  final_project.pdf               # Final PDF deliverable
  final_project.qmd               # Quarto document for the PDF deliverable
  pytest.ini                      # Configure pytest settings
  requirements.txt                # Python dependencies for the project
  setup.py                        # Distribution installation metadata & instructions
  tests                           # Unit tests for the project
      test_sampler.py             # Tests for sampler module
      test_utils.py               # Tests for utils module
      test_validation.py          # Tests for validation module
```

3. Installation of the package and code exceution

We've included specific files called `requirements.txt` and `setup.py` which allow transforming our developed code into an installable package through the following command:

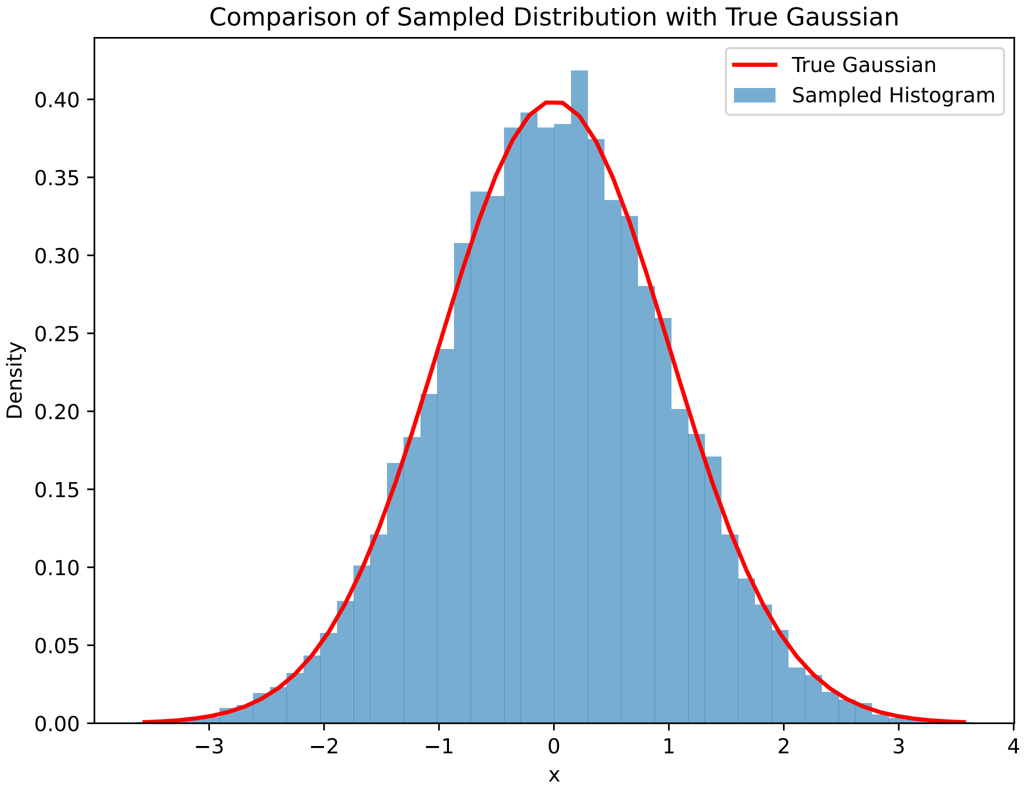
```
pip install .
```

After installation, the package can be run simply with a code similar to the example provided below:

```
import numpy as np
import ars
import matplotlib.pyplot as plt

def gaussian(x):
    return np.exp(-0.5 * x**2)

samples = ars.ars(gaussian, num_samples=10000, domain=(-5, 5), num_init_points=10)
```

```
Starting ARS ...
Searching for the domain ...
Finished sampling. Total samples collected: 10000
```

```
hist, bin_edges = np.histogram(samples, bins=50, density=True)
bin_centers = 0.5 * (bin_edges[:-1] + bin_edges[1:])
true_density = np.exp(-0.5 * bin_centers**2) / np.sqrt(2 * np.pi)

# Plotting
plt.figure(figsize=(8, 6), dpi=300)
plt.plot(bin_centers, true_density, label="True Gaussian", color='red', linewidth=2)
plt.bar(bin_centers, hist, width=bin_edges[1] - bin_edges[0], label="Sampled Histogram", alph
plt.xlabel('x')
plt.ylabel('Density')
plt.legend()
plt.title('Comparison of Sampled Distribution with True Gaussian')
plt.show()
```



4. Testing

For testing of the code, we took an iterative approach where some members of the team would work on particular core functions, while others would develop unit tests using `pytest`. The tests can be found in the `tests` subdirectory, with Python scripts written to correspond roughly to one of `sampler.py`, `validation.py`, and `utils.py`.

In `test_sampler.py`, we began by testing the user input checks on our `ars` function (e.g., ensuring the correct data types were passed, with the correct number of elements, etc.). We made sure that any error messages returned were informative, both for the user's sake and for our own debugging purposes. The second set of unit tests focusses on testing basic functionality of the sampling script and checking that the correct number of samples were returned for a given domain boundary. When we were considering implementing burn-in, we added a check to ensure that our code handled pre-sampling without returning an incongruous output length. The `test_adaptive_domain_search` function checks the behavior of our code over infinite domains, and again ensures that the desired sample size is returned.

The test_envelope_and_sampling function tests two components simultaneously: construct_envelope and sample_piecewise_linear. The construct_envelope function builds the upper hull envelope of the target function, and validates that the envelope output has the correct structure and that the number of z_points is consistent with the number of pieces. Additional checks confirm that the sampled points were scalars and lie within the defined domain. The test_stability function evaluates our code across a range of small and large variances, since we were concerned how our code would perform numerically on these edge cases. Finally, we tested for the distribution accuracy by comparing the sampled distributions to the true target densities, and ensuring the final result falls within a given tolerance.

In test_validation.py, we wrote the test_is_log_concave function to ensure we were correctly identifying log-concave target functions. Part of this testing included verifying edge cases such as valid log-concave functions (e.g., Gaussian and exponential), non-log-concave functions (e.g., quadratic), and improper inputs like negative values or constant functions. Additionally, the KS-related tests evaluated the compare_samples_to_distribution function in validation.py by mocking outputs and verifying that the printed warnings and/or results were appropriate.

The last series of unit tests were included in test_utils.py. Included in this quite were tests to ensure that our h_log function handled invalid/edge inputs such as None, zero values (invalid for logarithmic operations), and non-callable objects. We also tested its performance with very large inputs, to see if it would run to completion without overflow or other numerical errors.

5. Statement of Contribution

- **Aparimit Kasliwal**: Initial Functional Structure & Ideation; AD Implementation using `jax`; Addressing numerical issues during computation
- **Treves Li**: Testing; installation and package setup; Finalization of the Funtional Structure
- **Yuyang Wu**: Initial Functional Structure; Algorithm Development