# Code Documentation for "Synthesizing Precise Static Analyzers for Automatic Differentiation"

Jacob Laurel, Siyuan Brant Qian, Gagandeep Singh, Sasa Misailovic

July 2023

## Introduction

We now describe the source code accompanying our artifact for the paper "Synthesizing Precise Static Analyzers for Automatic Differentiation"

## Forward Mode (non-tensorized) AD Source Code

As Pasado supports both forward mode and reverse mode AD, we provide implementations for both. For forward mode AD, the first part of the source code can be found in the `forward_mode_non_tensorized_src/` directory. This code is non-tensorized because unlike neural networks, general computational graphs do not necessarily always obey a tensorized structure that would allow for vectorizing each operation. Thus, this is why our source implementation has both tensorized and non-tensorized versions. These source files are used for the experiments in **Section 5.2** of the paper, as well as in the **Example** section of the paper.

1. `dual_intervals.py`: This file is the main abstraction for obtaining interval bounds on dual numbers. Note that this file supports not just the interval domain, but also zonotopes. The abstract interpretation of forward mode AD for every language primitive (meaning primitive functions like exp and primitive arithmetic operations like +) is defined here. Specifically, the main class for (abstract) forward mode AD is `Dual` where each operation (`__add__`, `__mul__`, etc.)is overloaded for the abstract domain (e.g. intervals or zonotopes). The naming convention is such that functions like `SigmoidDual` use the standard interval or zonotope abstract transformers whereas the functions with the `-Precise` suffix, such as `SigmoidDualPrecise` use Pasado's precise, synthesized abstract transformers.

2. `quotient_rule.py`: This file contains the code for the Quotient Rule synthesized abstract transformer as well as for the Product Rule abstract transformer. These abstract transformers are described in the paper in **Section 4.2** and **Section 4.3**. The synthesized Quotient Rule abstract

transformer is itself implemented in `SynthesizedQuotientRuleTransformer` for the regular zonotope domain and `SynthesizedQuotientRuleTransformer_mixed` for the reduced product of the Zonotope domain with the interval domain. Likewise the synthesized Product Rule abstract transformer for the zonotope domain is found in the `SynthesizedProductRuleTransformer` function and similarly the synthesized abstract transformer for the reduced product of Zonotopes with intervals is found in the function named `SynthesizedProductRuleTransformer_mixed`.

3. `CubicEquationSolver.py`: This file comes from [Kum17] and contains the functionality to solve for roots of a cubic equation - this functionality is needed for the abstract transformer for the quotient rule (**Section 4.3**, **Optimization Problem** in the paper).

4. `invert_gaussian_deriv.py`: For the chain rule abstract transformer we require a way to find roots for general nonlinear functions, not just cubics, (**Section 3.3** and **Section 4.1, Optimization Problem** in the paper), hence this code corresponds to the Newton-Raphson root solver, particularly the `my_newton` function. In particular, we use this script for the chain rule abstract transformer for the NormalCDF function (used in the Black-Scholes benchmarks). It is worth noting that in the general case, using the Newton-Raphson method is not necessarily sound (since it is not guaranteed to find *all* roots), however for the functions we evaluate, the combination of the Newton-Raphson procedure along with our case-by-case analysis of the NormalCDF function's second derivatives (in the `invert_erf_2nd_deriv` function), does ensure that all potential roots (representing critical points) are found.

5. `MixedAAIA.py`: This code mixes together affine arithmetic (AA), also called zonotopes, with interval arithmetic (IA). This combination is equivalent to the reduced product of the zonotope abstract domain with the interval abstract domain. Thus the `MixedAffine` class defines an abstract element from the reduced product domain. The `get_bounds` function implements the bounding box function of the reduced product of zonotopes with intervals defined **Section 3.3** item 3 of the paper. The `__mul__`, `__truediv__`, `__sub__`, etc. function overloadings define standard abstract transformers for each of those respective arithmetic operations for the reduced product abstract domain, as required in **Section 3.3**, item 5. The refinement operation of the reduced product whereby the results of one of the abstract domains is used to refine the other is encoded in the `interval_intersection` function.

6. `runge_kutta.py`: This code implements the Runge-Kutta ODE solver algorithm, as well as the Euler ODE solver, the latter solver being described in the **Example** section with the notation $\texttt{ODESolve}_f$. All the operations of these ODE solvers are overloaded to use the abstract interpretation, instead of the typical concrete interpetation so that one can abstractly

perform the AD sensitivity analysis on these ODE solvers' outputs. Specifically, the Runge-Kutta solver is implemented in the `runge_kutta` function and the Euler ODE solver is likewise encoded in the `euler` function.

7. `synthesized_transformer.py`: This code implements the Pasado synthesized chain rule abstract transformers for the nonlinear, univariate functions (e.g. exp, tanh) the zonotope abstract domain. Because these are (synthesized) abstract transformers for the chain rule patter, which is $g(x, y) = f'(x) \cdot y$ for the given differentiable function $f$ (e.g. exp, tanh), the naming convention is always the name of the function $f$, followed by "`PrimeProductTransformer`". Thus for example the function `SynthesizedFourthPrimeProductTransformer` corresponds to the synthesized chain rule abstract transformer when $f(x) = x^4$ (the fourth power function). Likewise the functions `SynthesizedSqrtPrimeProductTransformer` and `SynthesizedLogPrimeProductTransformer` correspond to the synthesized chain rule abstract transformers when $f(x) = \sqrt{x}$ and $f(x) = \log(x)$, respectively. Note that because this file implements the synthesized abstraction for the zonotope domain, the underlying data type used by these transformers will be the `Affine` arithmetic datatype from the `affapy` library [Hel21]. These abstract transformers are discussed in depth in **Section 4.1** of the paper.

8. `synthesized_transformer_mixed.py`. This file is virtually identical to `synthesized_transformer.py`, with the caveat that all the synthesized chain rule abstract transformers are defined for the reduced product of zonotopes with intervals. Hence the underlying data type used by these abstract transformers will be the `MixedAffine` class from the `MixedAAIA.py` file. Again, these abstract transformers are discussed in depth in **Section 4.1** of the paper.

9. `tests/`: This sub-directory just contains simple test cases that we used as we developed our tool. In particular, `test.py` tests the implementation of each of the abstract transformers, `runge_kutta.py` tests the implementation of the runge-kutta solver on various test-case ODE models, `mixed_test.py` tests the implementation on the reduced product of the zonotopes and intervals. These small test scripts served simply to sanity check our development over the duration of the project. Anyone wishing to expand upon our work should add their new test cases to this folder.

# Reverse Mode (non-tensorized) AD Source Code

As mentioned previously, Pasado supports both forward mode and reverse mode AD, hence we provide implementations for both. For reverse mode AD, the first part of the source code can be found in the `reverse_mode_non_tensorized_src/` directory. As in the previous section, this code is non-tensorized because unlike neural networks, general computational graphs do not necessarily always

obey a tensorized structure that would allow for vectorizing each operation. In particular, the Black-Scholes benchmark we evaluate does not have the tensor structure. This reverse-mode AD code is used for the experimental evaluation in **Section 5.3** of the paper. As a first step, we overload the `micrograd` library [Kar20] in order to be able to perform reverse mode AD with abstract domains instead of concrete points.

1. `interval_rev.py`: This code contains the interval arithmetic abstract interpretation of reverse-mode AD. We implement this solely as a baseline, against which we compare Pasado. The interval abstraction for each function is given with the following naming convention of "`i_`" followed by the function name. For example `i_exp` corresponds to the interval abstraction of the exp function.

2. `zonotope_rev.py`: This code contains the standard zonotope arithmetic abstract interpretation of reverse-mode AD. We implement this in part as a baseline, against which we compare Pasado. The naming convention for the zonotope baseline we compare against is given as "`z_`" followed by the function name. For example the `z_exp` function corresponds to the standard zonotope abstract transformers for the exp function. In addition to the standard abstract transformers, we also applied Pasado's synthesized abstractions to the zonotope domain (without reduced products). These versions of the abstract transformers are given by the naming convention "`precise_z`" followed by the function name. For example the `precise_z_div` function corresponds to Pasado's *reverse-mode* synthesized abstract transformer for the quotient rule for the zonotope domain. Again, note that because this file implements the synthesized abstraction for the zonotope domain, the underlying data type used by these transformers will be the `Affine` arithmetic datatype from the `affapy` library [Hel21].

3. `mixed_zono_rev.py`: This code contains the reverse mode implementation of Pasado for the abstract domain corresponding to the reduced product of zonotopes with intervals. The reverse mode implementation of Pasado for just the regular zonotope domain (without the reduced product) is in the previous file `zonotope_rev.py`. The name of the file reflects the fact that here Pasado uses a reduced product of the zonotope domain with the interval domain, thus it can be seen as mixing the two abstract domains together. Hence the underlying data type used by these abstract transformers will be the `MixedAffine` class from the `MixedAAIA.py` file.

4. `tests/` Similarly, this sub-directory just contains small, simple tests cases that we used as sanity checks during our development. The `interval_rev_test.py`, `zono_rev_test.py`, and `mixed_zono_rev_test.py` respectively test the `interval_rev.py`, `zonotope_rev.py` and `mixed_zono_rev.py` files. Again, anyone wishing to expand upon our work should add their new test cases to this folder.

## Forward Mode Tensorized AD Source Code

This last part of the source code implements forward mode AD, but has been specialized for the tensor structure of DNNs. Because DNNs have a fixed structure, abstract domains like zonotopes can leverage this, and store the abstract element (e.g. the zonotope or interval) in a tensor, and perform the propagation of the abstract element through the computation using only tensor operations. This tensorized code is used for the experimental evaluation in **Section 5.4** of the paper. This code may be found in the `forward_mode_tensorized_src/` directory. To implement the tensor operations we use PyTorch [Pas+19].

1. `Duals.py`: This code is taken from [Lau+22b; Lau+22a]. This code corresponds to the dual interval abstract domain and serves as a baseline against which we compare Pasado.

2. `SimpleZono.py`: This code is taken from [Lau+22b] and implements the Zonotope abstract domain for tensors. We note that this code is also used to compare against as a baseline method.

3. `precise_transformer.py`: This code implements a *tensorized* version of our synthesized chain rule abstract transformer. This means that all the operations, such as the linear regression (for finding coefficients) and root solving (for finding critical points) are done with PyTorch tensor-level operations. This is only possible because DNNs satisfy this tensor structure. However this technique allows us to scale up our analysis to large benchmarks such as the neural networks we provide. Of all the files in tis directory, this one represents the core contribution of Pasado - the other files are just necessary for allowing us to compare against baselines. The theory behind this code is discussed in depth in **Section 4.1** of the paper.

4. `HyperDuals.py`: This code is taken from [Lau+22b] and is used solely for the purpose of providing functionality that the baseline (against which we compare Pasado) requires.

5. `HyperDualZono.py`: This code is taken from [Lau+22b] and is used solely for the purpose of providing functionality that the baseline (against which we compare Pasado) requires.

6. `conv.py`: This code is used to transform a 2D convolutional layer (a `torch.nn.Conv2d` object) into a mathematically equivalent affine layer (an `nn.Linear` object).

## Section 5.2 Experiments Source Code

We now describe the source code of the experimental driver scripts for the **Section 5.2** experiments. These scripts may be found in the `Section_5_2` directory. This explanation is designed for those who might wish to expand upon our work and run new experiments.

1. `climate_ode_script.py`: This script is the main experimental driver, and thus is a wrapper around the `climate_ode_v2.py` script, which is called as a subroutine. The different parameters we evaluate are stored in `y0_list`, `R_list`, `Q_list`, `alpha_list`, `sigma_list`, `iter_list` and `h_list`. The body of the main function loops over every combination of these parameter intervals to run an experiment for each configuration.

2. `climate_ode_v2.py`: This script contains the main climate model. Inside this script we have implemented the climate model for interval, zonotope and Pasado's abstract forward-mode AD.

3. `climate_plotter.py`: This script plots the results for the climate model that are saved in the `climate_step.jpg` file (which is Fig. 3 of the paper).

4. `climate_ratio_scatter.py`: This script produces the scatter plots for the climate model shown in Fig. 4b of the paper.

5. `climate.sh`: This script calls the experimental driver `climate_ode_script.py` as well as both of the plotting scripts (`climate_plotter.py` and `climate_ratio_scatter.py`) in order to provide a single push-button script to run everything relevant to the climate benchmark.

6. `chemical_ode_script.py`: This script is the main experimental driver for the chemical ODE experiments. Similarly to the `climate_ode_script.py`, the `chemical_ode_script.py` sweeps through all the different parameter configurations.

7. `chemical_example.py`: This code contains the actual chemical Neural ODE benchmark that gets called by the `chemical_ode_script.py` driver. Thus `chemical_example.py` contains the code for running the abstract AD sensitivity analysis for the interval domain, zonotope domain, and with Pasado's synthesized abstract transformers.

8. `chemical_plotter.py`: This script plots the results that are shown in Fig. 5 of the paper.

9. `chemical_ratio_scatter.py`: This script plots the results that are shown in Fig. 4a of the paper.

10. `chemical.sh`: This shell script calls the experimental driver `chemical_ode_script.py`, as well as the plotting scripts (both `chemical_plotter.py` and `chemical_ratio_scatter.py`) in order to provide a single push-button script to run everything relevant to the chemistry Neural ODE benchmark.

11. `clear.sh`: This shell script simply clears the outputs obtained by running the experimental scripts. This shell script is useful if one wished to rerun the experiments, perhaps after building upon our code.

## Section 5.3 Experiments Source Code

We now describe the source code of the experimental driver scripts for the **Section 5.3** experiments. These may be found in the `Section_5_3` directory. This explanation is designed for those who might wish to expand upon our work and run new experiments.

1. `black_scholes_rev_script.py`: This script corresponds to the main experimental driver for running the abstract reverse mode AD on the Black-Scholes benchmark. This script sweeps through different paramter configurations to evaluate how well Pasado performs (compared to interval and zonotope abstract AD) for various input ranges. The `black_scholes_rev_script.py` script is essentially a wrapper function that calls the `black_scholes_rev.py` script.

2. `black_scholes_rev.py`: This script contains the actual encoding of the Black-Scholes benchmark for Pasado, as well as for the interval and zonotope domain comparisons.

3. `black_scholes_ratio_scatter.py`: This script generates the plots and figures for the experimental evaluation of the Black-Scholes model. Specifically, this script generates Fig. 6 of the paper.

4. `black_scholes.sh`: This shell script calls the `black_scholes_rev_script.py` experimental driver, as well as the `black_scholes_ratio_scatter.py` plotting scripts in order to provide a single push-button script to run everything relevant to the Black-Scholes benchmark.

5. `clear.sh`: This shell script simply clears the outputs obtained by running the experimental scripts. This shell script is useful if one wished to rerun the experiments, perhaps after building upon our code.

## Section 5.4 Experiments Source Code

We now describe the source code of the experimental driver scripts for the **Section 5.4** experiments. These may be found in the `Section_5_4` directory.

1. `get_lipschitz.py`: This is the main experimental driver script for the FFNN local Lipschitz constant experiments. In particular, the `forward_interval(x)` function corresponds to the dual interval arithmetic baseline of [Lau+22a] applied to the FFNN, the `forward_zono(x)` function corresponds to the zonotope abstract AD baseline of [Lau+22b] applied to the FFNN and the `forward_zono_precise(x)` function corresponds to Pasado applied to the FFNN. If someone wishing to expand upon our work wanted to define a new abstraction (e.g. a new abstract domain or abstract transformer for AD), then they would need to call that new abstraction inside this script in order to compare against our existing works.

2. `get_lipschitz_cnn.py`: This is the main experimental driver script for the CNN local Lipschitz constant experiments. If someone wishing to expand upon our work wanted to define a new abstraction (e.g. a new abstract domain or abstract transformer for AD), then they would need to call that new abstraction inside this script in order to compare against our existing works.

3. `Plot.ipynb`: This jupyter notebook allows for plotting of all the FFNN experimental results. Specifically, this includes Fig. 8 of the paper.

4. `Plot_cnn.ipynb`: This jupyter notebook allows for plotting of all the CNN experimental results. Specifically, this includes Fig. 9 of the paper.

5. `get_times.py`: This script outputs the runtimes for each method (Pasado as well as the Zonotope and Interval abstraction baselines) corresponding to [Lau+22b] and [Lau+22a] respectively.

6. `get_times_cnn.py`: This script outputs the runtimes for each method (Pasado as well as the Zonotope and Interval abstraction baselines) for the CNN benchmarks.

7. `model.py`: This file specifies the architecture for the neural networks on which we evaluate. While we provide the pre-trained networks, *this file is what would be changed* if someone wished to try our technique with new neural network benchmarks.

8. `lipschitz.sh`: This shell script calls the `get_lipschitz.py` experimental driver script on all of the FFNN architectures we provide, thus giving a unified, single push-button script to run all the FFNN Lipschitz robustness experiments.

9. `lipschitz_cnn.sh`: This shell script calls the `get_lipschitz_cnn.py` experimental driver script on all of the CNN architectures we provide, thus giving a unified, single push-button script to run all the CNN Lipschitz robustness experiments.

10. `train_mnist.py`: This script is taken from [Lau+22b] and provides the code needed to train a network from scratch. If one wished to build upon our code and try new neural network benchmarks, *this script would need to be modified* in order to train those new benchmarks.

11. `test_mnist.py`: This script computes the test accuracy of the trained networks. In our case, because we have pre-trained the networks this script is necessary, however for someone wishing to build upon our work by trying new neural network benchmarks this script would be necessary.

12. `train_cnn.py`: This script provides the code needed to train and test a CNN from scratch. If one wished to build upon our code and try new neural network benchmarks, *this script would need to be modified* in order to train those new benchmarks.

13. `cnn_helper.py`: This script provides the code needed to pass the abstractions, i.e., `DualIntervalTensor` and `DualZonotope` (for both zonotope AD and Pasado), through our CNN instances. Note that these functions are *specific* to our `Conv` class inside `model.py`.

## Section 5.5 Experiments Source Code

We now describe the source code of the experimental driver scripts for the **Section 5.5** experiments. These may be found in the `Section_5_5` directory. This explanation is designed for those who might wish to expand upon our work and run new experiments.

1. `adult_script.py`: This script corresponds to the main experimental driver for running the abstract forward-mode AD on the Adult Income benchmark. This script sweeps through different paramter configurations to evaluate how well Pasado performs (compared to interval and zonotope abstract AD) for various input ranges. The `adult_script.py` script is essentially a wrapper function that calls the helper functions in the `adult_eval.py` script.

2. `adult_eval.py`: This script contains the actual encoding of the Adult Income benchmark for Pasado, as well as for the interval and zonotope domain comparisons.

3. `adult_train.py`: This script trains an MLP based on the Adult Income dataset [BK96] and save the model weights as files.

4. `adult_plot.py`: This script generates the figure for the experimental evaluation of the Adult Income network. Specifically, this script generates Fig. 10 of the paper.

5. `adult.sh`: This shell script calls the `adult_script.py` experimental driver, as well as the `adult_plot.py` plotting scripts in order to provide a single push-button script to run everything relevant to the Adult Income benchmark.

6. `clear.sh`: This shell script simply clears the outputs obtained by running the experimental scripts. This shell script is useful if one wished to rerun the experiments, perhaps after building upon our code.

## References


[BK96] Barry Becker and Ronny Kohavi. *Adult*. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5XW20. 1996.

[Kum17] Shril Kumar et al. "CubicEquationSolver". In: (2017).

[Pas+19] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[Kar20] Andrej Karpathy et al. “micrograd library”. In: (2020).

[Hel21] Thibault Helaire et al. “affapy library”. In: (2021).

[Lau+22a] Jacob Laurel et al. “A dual number abstraction for static analysis of Clarke Jacobians”. In: *Proceedings of the ACM on Programming Languages* 6.POPL (2022), pp. 1–30.

[Lau+22b] Jacob Laurel et al. “A general construction for abstract interpretation of higher-order automatic differentiation”. In: *Proceedings of the ACM on Programming Languages* 6.OOPSLA2 (2022), pp. 1007–1035.