Interactive Visualization for Data Science Scripts

Rebecca Faust, Carlos Scheidegger, Katherine Isaacs, William Z. Bernstein, Michael Sharp, and Chris North

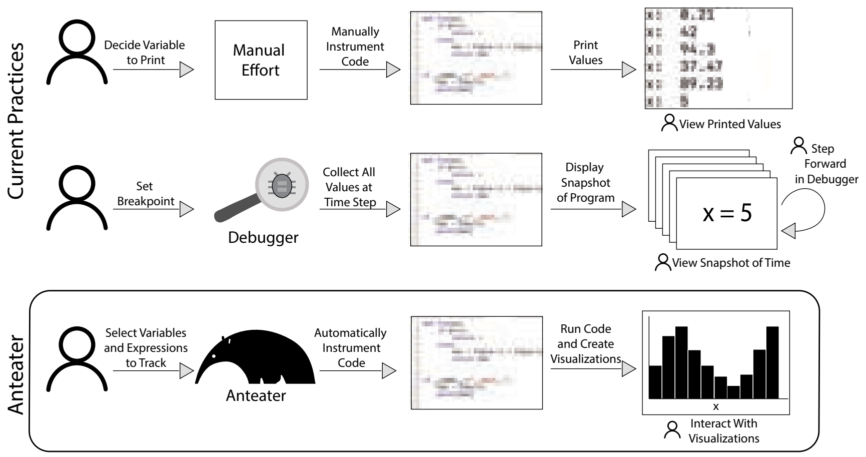


Fig. 1. An analyst inspects the behavior of a value in their script. One common practice (top row) is to instrument the script manually to collect variables of interest (here, x), and print their values. Manual instrumentation, however, is itself repetitive and error-prone. Another common practice (second row) is to use a debugger to stop the execution of the script and view each individual value assignment of x, providing a precise, but narrow, one-at-a-time view of the values. Anteater (bottom row) automatically instruments the script to trace the script and record variables along with execution structures. Its interactive visualizations provide global views of trace data, enabling easy observation of value behavior as well as interactions that narrow down the views to specific subsets of the execution.

Abstract— As the field of data science continues to grow, so does the need for adequate tools to understand and debug data science scripts. Current debugging practices fall short when applied to a data science setting, due to the exploratory and iterative nature of analysis scripts. Additionally, computational notebooks, the preferred scripting environment of many data scientists, present additional challenges to understanding and debugging workflows, including the non-linear execution of code snippets. This paper presents Anteater, a trace-based visual debugging method for data science scripts. Anteater automatically traces and visualizes execution data with minimal analyst input. The visualizations illustrate execution and value behaviors that aid in understanding the results of analysis scripts. To maximize the number of workflows supported, we present prototype implementations in both Python and Jupyter. Last, to demonstrate Anteater's support for analysis understanding tasks, we provide two usage scenarios on real world analysis scripts.

Index Terms—Interactive Visualization, Program Traces, Jupyter, Debugging

2022 IEEE Visualization in Data Science (VDS) | 978-1-6654-5721-7/22/$31.00 ©2022 IEEE | DOI: 10.1109/VDS57266.2022.00009

1 Introduction

From industry to academia, data science has become ubiquitous across a wide range of domains and is central in data-driven decision making. These decisions frequently have real world impact and, as such, it is imperative that people adequately understand the behavior of their analyses. Current practices for understanding analysis scripts are generally limited to standard program debugging practices, including print statements, logging files, and step-through debuggers. [9] However, these practices, while notoriously difficult in standard software engineering settings, become even more cumbersome in a data science setting.

- *Rebecca Faust and Chris North are with the Department of Computer Science, Virginia Tech. E-mail: {rfaust,north}@vt.edu.*
- *Carlos Scheidegger is with the HDC Lab, Department of Computer Science, University of Arizona. E-mail: cscheid@arizona.edu*
- *Katherine Isaacs is with the SCI Institute and School of Computing at the University of Utah. E-mail: kisaacs@sci.utah.edu*
- *William Z. Bernstein is with Air Force Research Laboratory. Email: william.bernstein@us.af.mil*
- *Michael Sharp is with NIST. E-mail: michael.sharp@nist.gov*

Data scientists exist in a wide range of fields and, although they exhibit sufficient programming skills, they often come from non-computing backgrounds where they did not receive formal training in programming practices [23]. As such, expecting all data scientists to be proficient enough in standard debugging methods to navigate and deeply understand their scripts is often unrealistic.

The rise of computational notebooks for data science further complicates the debugging practices for data scientists. Computational notebooks support exploratory programming behaviors that are common to data scientists [24], such as applying multiple analyses or parameter settings to a single dataset to achieve the best results. However, they also create several pain points when it comes to understanding and debugging analysis behavior, including out of order execution and workflow disruption [9]. Not only does out of order execution make it difficult to track the flow of the script, but it renders any standard debugging tools somewhat ineffective as they cannot track and illustrate the order of execution. Additionally, computational notebooks do not have built in debuggers. Some debugging extensions exists (e.g. Xeus for JupyterLab [15]) but are not globally supported or included by default. This causes additional barriers to understanding scripts for data scientists working in computational notebooks.

In 1993, after observing that debugging practices had not deviated substantially from iterative breakpoint debugging over the previous ˜˜ years, Steven Reiss proposed a the idea of trace-based debugging -

collecting execution information automatically in a program trace and allowing people to inspect the collected information [26]. Reiss identified the primary benefits of this approach as the ability to inspect any point of the past state (alleviating the need for multiple runs with varying breakpoints) and eliminating the problem of multiple runs yielding different behaviors (easing the challenges of reproducing erroneous behaviors).

While these benefits are of great importance in general debugging tasks, we see an additional opportunity to use trace data to support data scientists in understanding their exploratory programming tasks. Exploratory programming tasks come with subsequent exploratory debugging tasks, such as understanding or evaluating the "goodness" of analysis results. For example, an analyst may want to observe the evolution of the proposed solution in gradient descent to determine the algorithms stability or inspect the correlation of the proposed solutions in a multi-objective optimization to learn the priorities of the optimization. Traditional debugging methods fail to support these types of debugging tasks because the rely on the serial inspection of values. This requires analysts to build and maintain a mental image of value trends and behaviors, an extremely difficult task.

However, because traces record complete histories of values automatically, they present the opportunity to support broader overviews of value trends that alleviate the burden of building mental images. Traditional methods of serially inspecting values directly contrasts the fundamental principles of visualization. We have seen time and again the power of interactive visualization for presenting overviews of data in an easily consumable manner that is far less burdensome than serially inspecting individual values. In fact, the widely used "Visualization Information Seeking Mantra" emphasizes the need for overviews of data first, with controls to filter and inspect details as needed [32]. However, these visualization principles have yet to be applied in a debugging setting. Thus, the question remains: How can we visualize traces of data science scripts to reveal data behaviors?

To address this question, we developed a trace-based visual debugging method, Anteater, that we builds on the idea of trace-based debugging by combining it with well-principled visualizations. Fig. 1 contrasts our method with traditional practices. Rather than manually inserting logging statements or searching for a breakpoint, we automatically collect a trace of the program. Then, once collected, we present the trace through overview visualizations that illustrate the behavior of the program execution and values. From here, we provide people with controls to winnow down to specific areas or values of interest. To support multiple data science workflows, we present two prototype implementations of our method: one in Python and one in Jupyter notebooks. Fig. 2 presents an overview of the Juypter implementation of Anteater and examples of the trace visualizations.

In summary, the contributions of this paper include:

- A trace-based visual debugging method based on automatically gathering, organizing, and visualizing script data.
- Prototype implementations in both Python and Jupyter notebooks.
- Two usage scenarios using real world programs to demonstrate the benefits of our method on data science scripts.

2 Related Work

Debugging in Computational Notebooks Recent years have seen a massive increase in the usage of Jupyter for data science [24]. Jupyter appeals to analysts because of its support for exploratory, iterative analyses. However, despite the benefits, computational notebooks present several challenges that hinder data science workflows.

Notably amongst these challenges, is the non-linearity of execution flows [9, 30]. Jupyter allows analysts to run cells one at a time, more than once, and in any order. This makes tracing the execution flow extremely difficult to follow [9].

Additionally, non-linearity contributes to a larger problem of messiness in notebooks [18, 30]. Because analysts can rapidly add new analyses and transformations, notebooks can quickly become overrun with outdated, unused, or purely exploratory cells. However, ma

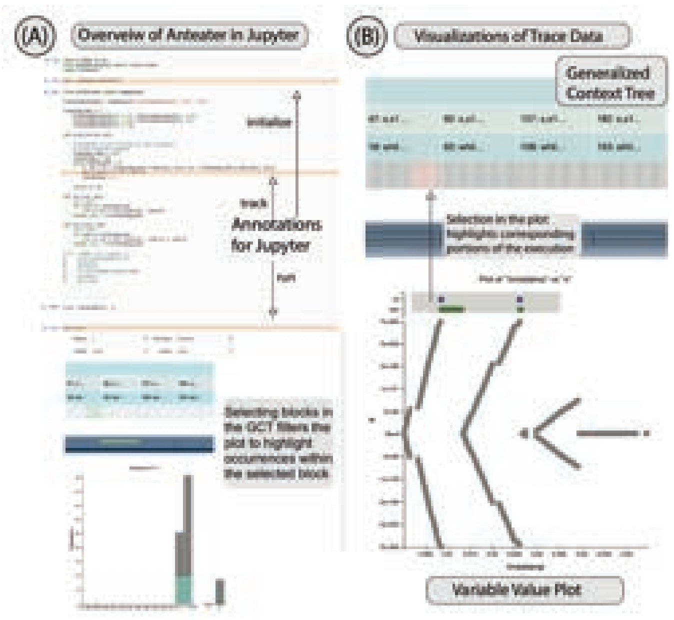

(A)
Overveiw of Anteater in Jupyter
initialize
track
Annotations for Jupyter
Selecting blocks in the GCT filters the plot to highlight occurences within the selected block
(B)
Visualizations of Trace Data
Generalized Context Tree
Selection in the plot highlights corresponding portions of the execution
Variable Value Plot

Fig. 2. An overview of Anteater in Jupyter. (A) shows an example notebook where an analyst ran the same gradient descent cells under four different training rates, highlighting the annotations needed to run Anteater. (b) shows a close up of the two types of trace visualization with an example interaction.

analysts are reluctant to fully remove old cells for fear of deleting something they may actually want later [30].

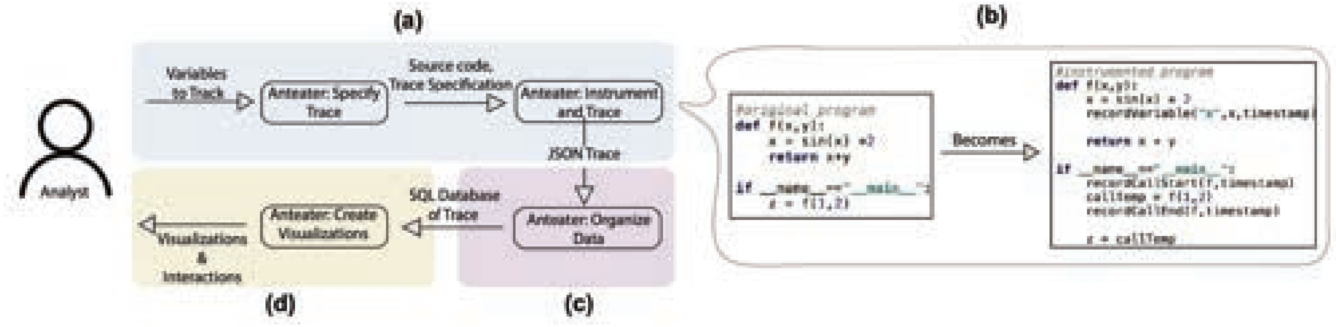
Chattonpadhyay et al. identify several additional pain points with computational notebooks [9]. While these challenges range from data loading to reproducibility, several center around the ability to get immediate feedback without disrupting their workflow. Specifically, they identified pain points with debugging due to the lack of adequate in notebook tools for inspecting variable values and script flow.

Several tools have been proposed to help address these challenges. Xeus is notebook extension in JupyterLab that provides a step-through debugging in Jupyter [15]. While it provides inspection of variable values in notebooks, it still suffers from the limitations of breakpoint debuggers on exploratory debugging tasks. Namaki et al.'s Vamsa provides automated data provenance in ML models that illustrates which data the model used to produce specific outputs [21]. Weinmen et al.'s Fork it [36] allows analysts to create multiple interpreter sessions to support side-by-side comparison of analyses workflows.

While each of these tools takes a step towards supporting analysts in notebooks, they do not quite address the problem of non-linear workflows and understanding the results of exploratory programming tasks. Our method is designed to add additional support for these problems. It allows for easy, in workflow inspection of variable values and the execution structure. Anteater accounts for the non-linearity of executed cells and can present the overall execution structure, even if it does not align with the linear flow of the notebook.

Visual Debugging Many attempts have been made to leverage visualization principles to augment the debugging process. Some efforts add visualization to breakpoint and step-through debuggers. Traditional visual debuggers typically provide visualization views of variables at a specific instance in time, much like traditional debuggers. Several of these tools add visualizations of objects to a breakpoint debugger [7, 10, 29]. Others provide visualizations to show task-specific information about the execution, such as an overview of the heap and stack [2, 20] the impact of resource utilization on control flow [22], object mutation [31], or run-time state and data structures of the program [34].

Generally, these tools present localized views that describe one particular state of the execution. Some tools provide additional context by allowing back-stepping in the debugger or providing a history of the execution [11, 19, 27]. In addition, some tools provide global views ow the behavior of values over the entire execution. Some tools


(a)
Variables to Track
Anteater: Specify Trace
Source code, Trace Specification
Anteater: Instrument and Trace
JSON Trace
Analyst
SQL Database of Trace
Anteater: Create Visualizations
Anteater: Organize Data
Visualizations & Interactions
(d)
(c)
(b)
Becomes

```
#original_program
def f(x,y):
    x = sin(x) +2
    return x+y

if __name__=="__main__":
    z = f(1,2)
```

```
#instrumented program
def f(x,y):
    x = sin(x) + 2
    recordVariable("x",x,timestamp)

    return x + y

if __name__=="__main__":
    recordCallStart(f,timestamp)
    callTemp = f(1,2)
    recordCallEnd(f,timestamp)

    z = callTemp
```

Fig. 3. An overview of the Anteater method. In (a), an analyst chooses variables and expressions to track, defining the trace specification. Then, Anteater passes the trace specification and the source code to the tracer. In the tracer, Anteater instruments the source code and traces the script. (b) shows a simplified version of this instrumentation. Next, in (c), it organizes the data in preparation for visualization. Last, in (d), Anteater automatically generates and presents visualizations of the trace.

give global views of value behaviors by introducing sparklines next to the line of source code defining the value [4, 14]. In contrast, Anteater displays global views that take the execution context into account. As we show in our evaluation, this perspective can be particularly helpful in debugging scenarios.

Hoffswell et al. [13] and Burg et al. [8] describe systems for visually debugging user interactions, one on Vega specifications and the other on web applications in general. Similar to Anteater, both systems recognize the importance of recording program behavior and providing global views of data to understand the inner-workings of a program. They differ from Anteater in their focus on debugging interactions with an application rather than the execution of a program.

Alsallakh et al. [3] created an Eclipse plugin that tracks specific tracepoints (equivalent to a breakpoint in a debugger) throughout a program's execution. Watchpoints can also be added to a field on which the tracer will track assignments. While the plugin's goals closely relate to those of our method, our method stands apart for two reasons: (1) we trace all execution structures, rather than user-defined tracepoints, along with the values desired by the user and (2) Anteater presents all this information in linked visualizations of the execution and values, providing necessary context.

Kang et al.'s [16] Omnicode provides run-time visualizations of program states, designed to aid novice users in building mental models about programs. Crucially, Omnicode visualizes values in a live coding environment which updates in real time. The primary visualization provided is a scatterplot matrix displaying plots for each variable over all execution steps. While Omnicode and Anteater have much in common, they were designed for different audiences (novices vs. data scientists) and thus support different types of programs and debugging tasks.

Trace Visualization Trace visualizations are often applied in support of understanding parallel programs [17, 33, 35]. Often, trace visualizations leverage icicle plots and flame graphs as the primary visual representation [5, 12, 17, 28, 35]. Anteater uses a visual encoding reminiscent of icicle plots and flame graphs to plot the execution trace, which we will call the generalized context tree (GCT), after Boehme et al. [6] However, Anteater differs in its definition of *trace*. While these previous traces capture the calling structure of the execution, Anteater extends this to capture values of marked variables and expressions, as well as loop behaviors. This extension provides additional context for how values are reached.

3 A TRACE-BASED VISUAL DEBUGGING METHOD

We present a new method for exploring and interacting with analysis executions, helping people to gain a deeper understanding of the inner-workings of their scripts that they cannot get from traditional debugging tools. It gathers information into traces as the script executes and automatically creates interactive visualizations of the gathered information. Fig. 3 presents an overview of the Anteater method. The method consists of 3 stages: (1) specifying and collecting the trace, (2) organizing the program data for interactive visualization, (tomatically generating interactive visualizations of trace data. In the remainder of this section, we describe each stage in more detail, along with implementation details for the Python version, and discuss the modifications necessary for debugging in Jupyter[1].

3.1 Specifying and Collecting Traces

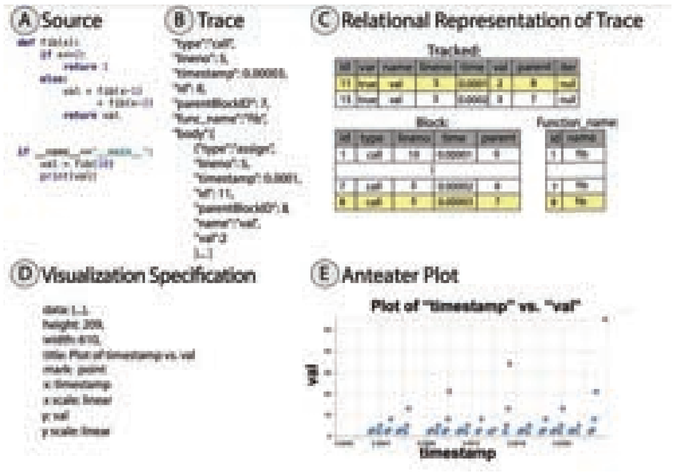
At the first stage, (Fig. 3(a)), analysts must specify what the trace collects. This stage is crucial because it dictates what data people can access and analyze. The trace must be carefully designed to provide the information desired for the given program debugging task. Standard traces typically focus on execution structure (e.g. function calls and iterative behaviors). In the context of exploratory debugging, we identified the importance of collecting intermediate variable values, in the context of the execution structure, to provide insight into value behaviors as well as the execution structure. Thus, the Anteater prototype emphasizes the collection of intermediate variable values to support more exploratory debugging tasks.

At this stage we must also consider the level of human input required for defining the collected data. We could take a "top-down" refinement approach where, by default, the tracer collects all program data available to it while providing controls for people to exclude certain values and execution structures from collection. This approach benefits people who have no clear idea of where to begin their exploration by providing them access to all of the intermediate values. On the other hand, this approach often generates a massive amount of data that can easily overwhelm people. In contrast, we could also take a "bottom-up" specification approach where people must specify precisely what should be collected. However, this approach requires substantially more effort from the analyst, negating some of the benefits of automation in this the method. After considering these two approaches, we take a mixed approach where Anteater automatically collects execution structure and relevant values (e.g. iterator values), but requires people to specify the variable values to collect. Ultimately, we chose this to help limit the size of the trace and allow Anteater to support larger programs.

Implementation As mentioned above, Anteater requires people to specify values to track. These values may be defined variables, expressions within a statement, or custom expressions that the program does not directly evaluate but provide insight into the behavior. Additionally, to eliminate unimportant functions from the trace, people may specify functions and libraries to exclude from the trace. Together, these two pieces create a trace specification.

Once defined, Anteater passes this trace specification to the tracer. Our tracer directly modifies the abstract syntax tree (AST) to instrument the source code (pre-execution) so that it records the trace data as the code executes. When creating the instrumentation, we must be careful to ensure to not disrupt the correct execution of the program. This requires careful definition of how to record execution structures, such

[1]An online Python version of Anteater can be found at `https://rjfaust.github.io/files/Anteater/`. The Anteater library for Jupyter will be re-sed on GitHub.


A Source
B Trace
C Relational Representation of Trace
Tracked:
Block:
Function_name
D Visualization Specification
E Anteater Plot
Plot of "timestamp" vs. "val"
val
timestamp

```
[illegible]
```

```
"type":"call",
"lineno":5,
"timestamp":0.00003,
"id":8,
"parentBlockID":7,
"func_name":"fib",
"body":[
    {"type":"assign",
    "lineno":5,
    "timestamp":0.0001,
    "id":11,
    "parentBlockID":8,
    "name":"val",
    "val":2
    [...]
```

```
data: [...],
height: 200,
width: 610,
title: Plot of timestamp vs. val
mark: point
x: timestamp
x scale: linear
y: val
y scale: linear
```

Fig. 4. An overview of how Anteater goes from source code to visualization. (A) shows the initial source code where we will track the variable "val". (B) shows a snippet of the trace produced as the instrumented script executes. Anteater then puts the JSON into a SQL table as shown in (C). From there, Anteater queries database to build the dataset of "val" instances and passes them to Anteater's visualization generator which generates a visualization specification (as shown in (D)) for the corresponding plot. Anteater then renders the specification to create a scatterplot of those points over time (shown in (E)).

as function calls. For example, if the program contains nested function calls, we must ensure that the we fully execute and record the inner call before executing the outer call. Additionally, we opt to collect all instances of tracked variables, not just the specified instance. To accommodate both of these constraints, we must first do an initial pass through the program to determine the scope of each variable to ensure that we only capture the desired instances and extract all function calls into isolated statements.

After completing the initial pass, we instrument the program with logging functions to record all executions structures and all specified program values. Fig. 3(b) shows a simplified example of our instrumentation. Once instrumented, Anteater compiles the AST into an executable program that generates the trace as it executes.

3.2 Organizing Program Data

Once we define what the trace will collect, we need to determine the best way to store that data (Fig. 3 (c)). The most important consideration here is how to store the data for easy, rapid querying of the data to build interactive visualizations. To enable people to efficiently navigate and inspect program data, we must provide responsive interactions that allow people to easily navigate between different portions of the data. In Anteater's traces, we identified two primary types of data: the hierarchical execution structure and sequential lists of instances of variable values collected throughout the execution. While the variable values reside in the hierarchical execution structure, for the purpose of creating global visualizations of value behavior, they better correspond to tabular data. Thus, the two types of data are fundamentally different and we need a data organization and that handles them separately to facilitate rapid querying and visualization of both types.

Implementation Fig 4 illustrates how we go from the source code to visualizations. Anteater writes the raw trace as a simple JSON file, shown in Fig. 4 (B). This allows it to easily capture the hierarchical structure of the execution as well as record meta-data about program blocks. However, this hierarchical JSON structure is extremely inefficient for querying the variable values in the trace. It requires traversing the hierarchy to gather the values every time they change. Additionally, it limits the support for more complex queries and interactions, such as custom filtering and joining. To combat this problem, we convert the hierarchical trace into a SQL database.

Data Type	Plot Type	Query
Q	Histogram	`SELECT`
N	Bar plot	`SELECT`
QxQ	Scatter	`SELECT`, `JOIN`
QxQxQ...	Parallel Coordinates	`SELECT`, `JOIN`
N, Q, QxQ	Small Multiples	`SELECT`, `JOIN`, `SORT ON`

Table 1. The above table shows the current visualizations supported and the SQL queries used to create these visualizations. We use "Q" for quantitative data and "N" for nominal.

Converting the trace to SQL yields several advantages. First, querying becomes much simpler. For basic visualizations, we now must simply write a `SELECT` statement to gather all instances of a tracked variable. To filter instances, we can simply add a `WHERE` clause to the SQL statement. Similarly, joining two variables becomes much simpler through the use of `JOIN`. Table 1 shows a table of visualizations supported by Anteater and the corresponding SQL query keywords used to collect the data.

Second, Anteater supports any visualization for which there exists a SQL query to select the appropriate data. In other words, forming the proper query becomes the only restriction to the range of possible visualizations. While the current implementation only supports a few visualizations, we could easily extend it to support others.

The last advantage comes from the decoupling of the visualizations and the data representation. The specification of the visualizations does not inherently depend on the representation of the data. A SQL query simply returns a list of data points for Anteater to use in the visualization. Because of this, we can easily swap in different visualization implementations, depending on the setting. For example, in our standalone tool, we found Vega-lite simpler and cleaner for generating visualizations but in our Jupyter adaptation, D3 was easier to apply and we were easily able to swap in the D3 implementations. The ability to quickly adapt for new visualization implementations further increases the extensibility and flexibility of Anteater.

3.3 Automatically Generating Interactive Visualizations

The last stage of our method is the automatic generation of interactive visualizations (Fig. 3 (d)). The visualizations should be well-principled and automatically generated with some support for customization.

Well-principled visualizations are those that following the most basic guidelines for visualization. Specifically, they should present overviews of the program first, rather than showing the details of individual time slices first like traditional debugging methods. The overviews give people a starting point for identifying interesting behaviors and replaces the trial and error of choosing appropriate breakpoints or locations for logging, particularly in exploratory debugging tasks where there may not be one set location for breaking. Additionally, there may be many value behaviors to inspect in a single execution. However, using traditional methods, serially inspecting and mentally tracking the behavior of multiple values at once requires substantial mental effort and memory recall to build a mental image of multiple behaviors. With our method, people can generate global views of each individual behavior, as well as generate combined views to inspect correlations between behaviors. After viewing the overview, we must supply people with the tools to filter down to execution areas or value subsets of interest and view the details of any relevant data.

Supporting automatic visualization removes the burden of processing and determining how best to inspect data from the shoulders of the analyst. However, in a general setting, it is difficult to definitively predict the needs of analysts. As such, some customization and alternative views should be supported. For example, non-linear scales may be better suited for some visualizations. Providing people with small, easy customization options enables them to adjust visualizations more appropriately for their task.

Implementation Table 1 illustrates how we choose the appropriate visualization for variable values. We determine the type of data selected (quantitative or nominal) and the corresponding visualizations. For ıple, a single quantitative variable is visualized using a histogram or a scatterplot against time In our prototype, we intentionally focused on quantitative data and chose well known visualizations to reduce the learning curve of using a new debugger. We further discuss the details of our visual design and implementation in Section 4.

3.4 Modifying Anteater for Debugging in Jupyter

To support inspection of analyses in Jupyter, we must make a few modifications to Anteater. Rather than directly modifying Jupyter, we ported Anteater into a library that analysts call from their Jupyter code. The library requires analysts to insert minor annotations into their program to indicate the entry point at which to start tracing, the values to track as they occur in the program, and the point at which to execute the trace. However, these annotations are comparable to the interactions required in the Python version.

Unlike the Python version, the library cannot automatically collect data from the entire execution, just the portion between the defined entry point and the trace point. As such, the notebook must execute fully (reaching the calls to the Anteater library) before Anteater can trace it. Thus, the section of traced code must execute twice - once to initiate and specify the trace and once to actually trace it. As a result, Anteater must ensure that the second execution exactly duplicates the original. To do this, Anteater creates a shadow version of the program that uses the same input state but executes independently of the original version. We describe the details of the three types of analyst annotations and the shadow execution below.

Initialization When the initialization call executes, it records its location (i.e. the executed cell) as the starting point for the trace. It then duplicates the current state of the program into a shadow state. This ensures that we save all of the inputs into the traced portion of the program. Additionally, we randomly generate and set a random seed to ensure that any random values in the script remain consistent across the two executions.

Setting Values to Collect Analysts must insert statements to specify the values they wish Anteater to record. When the script reaches one of these statements, Anteater records the names of the values to be tracked and the cell where they exist.

Running the Trace Once we reach the trace point, we must piece together the portion of the program to be traced. In the local state, Jupyter stores a log of the previously executed cells. Using this, we locate the starting point (as recorded during initialization) and merge all of the executed cells into a single program. Then, we must transform the program into a "shadow" version by renaming all of the defined variables and functions, and setting the input state to the shadow state saved during initialization. This ensures that the traced program executes from the same initial state as the original execution.

From here, the tracing and data organization follows that of the standalone tool, described above. To create the visualizations, we simply run plotting scripts using Jupyter's Javascript support, displaying them in the output area of the cell running the trace.

4 Anteater's Visualization Design

Anteater presents a new way of exploring and interacting with program executions helping people get a deeper understanding of the inner-workings of their programs that they cannot get from traditional tools. In this section, we describe the visualization design of Anteater and the features that facilitate the exploration of the execution trace.

4.1 Visualizing Program Data

Once Anteater collects and organizes the program data, it automatically presents interactive visualizations. Two types of visualizations are provided: a view of the execution structure, which we call the generalized context tree, and a visualization of the variable values. For ease of use, Anteater provides well understood visualizations of the program information but can be easily extended to support more complex or custom visualizations.

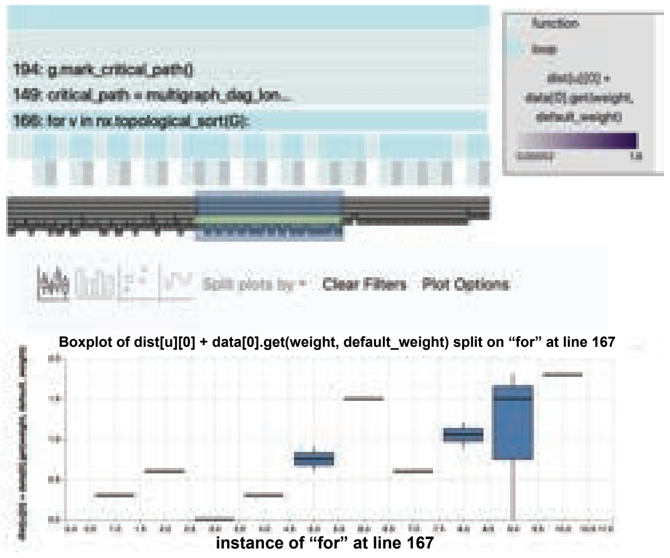

function
loop
dist[u][0] + data[0].get(weight, default_weight)
194: g.mark_critical_path()
149: critical_path = multigraph_dag_lon...
166: for v in nx.topological_sort(G):
Split plots by
Clear Filters
Plot Options
Boxplot of dist[u][0] + data[0].get(weight, default_weight) split on "for" at line 167
instance of "for" at line 167

Fig. 5. An example of Anteater splitting the data by a structural element. Anteater splits the data by instances of a for loop at line 167, which corresponds to iterations of the loop at line 166 (the selected block in the generalized context tree). The plot shows one boxplot per loop instance.

4.1.1 Generalized Context Tree

The generalized context tree (GCT), shown on the right side of Fig. 2-B, provides an overview of the execution structure. It illustrates the hierarchical structure of the execution, clearly showing when a function is called and from where. When inspecting program values, it provides the context of the values location in the overall execution.

The visualization has its origins in flame graphs and icicle plots. We chose this type of visualization because it is well known and understood for visualizing traditional execution traces. In Anteater, each rectangular block in the plot represents one of three things: a function call, a loop, or a collected value. The plot illustrates the hierarchy such that, for a given block, everything executed within that block (e.g. all values assigned within a function call) is drawn within the bounds of the bottom edge of the block. Time increases when moving form left to right in the plot such that everything to the left of a block was fully executed before that block. This allows analysts to easily read the visualization and understand when blocks execute relative to other blocks.

Additionally, the GCT highlights a single variable corresponding to the variable on the x-axis of the current plot. When someone assigns a variable to the x-axis, the GCT colors all blocks in the tree corresponding to that variable (which reside at the leaf level) by the value of the corresponding instance. Positive values range from white (low) to purple (high), while negative values range from white (least negative) to orange (most negative). Additionally, special colors are used for non-numeric values such as infinity and NaN. Fig. 6 colors the leaf nodes representing the variable "x" based on their value.

4.1.2 Variable Value Plots

The second visualization provided by Anteater, is a plot of tracked variables. To generate the plot, Anteater only requires people to specify which variable(s) they wish to view. Anteater then queries the database to retrieve a list of the selected values, in order of occurrence. when creating a plot, Anteater first checks the data types of each involved variable before looking up the plot type appropriate for the selected variable(s) (based on Table 1). Once Anteater determines the correct plot type, it generates the visualization specification. First, it determines the appropriate mark for the plot (bar, point, line, etc.) and plots the initial data. At this time, Anteater performs any necessary filtering and transformations (e.g. aggregation for histograms and filtering out non-numeric values in quantitative data such as "NaN" values). If quantitative variables have non-numeric values Anteater will concatenate additional subplots (horizontally or vertically depending on which

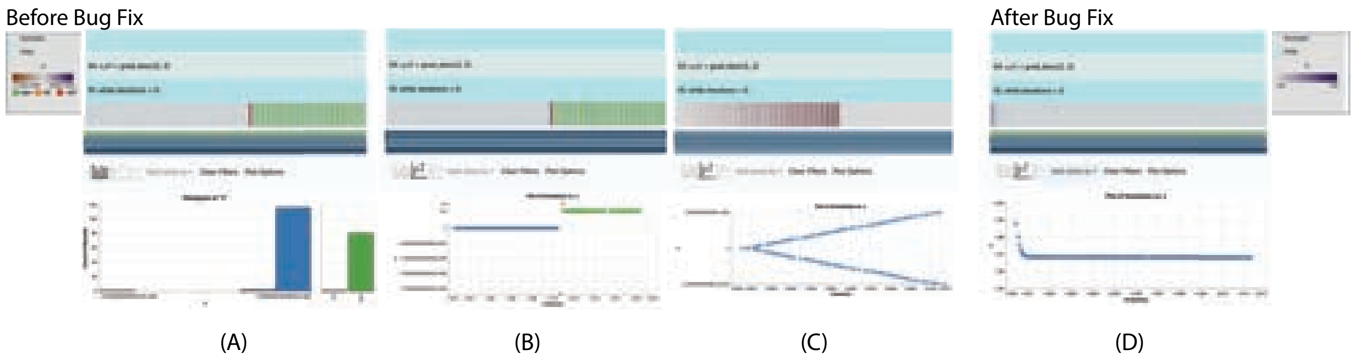


Fig. 6. Debugging Gradient Descent with Anteater, as described in Section 5.1. (A) - (C) show different visualizations of the buggy values. (D) shows the visualization after correcting the bug.

variable contains the values) to show these values, as in Fig. 6 A and B. This builds the base visualization for the specified variables.

4.2 Interacting with the Visualizations

Anteater's interactions are key in helping people get better understand their program behavior. We organize our interactions based on Yi et al.'s categories of interaction: Select, Explore, Reconfigure, Encode, Abstract/Elaborate, Filter, and Connect [37].

Select and Connect Anteater provides interactions to link the generalized context tree and the plot view, in both directions. These interactions enable filtering of data and refinement of views to allow for the inspection of subsets of the program data.

Anteater provides interactions on the plots and the GCT to link the two together. When an analyst selects a block in the GCT, the values shown in the plot filter down to include all values in the subtree rooted at the selected block, as shown in Fig. 2-A. In addition, to provide global context, the plot shows the values from the subtree rooted at the parent of the selected block. In the histogram, Anteater colors the bar representing the selected instance(s) blue while the coloring rest of the bars gray for context. Similarly, in the scatterplot, it colors the points representing selected instances while leaving the rest gray. These interactions enable people to narrow their scope to a specific portion of the execution. For example, if they want to look at the values that occur within a specific function call, the easily can do this by selecting the call in the GCT.

Anteater also provides linking from the plot back to the GCT. In the histogram, selecting a bar highlights the corresponding blocks in the tree. In the scatterplot, brushing over a set of points highlights the corresponding blocks in the trees, as shown in Fig. 2-B where the red blocks in the tree correspond to the brushed points. These interaction allows people to quickly locate where specific values occurred in the execution. For example, if a specific value looks appears abnormal, highlighting quickly illustrates where that value occurred and provides a location for inspecting additional values.

Explore Anteater supports two "explore" interactions: faceting values into groups and inspecting dependencies. The first interaction, faceting values into groups, enables people to view distinct subsets of a variable. Anteater provides grouping capabilities that allow analysts to facet the data into groups and create either a series of box and whisker plots on the same axes (one for each group) or small multiples of plots. The data can be split on either a related variable/expression from the trace (such as a boolean value) or a repeated structure in the execution, such as a loop, where each instance of the structure contains multiple instances of the tracked variables/expressions. For example, in Fig. 5, Anteater splits the plot on the outer loop and creates a box and whisker plot for each instance of the inner loop.

The second "explore" interaction supports the inspection of dependencies. Anteater determines what dependencies could exist for any instance of a variable. Prior to tracing, Anteater statically analyzes the source code to generate lists of potential dependencies for each tracked value. When someone selects a variable block in the GCT, Anteater accesses this list of potential dependencies and then, using context fr

the actual trace, prunes it exclude those that could not have occurred. Anteater the presents these dependencies by highlighting their corresponding blocks in the GCT. This allows analysts to quickly get an idea of which entities may contribute to that specific instance.

Reconfigure Anteater supports reconfiguration by allowing analysts to add multiple variables to a plot. This allows them to inspect relationships between pairs of collected values and observe any correlations that may occur. When someone selects multiple variables to plot, Anteater first determines their compatibility by attempting to align all or a subset of their instances via a common ancestor (e.g. a function call or loop). If the variables are compatible, Anteater plots them against each other in either a scatterplot or parallel coordinates (depending on the number of variables), allowing analysts to observe their relationship.

Encode Depending on the type of data presented, Anteater allows people to encode the data in a multiple ways. Using provided controls, people can quickly switch between the different plot types available for that datatype. Additionally, Anteater gives them controls to rearrange the axes of the plots as well as change the scales.

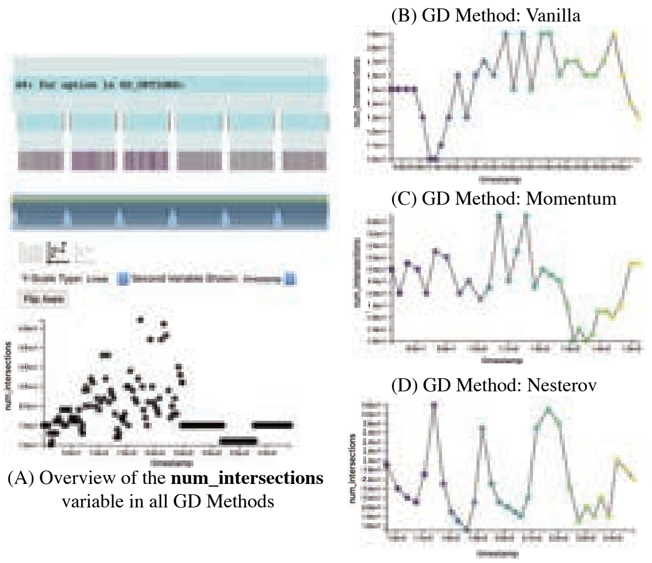
Filter Anteater supports three types of filter interactions on the plot and the generalized context tree to help people filter out unimportant information and emphasize important parts of the execution. The first type of filtering was mentioned above where clicking on deeper nodes in the context tree filters the value plots. Through this interaction, people can filter down the plot to interesting subsets of the data.

In the scatterplot, analysts can brush over a subset of points, right click, and select to filter out the values not in their brush. Anteater then removes all other points from the plot, effectively zooming in on selected points, and grays out any block not on the path to a shown point. Examples of this can be seen in Fig. 6-C. Similarly, in a bar plot or histogram, analysts can select bars and filter down to the corresponding values in the same manner.

One last way analysts can filter the visualization is by hiding parts of the generalized context tree. Right clicking on a block in the tree will expand the block to take up the entire width of the interface, increasing the size of all of its children and thus making them easier to see. However, in doing this, analysts might lose context of where they are exploring with respect to the execution. To retain this context, Anteater adds a smaller, grayscale version of the generalized context tree with a highlighter bar over it. When an analyst zooms in on a block, the highlighter narrows to indicate its place in the overall context tree. It also highlights the selected block in yellow, as well as any other blocks that are highlighted in the generalized context tree (from dependencies and brushed values). This allows analysts to see highlighted blocks even if they are outside of the visible portion of the generalized context tree. In Fig. 5, we zoomed in on the loop at line 166, but we see our location with respect to the whole GCT in the context bar.

4.3 How to Handle Objects

While Anteater will not directly collect objects, it provides a way nalysts to collect the information that interests them from the

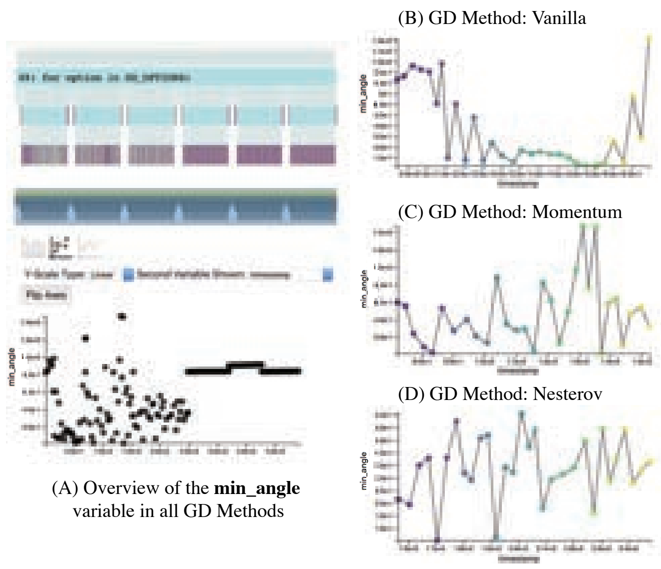

(B) GD Method: Vanilla

(C) GD Method: Momentum

(D) GD Method: Nesterov

(A) Overview of the **num_intersections** variable in all GD Methods

Fig. 7. Anteater views of the *num_intersections* variable over all gradient descent methods in (A), with closer inspection in three methods: Vanilla, Momentum, and Nesterov in (B), (C), and (D) respectively.


(B) GD Method: Vanilla

(C) GD Method: Momentum

(D) GD Method: Nesterov

(A) Overview of the **min_angle** variable in all GD Methods

Fig. 8. Anteater views of the minimum crossing angle variable over all gradient descent methods in (A), with closer inspection in three methods: Vanilla, Momentum, and Nesterov, in (B), (C), and (D) respectively.

object. To do this, the analyst locates the place in the program where they wish to inspect the object. At this point, they choose to create a custom expression for Anteater to evaluate and record that accesses the data of interest in the object (e.g. `obj.attr_1`). Each time the execution reaches this point, Anteater will evaluate and record result of the expression. This enables analysts to indirectly gather all of the information from objects that they wish to inspect without directly collecting the entire object.

The central challenges of collecting entire objects are the detection of every object modification and visualizing all information within an object. The first challenge would require Anteater to detect every time the object is mutated and record the new object state. Not only is the detection a difficult task, but the collection of all mutations of the object will inevitably lead to unmanageably large trace files. The second challenge requires additional input from the analyst on how to design the visualization of a given object. Rather than have analysts create their own visualizations, Anteater has them select the data they want to visualize from objects ahead of time and creates the visualizations for them.

5 Usage Scenarios

Here, we present two real-world scenarios, showcasing how Anteater derives insight into program behavior. These scenarios were developed on real programs through the author's debugging efforts using Anteater.

5.1 Gradient Descent

The first usage scenario we present inspects a script performing gradient descent. This script was collected from a Stack Overflow post [1]. The programmer struggled to determine why the resulting values of the variables "x" and "$x1$" were NaNs. In this scenario, we walk through how to use Anteater to understand the behavior and correct it.

First, we run the script with Anteater to track one of the misbehaving variables, "x." Fig. 6-A shows the resulting GCT and histogram. The histogram shows that much of the descent generates NaNs (the green bar). As a natural next step, we look at these values over time. We switch to a scatterplot which shows a plot of the variable "x" over time, shown in Fig. 6-B. Now, we clearly see that the value of "x" stays around zero, before becoming a very small negative, then going to infinity after which it reaches the NaNs. However, something strange happens where the value stays around zero and then suddenly becomes a very small negative. To investigate this, we filter the values to show only those points staying close to zero. We also switch to a symmetric log scale because we suspect that the values may not actually lie that close to zero. Fig. 6-C shows the resulting visualizations. We se

the value oscillates between increasingly large positives and negatives until it reaches infinity.

Now that we know the problem, we try to fix it. The oscillating values suggest that the gradient is exploding due to a training rate that is too large. In Fig. 6-D, after lowering the training rate and re-running the trace, the value quickly converges, as expected. Using Anteater, we quickly and easily track the variable "x" and observe its behavior throughout the execution. In a traditional debugger, detecting this behavior requires stepping through many iterations to view the values. After lowering the training rate, we repeat this process to determine if that fixed the problem. This involves significantly more interaction with the debugger than when using Anteater.

5.2 Graph Edge Crossing Angle Maximization

In this case study, we investigate a script that tries to minimize the number of edge crossings in a graph with while maximizing the size of the minimum crossing angle. Researchers in this space test out several gradient-descent methods to determine which one best balances these two costs. In this case study, we use Anteater to compare all of the different gradient-descent methods as well as investigate the differences between multiple runs of the best performing gradient-descent method.

5.2.1 Comparing Gradient Descent Methods

We first inspect how well each of the six gradient-descent methods balances the number of edges and the minimum crossing angle and how the optimization progresses towards its final result. We expect that each method would progressively work towards a better solution, generally increasing the minimum crossing angle while decreasing the number of edge crossings. The six methods considered are vanilla, momentum, Nesterov, Adagrad, Rmsprop, and Adam. Anteater tracks the number of intersections and minimum crossing angle throughout each optimization. The resulting visualizations of the number of intersections and the minimum angle are shown in Figs. 7 and 8, respectively. From Fig. 7-A we immediately see the last three methods (Adagrad, Rmsprop, and Adam) do not improve on the number of edge crossings. Fig. 8-A shows that the minimum angle behaves similarly. Thus, we immediately rule these three optimizations out because they do not seem to actually optimize the parameters.

Next, we take a closer look at the other three. First we consider the vanilla gradient descent, shown in Fig. 7-B and Fig. 8-B. In Fig. 7-B, the number of intersections initially decreases (as desired) but quickly starts to increase. It stays high for several iterations before dropping off at the very end, providing a good result. Similarly, in 8-B, we e that initially the minimum crossing angle is high (as desired) but quickly drops and stays low until the very end when it spikes back up. In general, we see that the optimization balances the the two parameters as desired, such that a lower number of edges corresponds to a higher minimum angle and vice versa. However, the vanilla gradient descent does not get progressively better results throughout the descent although it still returns a good result in the end.

We next look at Momentum gradient descent, shown in Figs. 7-C and 8-C. In Fig. 7-C we see a sort of oscillating pattern in the progression of the number of intersections where every time it reaches a low value, it starts increasing. Thus, every time it finds a decent value for the number of intersections, it starts moving back toward worse ones. In the end, we end up at a fairly mediocre value. The minimum angle behaves in a complementary way, every time it hits a high minimum angle, it starts decreasing. Again, we see that the optimization balances the two parameters as desired. However, to get a good result from this method, we have to hope that it terminates at a good solution before jumping to a bad one.

Last, we look at Nesterov gradient descent, shown in Fig.s 7-D and 8-D. In this method we see a pattern similar to Momentum gradient descent. In Fig.s 7-D, we see a trend where the number of intersections decreases for a while and then jumps very high and repeats. The minimum angle follows an inverse pattern where it becomes high and then drops very low before increasing again. To get a good result from Nesterov gradient descent, we again have to get lucky and end on a high point in the angle which corresponds to a lower number of intersections.

In the end, we chose Vanilla gradient descent because, while it did explore bad solutions initially, eventually it found a region of good solutions. Once in this region, the solution got progressively better and in the end vanilla gradient descent returned a good solution.

Using Anteater, we quickly and easily see the progression of each gradient descent method and how each optimization balances the two parameters, without any manual instrumentation or serial inspection of values. Without such a system, the burden of recording and creating a mental image of relevant values falls on the shoulders of the analyst. Additionally, serially inspecting recorded values makes it more difficult to observe these patterns. Thus, our method allows people to easily track and observe value patterns with minimal effort.

5.2.2 Inspecting a Single Gradient Descent

In the previous case, we determined that the vanilla gradient descent reached the best result, and while initially struggling to make positive progress, eventually began a progression towards a good result. However, because of this, we want to inspect a few more executions of the optimization to determine the consistency and stability of the optimization. In general, we found that in most cases, Vanilla gradient descent returns a good solution, however occasionally it would return a very bad result. Using Anteater, we inspected two contrasting instances, shown in Fig. 9. Fig. 9-A shows a particularly good instance where it immediately begins moving toward a good solution and never turns back. In contrast, Fig. 9-B, shows an instance where the optimization initially starts moving towards a bad solution, and never recovers. The instance described in the previous case falls somewhere in between these two extremes. Therefore we conclude that, while it tends to produce good results, it is perhaps not the most stable and we should explore other parameter settings to attempt to improve its stability.

6 Discussion and Future Work

Choosing what to track As stated earlier, Anteater only collects the variables and expressions that the programmer specifies. We explicitly chose to do this because it reduces the amount of unnecessary information presented and reduces the size of the trace. However, in some cases, such as when a programmer does not quite know what variable contains the bug, people may want suggestions of variables to inspect or they may want to inspect all variables. One future direction of work would be to explore methods for automatically suggesting variables to trace. One such method could be to sample all variables and present a sampling of the values. Another direction could be to apply machine learning to execution traces to identify potential areas of interest to help analysts find a starting point for their exploration

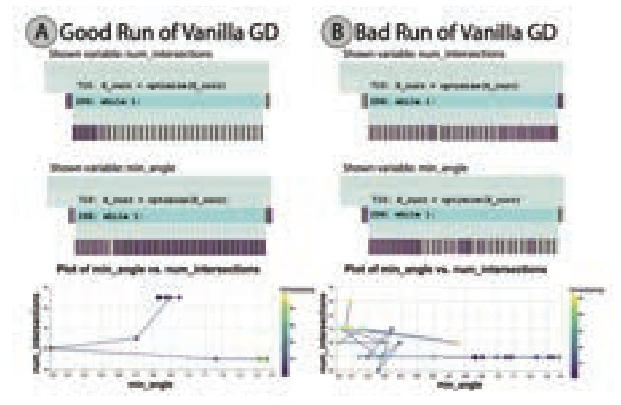

A Good Run of Vanilla GD
B Bad Run of Vanilla GD
Shown variable: num_intersections
Shown variable: min_angle
Plot of min_angle vs. num_intersections
min_angle

Fig. 9. Using Anteater to compare two runs of the Vanilla gradient descent that should maximize the minimum crossing angle while minimizing edge crossings. In (A) we see that the number of intersections rapidly decreases while the minimum angle increases. In contrast, in (B) we see that the number of intersections is increasing throughout the descent while the minimum angle decreases.

Further Support for Jupyter While the approach presented here provides greater support for inspecting program values than exists currently in Jupyter, it still does not quite fully support the Jupyter workflow. Because our approach uses requires calls to a library it cannot automatically update and detect when intermediate cells are re-executed. To support this, we either must require more substantial annotation or the ability to automatically detect when a cell runs, which requires under the hood modifications of Jupyter. As we do not want to bog people down with additional modifications, the latter presents a more promising opportunity and would allow us to hide the Anteater logic. In future work, we will explore how to integrate our tracing and visualization infrastructure directly into Jupyter to better capture and present the a-linear nature of Jupyter notebook executions.

Scaling Anteater will not scale to scripts that generate excessively large traces. Such programs typically make many calls or assign to tracked variables many times. In these programs, the traces become too large and the visualizations unreadable. Research exists on collecting the entire trace of large programs [25]; future work is needed to evaluate how to integrate Anteater with this method. In addition, Anteater works best with numerical data and has limited support for other datatypes. While it can present numbers, strings, and booleans, it does not support compound objects directly. Information about variables of these datatypes can still be visualized through the use of custom expressions, but we leave first-class support for more datatypes for future work.

7 Conclusion

In this paper, we presented a trace-based visual debugging method, Anteater and two prototype implementations, as a solution for improving the understanding and debugging of data science scripts. Through two usage scenarios, we demonstrated Anteater's ability to illustrate important execution behaviors that provide insight into exploratory debugging tasks. Our method serves as a preliminary step towards more effective debugging methods for data scientists, with several avenues of future work to further improve analysts workflows.

Acknowledgments

This work is partially supported by the NIST Graduate Student Measurement and Engineering Fellowship, through a grant with the GFSD, and the National Science Foundation under Grant # 2127309 to the Computing Research Association for the CIFellows 2021 Project.

References

[1] Gradient descent implementation in python returns nan. https://stackoverflow.com/questions/15211715/gradient-descent-implementation-in-python-returns-nan, 2013. Last visited on 2020-04-30.

[2] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS, pp. 53–62. ACM, New York, NY, USA, 2010. doi: 10.1145/1879211.1879222

[3] B. Alsallakh, P. Bodesinsky, A. Gruber, and S. Miksch. Visual tracing for the eclipse java debugger. In *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 545–548. IEEE, 2012.

[4] F. Beck, F. Hollerich, S. Diehl, and D. Weiskopf. Visual monitoring of numeric variables embedded in source code. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 1–4. IEEE, 2013.

[5] C.-P. Bezemer, J. Pouwelse, and B. Gregg. Understanding software performance regressions using differential flame graphs. In *IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, SANER, pp. 535–539, mar, 2015. doi: 10.1109/SANER.2015.7081872

[6] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz. Caliper: Performance introspection for hpc software stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '16, pp. 47:1–47:11. IEEE Press, Piscataway, NJ, USA, 2016.

[7] B. Borkholder. Mirur, 2014. `https://mirur.io`, last visited on 2020-04-30.

[8] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, pp. 473–484. ACM, New York, NY, USA, 2013. doi: 10.1145/2501988.2502050

[9] S. Chattopadhyay, I. Prasad, A. Z. Henley, A. Sarma, and T. Barik. What's wrong with computational notebooks? pain points, needs, and design opportunities. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2020.

[10] Y.-P. Cheng, C.-Y. Ku, W.-C. Pan, C. Yang, and T.-S. Lin. Toward arbitrary mapping for debugging visualizations. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pp. 605–608. ACM, New York, NY, USA, 2016. doi: 10.1145/2889160.2889167

[11] P. Gestwicki and B. Jayaraman. Methodology and architecture of jive. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pp. 95–104. ACM, New York, NY, USA, 2005. doi: 10.1145/1056018.1056032

[12] P. Gralka, C. Schulz, G. Reina, D. Weiskopf, and T. Ertl. Visual exploration of memory traces and call stacks. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 54–63. IEEE, 2017.

[13] J. Hoffswell, A. Satyanarayan, and J. Heer. Visual debugging techniques for reactive data visualization. *Comput. Graph. Forum*, 35(3):271–280, June 2016.

[14] J. Hoffswell, A. Satyanarayan, and J. Heer. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, p. 532. ACM, 2018.

[15] P. Jupyter. A visual debugger for jupyter, Mar 2020.

[16] H. Kang and P. J. Guo. Omnicode: A novice-oriented live programming environment with always-on run-time value visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pp. 737–745. ACM, 2017.

[17] B. Karran, J. Trümper, and J. Döllner. Synctrace: Visual thread-interplay analysis. In *Proceedings of the 1st Working Conference on Software Visualization*, VISSOFT, p. 10. IEEE Computer Society, 2013. doi: 10.1109/VISSOFT.2013.6650534

[18] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pp. 1–11, 2018.

[19] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '95, pp. 480–486. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1995. doi: 10.1145/223904.223969

[20] S. Litvinov, M. Mingazov, V. Myachikov, V. Ivanov, Y. Palamar, P. Sozonov, and G. Succi. A tool for visualizing the execution of programs and stack traces especially suited for novice programmers. *arXiv preprint arXiv:1711.11377*, 2017.

[21] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, Y. Wu, Y. Zhu, and M. Weimer. Vamsa: Automated provenance tracking in data science scripts. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1542–1551, 2020.

[22] T. Ohmann, R. Stanley, I. Beschastnikh, and Y. Brun. Visually reasoning about system and resource behavior. In *Proceedings of the 38th International Conference on Software Engineering Companion*, ICSE '16, pp. 601–604. ACM, New York, NY, USA, 2016. doi: 10.1145/2889160.2889166

[23] P. Pereira, J. Cunha, and J. P. Fernandes. On understanding data scientists. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pp. 1–5. IEEE, 2020.

[24] J. M. Perkel. Why jupyter is data scientists' computational notebook of choice. *Nature*, 563(7732):145–147, 2018.

[25] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pp. 535–552. ACM, New York, NY, USA, 2007. doi: 10.1145/1297027.1297067

[26] S. P. Reiss. Trace-based debugging. In *International Workshop on Automated and Algorithmic Debugging*, pp. 305–314. Springer, 1993.

[27] S. P. Reiss. The challenge of helping the programmer during debugging. In *2014 Second IEEE Working Conference on Software Visualization*, pp. 112–116, Sep. 2014. doi: 10.1109/VISSOFT.2014.27

[28] M. Renieris and S. P. Reiss. ALMOST: Exploring program traces. In *1999 Workshop on New Paradigms in Information Visualization and Manipulation*, pp. 70–77. ACM, 1999. doi: 10.1145/331770.331788

[29] D. Rozenberg and I. Beschastnikh. Templated visualization of object state with vebugger. In *2014 Second IEEE Working Conference on Software Visualization*, pp. 107–111. IEEE, 2014.

[30] A. Rule, A. Tabard, and J. D. Hollan. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2018.

[31] R. Schulz, F. Beck, J. W. C. Felipez, and A. Bergel. Visually exploring object mutation. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pp. 21–25, Oct 2016. doi: 10.1109/VISSOFT.2016.21

[32] B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The Craft of Information Visualization*, pp. 364–371. Elsevier, 2003.

[33] D. Socha, M. L. Bailey, and D. Notkin. Voyeur: Graphical views of parallel programs. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, PADD, pp. 206–215. ACM, New York, NY, USA, 1988. doi: 10.1145/68210.69235

[34] J. Sundararaman and G. Back. Hdpv: Interactive, faithful, in-vivo runtime state visualization for c/c++ and java. In *Proceedings of the 4th ACM Symposium on Software Visualization*, SoftVis '08, pp. 47–56. ACM, New York, NY, USA, 2008. doi: 10.1145/1409720.1409729

[35] J. Trümper, J. Bohnet, and J. Döllner. Understanding complex multi-threaded software systems by using trace visualization. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS, pp. 133–142. ACM, New York, NY, USA, 2010. doi: 10.1145/1879211.1879232

[36] N. Weinman, S. M. Drucker, T. Barik, and R. DeLine. Fork it: Supporting stateful alternatives in computational notebooks. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pp. 1–12, 2021.

[37] J. S. Yi, Y. ah Kang, J. T. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1224–1231, 2007.