A VISUALIZATION FIRST PERSPECTIVE ON UNDERSTANDING PROGRAM BEHAVIOR

by

Rebecca Faust

Copyright ©Rebecca Faust 2021

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF COMPUTER SCIENCE

In Partial Fulfillment of the Requirements

For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2021

THE UNIVERSITY OF ARIZONA
GRADUATE COLLEGE

As members of the Dissertation Committee, we certify that we have read the dissertation prepared by: Rebecca Faust, titled: *[Enter Dissertation Title]*

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

_____ Date: _____
Carlos Scheidegger

_____ Date: _____
Kate Isaacs

_____ Date: _____
Michelle Strout

_____ Date: _____
William Bernstein

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copies of the dissertation to the Graduate College.

I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

_____ Date: _____
Carlos Scheidegger
Dissertation Committee Chair
Computer Science

Acknowledgements

Contents

List of Figures

List of Tables

Abstract

This dissertation re-frames the problem of program understanding as a data analysis problem: if we can understand the data that exists in a program, we can understand the program. From this perspective, we apply visualization principles to take a visualization first approach to understanding program behavior. In past research, visualization researchers have crafted a set of principles for creating visualizations that effectively present data for human understanding. These principles have successfully been applied when creating visualizations in a wide variety of domains, demonstrating the effectiveness of visualizations created using these principles at presenting data for human consumption. However, while there exists work in software visualization as well as understanding programs without visualization, limited research exists on directly applying visualization principles to the domain of program understanding and debugging. This dissertation addresses this gap along two primary avenues: (1) using visualization to understand general programs and (2) using visualization to understand specific categories of programs, namely non-linear dimensionality reductions.

Along the first avenue, we present two visualization tools Anteater and ProgDiff. Anteater defines a mapping from the data collected in program traces to a visualization design framework that enables us to then apply visualization principles. It defines how trace data maps to common data structures used in visualization, and how to map from those data structures to effective interactive visualizations. Anteater then operationalizes this mapping to create a prototype implementation of a system for visualizing general Python programs. ProgDiff extends Anteater's mapping to support the comparison of multiple executions of a program through visualizations that apply visualization principles for comparison.

ProgDiff supports visualizing the effects of change in general Python programs. However, by narrowing the scope to specific classes of programs and specific types of change, we can create more descriptive visualizations of the effect of those changes.

DimReader is an example of this where we narrowed the focus to non-linear dimensionality reductions. We augmented these programs with automatic differentiation to simulate changes in the input data and record their effect on the positions of the projected points. After simulating this change, we applied visualization principles to create explanatory visualizations for understanding the behavior of the projection.

In this dissertation, we have shown how a data analysis perspective enables the creation of novel and effective visualizations for program debugging and understanding. We have shown two extreme points in the design space: Anteater and ProgDiff assume very little structure in the program and apply to very general programs whereas assumes structure characteristic of dimensionality reduction programs that enables the use of automatic differentiation. A natural question remains: given these two extreme points, how can we find a middle ground that combines the explanatory features of DimReader with the generalizability of Anteater and ProgDiff? Modern machine learning systems, specifically deep learning systems, encompass a broader class of programs while supporting automatic differentiation, thus providing a natural target for future investigations.

Chapter 1

Introduction

Understanding program behavior is a famously hard problem. It encompasses a great portion of programming tasks and, as a result, many researchers dedicate their time to creating new solutions to alleviate the burden of these tasks. Programs contain a multitude of data, both explicit (e.g. calling structure and variable values) and implicit (e.g. derived values). Our current practices in debugging and understanding programs inherently rely on this data. For example, when trying to understand the execution path to a specific function call, the calling structure is of great importance. At its core, the calling structure this is just a hierarchical data structure generated as the program runs. To understand the execution path, a person simply needs to analyze this data structure. Data analysis methods exist for analyzing such hierarchical data structures **lamping1995focus**. This leads to the question, what does it look like if we take a data analysis perspective on program debugging and understanding?

In this dissertation we re-frame the problem of understanding program behavior to take a data analysis perspective. From this perspective, if we collect the data from within a program the only remaining barrier to understanding the program is the ability to analyze the collected data. Well studied and effective methods for data analysis, such as the use of visualization, can be employed to help analyze this data. In this work, we transform programs to collect a variety of data as a program executes. After collecting the data, we present it for analysis using visualization.

Past research shows the effectiveness of visualization for completing data analysis tasks. To help people create effective visualizations for data analysis, visualization research established a set of principles for creating effective visualizations. At the forefront lies the principle that global views of data are more effective at illustrating the behavior of a dataset than serial views of textual data. Shneidermann's well known mantra of "Overview first, zoom and filter, details on demand" embodies this principle.

This principle drives the way we design data visualizations. Other, more task specific principles exists that further guide how we design visualizations, such as principles for creating comparative visualizations. Research shows numerous successful applications of these principles in a variety of domains. Despite their widespread use, we have yet to apply them for understanding program behavior. While prior work on software visualization exists, these works typically adhere to traditional perspectives of program debugging and understanding, often by adding visualization on top of existing serial debugging methods.

In this dissertation, we explore the how we can enable program understanding tasks when we take a data analysis perspective and employ visualization principles.

1.1 Solutions

We present solutions along two directions: (1) understanding and debugging general Python programs and (2) understanding a specific class of programs - non-linear dimensionality reductions (NDR's). To support debugging and understanding tasks in general Python programs, we present Anteater and ProgDiff. DimReader creates visualizations for understanding the behavior of NDR's.

1.1.1 Anteater

In Chapter 2, we introduce Anteater. Anteater applies the basic principles of visualization to create visualizations that enable program and debugging tasks in general Python programs. To do this, we first map the data generated by a program as it executes and the tasks in program debugging and understanding that rely on this data to Munzner's framework for visual design . This mapping identifies the data structures and types of data generated by a program and how to create effective visualizations of the program data. Anteater operationalizes this mapping into a prototype debugging

system for Python programs. This system automatically instruments a program to collect a trace containing the execution structure and desired variable values. It then presents this data using a variety of interactive visualizations. Anteater demonstrates the effectiveness of a visualization first approach for general program debugging and understanding tasks.

1.1.2 ProgDiff

ProgDiff extends Anteater to support the comparison of consecutive program executions. In doing so, ProgDiff supports a common debugging practice of making minor changes and inspecting the effects on the programs execution. People commonly use this method for tasks such as to validating bug fixes and comparing the results of different parameter settings. Despite the popularity of this method, existing debugging and understanding methods do not inherently support the comparison of multiple program executions. ProgDiff extends the mapping defined in Anteater and applies visualization principles of comparison to support comparative debugging tasks. It modifies the tracing infrastructure to detect and record changes from the previous version of the program. After executing the new version of the program, ProgDiff creates a mapping of the new trace to the previous version, marking the parts that were added, deleted, or changed. It passes the marked trace to the front-end where it generates comparative visualizations that highlight the differences between the two executions.

1.1.3 DimReader

Chapter 4 we introduce DimReader. Unlike Anteater and ProgDiff, DimReader only supports a specific class of programs: non-linear dimensionality reductions (NDR's). NDR's share a common structure: they take in a high dimensional dataset, perform a series of calculations, and return a two-dimensional representation of the original data. Narrowing our focus to this class of programs allows us to take advantage of this

common structure and create more descriptive visualizations that rely on this structure. Like ProgDiff, DimReader focuses on evaluating the effects of change. However, while ProgDiff supports general program changes, DimReader focuses on perturbations of the input dataset. By inspecting the effects of perturbations of the input data, we build an understanding of how he input data to influences the position of projected points. Additionally, DimReader does not require the data to be physically perturbed. Instead, it simulates the perturbation of the data through the calculation of the derivatives of the projected coordinates. DimReader augments NDR's with automatic differentiation to calculate the derivatives of the projected coordinates as the projection executes. Once DimReader collects the derivatives, it applies visualization principles to create global visualizations of the effect of the perturbations on the overall projection.

Chapter 2

Anteater

2.1 Introduction

Debugging and understanding program behavior is notoriously one of the most burdensome aspects of programming. It often requires programmers to trace through the execution steps and values of their program. However, most tools require people to build mental traces of their programs through the serial inspection of program values. Current practices often involve stepping through debuggers, inserting logging statements, or searching through source code, either manually or with a code browsing tool [69].

Additionally, traditional debuggers require programmers to set breakpoints at which they inspect the program state, stepping through its line-by-line operation. Tiarks et al. [112] observed that programmers experience difficulties in choosing breakpoint locations, often forgetting analysis details while navigating the code. Furthermore, traditional debuggers only present one view of the program: the whole program state at a single step in time. While this view has its uses in debugging, it does not help with bugs that present themselves over time (i.e. bugs where viewing a single instance of a variable is insufficient for detecting the bug, see Gradient Descent usage scenario). To detect those bugs, programmers must serially step through the values to build a mental image of their behavior.

However, this method of incrementally inspecting values to build an internal mental image of data directly contrasts the fundamental principles of data visualization. Consider the traditional value proposition of data visualization. Visualization practitioners now have a well-defined set of principles to drive the design, development, and testing of interactive visualization software [10], [23], [103]. In contrast to inspecting datasets serially, one element at a time, well-designed visual encodings can provide richer, faster, and more global views of potentially important patterns.

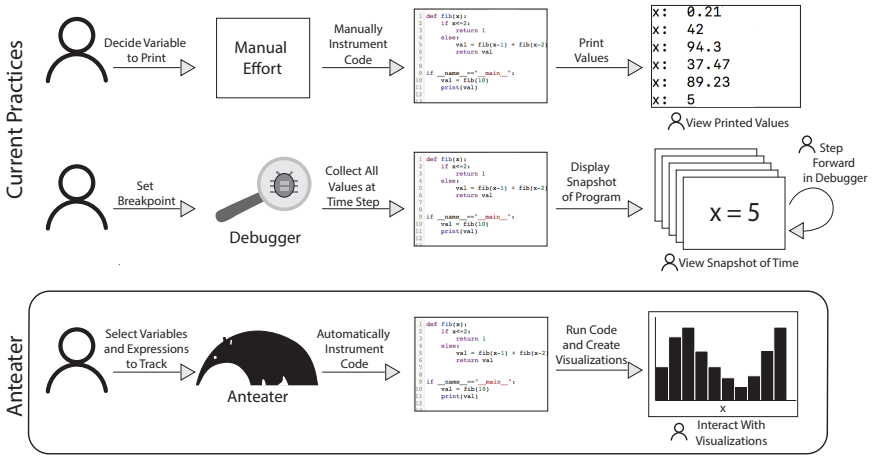

Figure 2.1: A programmer investigates a bug in their code. One common practice (top row) is to instrument the program manually to collect suspicious variables (here, x), and print their values. Manual instrumentation, however, is itself repetitive and error-prone. Another common practice (second row) is to use a debugger to stop the execution of the program 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 code to track variables along with the context of their execution. It presents the programmer with interactive visualizations providing a global view of values, enabling easy detection of erroneous values as well as interactions that narrow down the views to specific values.

Because traditional debugging methods only provide serial views of program data, they suffer from the same fundamental problem associated with the serial inspection of data. The widely used "Visual Information Seeking Matnra", as presented by Shneidermann [103], states "overview first, zoom and filter, then details-on-demand". We have seen numerous successful applications of this mantra to data analysis problems. However, we have yet to see this applied in a debugging context where serial inspection of data remains as the primary analysis method. We therefore see a need for an exploratory debugging solution that provides more effective global views of values, providing debugging the same set of affordances that interactive visualization provides to exploratory data analysis.

Consider the following debugging scenario. Programmer Patty has a bug in her code. Her program returns a value that seems unreasonable. She believes that the bug is occurring in a specific loop but cannot identify the root cause. Using a typical debugger, she sets a breakpoint at the beginning of the loop and runs the debugger. When the debugger reaches the breakpoint, she inspects the program values and takes a few steps through execution but does not yet see the bug. Patty continues the program until it hits the breakpoint again at the next iteration, repeating this process. She continues to step through each iteration of the loop but has little success in finding the bug.

After several iterations, Patty gives up on using the debugger and modifies the code with print statements. She prints the variable she believes causes the bug and runs the program. Patty scans through the printed values, trying to find any erroneous values, but her loop has many iterations and she quickly gets lost in the print statements.

Her next idea is to write the values to a file and plot them. Patty first alters her source code to write the values to a file. She then writes a script that reads the file and plots the values. Now she sees the behavior of every instance of the value and pinpoint the incorrect values. With this information, Patty returns to the debugger and stops

the program when it reaches the iteration containing incorrect values to find the root cause.

The scenario described above encompasses the typical ways programmers debug their programs [112]. While not every bug requires all of these methods, programmers typically use more than one of them. The fact that many programmers use a combination of independent debugging-methods when fixing their programs prompts the question: can we design a better debugger that 1) reduces the amount of manual instrumentation required, 2) gives the users greater control over the values they see, and 3) provides them with a visualization option automatically? While various debugging tools address aspects of these problems, no existing debugger comprehensively addresses all of them.

In response to these questions, we present Anteater, a system for debugging and understanding programs designed with principles of interactive visualization as a driving concern. We applied the framework for visual design as described by Munzner [81] to create a debugging system from a visualization perspective. Fig. 4.1 gives an overview of how Anteater compares to standard debugging practices. In taking a visualization-first approach, Anteater provides more informative overviews of a program's behavior while supporting interaction to dig deeper into the details of the execution. Rather than showing the whole state at a single step in time, it shows a single variable over the entirety of the execution. Anteater aims to reduce the effort required from a user by 1) automatically instrumenting programs to collect the values they want to inspect and 2) allowing them to browse values of interest easily throughout the entire execution, without resorting to a step-through debugger.

If Programmer Patty had been using Anteater, she could have easily set Anteater to track the value she believed to be raising issues along with any other values that she believed to be potential roots of causation. Anteater would then trace her program and provide her with visualizations to help her identify the iterations where the value was

incorrect. Patty could then filter down the execution tree to those iterations and inspect the rest of the values she tracked. With Anteater, Patty completes all of her debugging in one place using only a few interactions and requiring no manual instrumentation.

In this paper, we present a prototype implementation in Python that traces a Python program to capture not only the execution structure but also values of interest in context of the execution. Anteater then presents this trace to the user through interactive visualizations. Fig. 2.2 presents an overview of the visualizations provided by Anteater. In summary, this paper contributes (i) a goals-and-tasks analysis [68] of the typical practice of program debugging, (ii) a description and prototype implementation of Anteater in Python, aimed at providing interactive-visualization support to program debugging and understanding, (iii) a paired analytics evaluation of the prototype and its analysis, (iv) a comparative evaluation of Anteater and a traditional IDE debugger, and (iv) usage scenarios of real-world programs that show how our system compares to existing approaches.

2.2 Related Work

Literature Search We compare Anteater to work we have found in software engineering, user interface design, information visualization, and visual analytics. Specifically, we have searched the last 25 years of work related to visual debugging in the following venues: ACM ICSE, ACM CHI, ACM UIST, IEEE VIS, and the SoftVis symposium. The field of software visualization is large and we cannot hope to add every possible reference; we recommend both textbooks from Diehl and Stasko as starting points into the literature [37], [106]. Figure 4.1 gives a general overview of how Anteater compares to common debugging methods and table 2.1 gives an overview of how Anteater compares to the relevant existing work discussed in the this section.

Tool	● Supported Views				● Features			
	Single Variable, Single Time	Whole State, Single Time	● Single Variable, Whole Time	Whole State, Whole Time	Breakpoint/ Step-through	Variable Visualization(s) (*interactive)	● Execution Structure Visualization	Limitations
Anteater	✓		✓			✓*	✓	Medium Scale Python Programs
Omnicode [64]	✓	✓	✓	✓		✓*		Python Programs w/ < 10 variables
GDB	✓	✓			✓			
Print Statements	✓		✓					
Traditional Visual Debuggers [14], [30], [48], [89], [93]	✓	✓			✓	✓(* [14])	✓- [48], [89]	[14], [30], [48], [93] - Eclipse Plugin
Memory Visualization [2], [74], [108]	✓	✓			✓- [108]	✓(* [2], [108])		[74] - Eclipse Plugin
Trace Visualization [11], [53], [65], [90], [114]				✓			✓	Don't track values
Debuggers with Global Visualizations [4], [8], [19], [56], [97]	✓		✓	✓- [8]	✓	✓(* [4], [19],[56], [97])		[4], [8] - Eclipse Plugin, [19] - Web, [56] - Vega
[57]	✓	✓	✓	✓		✓*		Vega Specifications

Table 2.1: A comparison of Anteater with existing work in debugging visualizations. This table contains most references from the Related work section with the exception of [38], [73], [82], [104] which did not fit into the above categories. In this table, "single time" refers to a single instance at a specific point in the execution whereas "whole time" refers to every instance throughout the entire execution. The colored circles correspond to views and features that support the goals defined later in this paper. Note, because of the generality of G3, all systems aim to support this goal in some capacity and as a result, all features and views support it in some way. When a cell specifies specific references (e.g. ✓- [8] or [56] - Vega) this means that only those references have the corresponding view or feature.

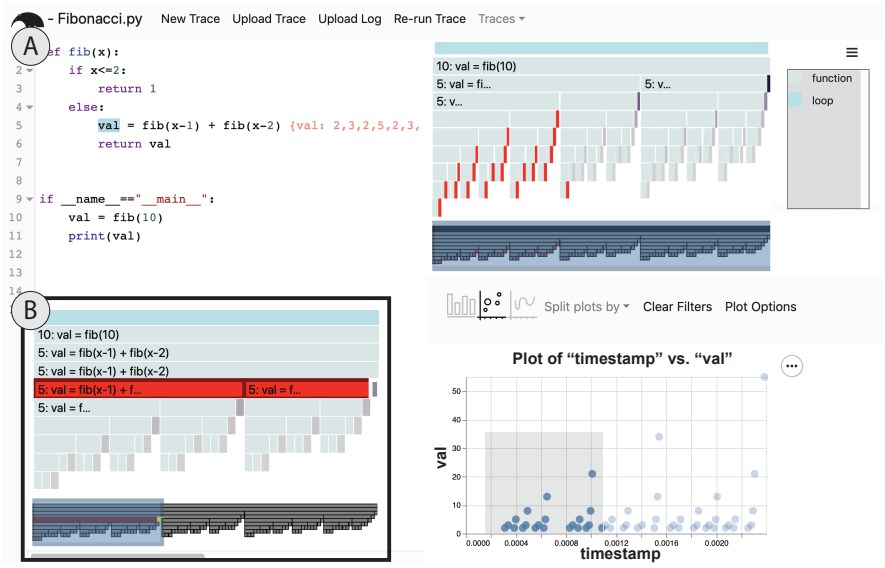

Figure 2.2: An overview of the Anteater UI on a recursive Fibonacci program, tracking the variable "val". (A) shows the UI presented by Anteater (not including (B)). The generalized context tree (GCT), or icicle plot, shown on the top right side of (A), shows the structure of the execution trace. The teal blocks represent function calls while the varying shades of purple represent the value of "val" at that instance. We can see the recursive calling structure of the Fibonacci function and can easily identify where it is repeating work. The plot currently shows a scatterplot view of the variable "val" over time. Brushing over the scatterplot highlights the corresponding instances in the GCT (the red blocks shown in the GCT on the right side of (A)) and the context bar. The scatterplot shows repetitive patterns that indicate that Fibonacci is doing redundant work. (B) shows a second view of the GCT (inset into the image of the main UI) after we've clicked on a block in the tree which caused its dependencies to be highlighted in red. This shows that the selected block (on the far right of the fifth row in the GCT in (B)), representing an instance of "val", depends on the prior two calls to the Fibonacci function (shown by the two blocks highlighted in red).

Visual Debugging Many attempts have been made to leverage visualization principles to augment the debugging process. Some efforts add visualization options to breakpoint and step-through debuggers [14], [30], [38], [73], [74], [89], [93]. 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 traditional debugger [14], [30], [93]. Others provide visualizations to show task-specific information about the execution, such as an overview of the heap and stack [2], [74] the impact of resource utilization on control flow [82], object mutation [97], or run-time state and data structures of the program [108].

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 [48], [73], [89]. In addition, some tools provide global views to show the behavior of values over the entire execution. Aftandilian et al. [2] give a global view of the heap by taking snapshots throughout the program. Schulz et al. [97] provide a global view of object mutations; if the object is numeric, the global view shows the value behavior throughout the execution. Some tools give global views of value behaviors by introducing sparklines next to the line of source code defining the value [8], [56]. 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. [57] and Burg et al. [19] 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. [4] created an Eclipse plugin that tracks specific tracepoints (equiva-

lent 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. The tool provides global views of tracepoint instances through line charts where interactions provide additional information about the program at that point and watchpoints through a step chart of the values over time. While the plugin's goals closely relate to those of our prototype, Anteater stands apart for two reasons. First, Anteater traces all calls and loops, rather than user-defined tracepoints, along with the values desired by the user. Second, Anteater presents all this information in a trace visualization with corresponding plots of the tracked values. This information can provide the context necessary to better understand why variables take on certain values.

The most similar tool to Anteater is Kang et al.'s [64] Omnicode. 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. general programmers) and thus support different types of programs. We compare Omnicode and Anteater directly in the Discussion section.

Trace Visualization Trace visualizations are often applied in support of understanding parallel programs [65], [104], [114]. Often, trace visualizations leverage icicle plots and flame graphs as the primary visual representation [11], [53], [65], [90], [114]. Anteater uses a visual encoding reminiscent of icicle plots and flame graphs in our plots of the execution trace, which we will call the generalized context tree (GCT), after Boehme et al. [12] 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 users with additional context for how values are reached; see Evaluation for a discussion of their utility.

2.3 Characterization of Anteater's Design

This section describes the visualization framework used to characterize the problem of program debugging from a visualization perspective. This perspective drove the design Anteater. Additionally, it uses the taxonomy presented by Maletic et al [77] to characterize the system design of Anteater.

2.3.1 A Visualization Perspective on Program Debugging

In this section, we use the framework for visual design described by Munzner [81] to characterize the problem of program debugging from a visualization perspective. This was the driving perspective used to create Anteater. The framework consists of 3 parts: (1) what - the data abstraction, (2) why - the task abstraction, and (3) how - the actual visualization design. This section describes how debugging maps to this framework, with the following three sections describing in detail how Anteater applies this perspective.

What - Data Abstraction: First, we need to understand the data involved in program debugging. As a program executes, it inherently creates a collection of data. This data includes items such as the values assigned to every variable, the value of parameters passed into function calls, the structure of the execution (e.g. calls and loops), time spent in each part of the program, etc. This data naturally maps to the data types outlined in the framework. We will focus on a subset of the data generated from sequential programs: the structure of the execution and the values assigned to variables. These two forms of data correspond to two data types outlined in Munzner's framework.

The first data type is a tree. A sequential programs naturally executes in a hierarchical tree structure: the root of the tree represents the entry point into the program, nodes represent execution steps (e.g. functions and loops), and the parent/child relationship signifies that the child was executed within the parent instruction (e.g. within a function call or loop iteration). The tree creates a node every time the program enters a function call or an iteration of a loop and it creates an edge between each node and the parent function or loop that contains its instruction. In Munzner's framework, this corresponds to the data types *node* and *link* and the dataset type *network/tree*. Additionally, each function call and loop contains additional *attributes*, such as the source code line that corresponds to its instruction, the name of the function, the value of the iterator for the loop, etc.

The second data type is a table. The values of program variables naturally organize into tables. Each instance of a variable is a data *item* (a row in the table) that contains several *attributes* that describe it (the columns in a table). To construct these tables, a program must create a record every time the program assigns to a variable. The attributes associated with a variables assignment include the line at which the assignment occurred, the node in the execution tree that contains that assignment, the actual value of the variable at that instance, etc. This clearly corresponds to *table* dataset in the framework, with the *item* and *attribute* data types describing the entries in the table.

Anteater uses this data abstraction to create a visual representation of a program. We describe the generation of this data with Anteater in more detail in the section titled "Tracing Infrastructure and Data Organization".

Why - Task Abstraction Now that we understand the data abstraction, we need to understand how the data analysis actions and targets outlined in the framework map to the domain of program debugging.

The high-level goal of debugging is to discover the source of unexpected or erroneous

program behavior. This behavior could either stem from misbehavior in the execution structure (e.g. a function not being called as expected) or misbehavior in the variable values (e.g. an incorrect calculation), or both. When debugging, programmers often inspect the programs data to generate a hypothesis about why the program is misbehaving or to validate an existing hypothesis about a bug. In Munzner's framework, this goal falls into the *consume* action of the **analyze** category. Additionally, this goal corresponds to the overarching aim of Anteater.

The framework allows us to separate the high-level goal of discovering unexpected program behavior into 4 mid-level actions that correspond to the programmers prior knowledge of the bug: *lookup*, *locate*, *browse*, and *explore*. These actions fall into the **search** category of Munzner's framework. First, a programmer may know precisely what to look for and where to look for it (corresponding to the *lookup* action). For example, if through prior debugging efforts they identified and corrected a calculation error, they may then re-execute the program to lookup the new value to ensure that it is correct. In this case, they know exactly what they are looking for and where to find it in the program data.

Second, a programmer may know what the bug is but not where it is occurring (corresponding to the *locate* action). For example, if a programmer knows that their program is producing an erroneous output value, they know that somewhere in the execution an erroneous value is assigned to the variable but they don't know precisely where.

Third, a programmer may know the general location of a bug, but not precisely what is causing it (corresponding to the *browse* action). For example, a program deviates from the expected execution path at line x but the programmer cannot immediately see what causes the deviation. They must browse the program data around this point to understand the behavior of the program at that point.

Fourth, a programmer may not know where the bug is or what is causing it (cor-

responding to the *explore* action). For example, a program finishes running but does not return from the expected point in the program. The programmer must explore the program data to locate where the program returns from and why it returns from this point instead of the expected point.

At the lowest level of action exist the **query** actions. These actions correspond to specific ways in which a programmer might query their program data for a debugging tasks. While performing a *lookup* or *locate* action where the programmer knows what variable or function call causes a bug, the may *identify* the specific instance of that call or variable to inspect all of the information collected about that instance. In contrast, when performing a *browse* or *explore* action, programmers want to *identify* areas in the program data that deviate from their expectations. A programmer may also want to *compare* the values of two variables to understand or verify an expected relationship between them. Last, for observing trends or patterns and identifying potential areas of erroneous behavior in variables or the execution structure, programmers may want to *summarize* the data with global views of the data.

The targets of these actions may be trends, outliers, or features of variable values or execution structure that highlight the misbehavior. They may also be correlations between variables that are perceived to be related or the distribution of a single variable. When identifying unexpected execution structure, the target may be the topology of the execution tree and paths through the tree that correspond to the execution stack of the program.

Anteater provides interactive views of both the execution structure and variable value data that allow people to perform these actions on program data. Global views of the program value allow people to *browse*, *explore*, and *summarize* the data. Interactions on the global views allow people to narrow their view to perform *lookup* and *locate* actions. The ability to plot multiple variables on a single plot allows people to *compare* variables.

How - Visual Design With the data abstraction and task abstraction defined, all that remains is creating visualizations of the data that facilitate the specified tasks. While there exist numerous options for creating visualizations of this data, we will focus solely on those supported by Anteater. We will not go into detail about the visual design here but will give a high level description of how Anteater's visualizations map to the framework. A full description of the visual design can be found in the section "Anteater's Visualization Design". The framework breaks up visualizations into four classes: **encode**, **manipulate**, **facet**, and **reduce**.

Anteater **encodes** the data using color and arrangement. For variable values, depending on the type of variable, Anteater arranges the data tables into histograms, barplots, scatterplots or parallel coordinates. Anteater arranges the execution tree into an icicle plot to illustrate the hierarchical structure of the execution and creates a color map to signfiy the type (function call, loop, etc.) of each block in the icicle plot.

Anteater allows programmers to **manipulate** the data through selections on the plots and execution tree. This enables them to connect the two views and inspect specific values in the visualization. Programmers then can **reduce** the data by filtering their selections to exclude irrelevant information.

Last, Anteater allows programmers to **facet** their data by partitioning it using a shared structure in the execution (such as a repeated function call or loop iterations) or related values from the program (such as a related boolean variable).

2.3.2 Characterizing Anteater's System Design

Several taxonomies exists for characterizing program visualizations [77], [86], [91], [101], [107]. While any of the taxonomies can apply to Anteater, we use the taxonomy from Maletic et al. [77] to describe it because we believe that it best characterizes Anteater with respect to the systems goals. This taxonomy breaks program visualizations into 5 dimensions: Tasks, Audience, Target, Representation, and Medium. We discuss each

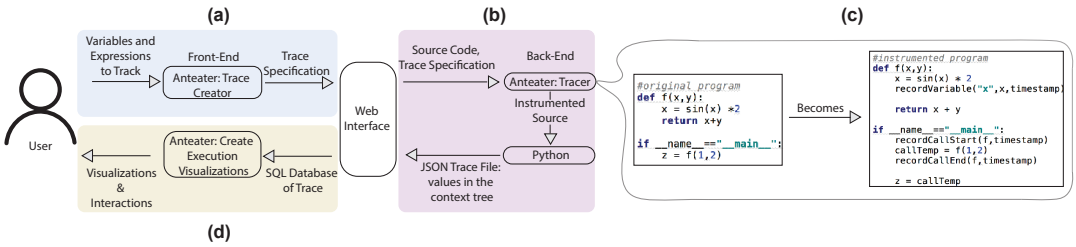

Figure 2.3: An overview of the Anteater system. In (a), a user chooses variables and expressions to track using the Anteater interface. This defines the trace specification. Then, Anteater sends the trace specification through the web interface to the python backend, along with the source code. Next, in (b), the Anteater tracer instruments the source code to collect execution information along with the specified values. (c) shows a simplified version of this instrumentation. After the code is instrumented, Anteater runs the program using python to create the program trace. This trace is passed back through the web interface to the Anteater front end where (in (d)) it is visualized and presented to the user.

of these dimensions individually in the remainder of this section.

Tasks The task dimension, as specified by Maletic et al., defines why the visualization is needed. Most standard debugging tools and methods lack support for global visual representations of the data internal to programs. They rely on serial approaches of inspecting a single instance of the data at a time. However, serial inspection of raw data tasks people with the significant mental burden of building an internal representation of an entire dataset [81]. Furthermore, because humans have a very limited ability to recall prior values when serially inspecting data, this internal representation suffers from inaccuracies caused by forgetting or misremembering past data. As Munzner stated, "Vis allows people to offload internal cognition and memory usage to the perceptual system" [81]. It does so by creating an external representation of the data that humans can comprehend more easily.

Anteater aims to create a debugging system that shifts the perspective from debugging programs through several serial views to take the previously described visualization first perspective on debugging. It focuses on giving programmers an overview of the

data within their program first and then providing them tools that allow them to delve into the details as desired. The "Task Analysis" section provides a more in depth inspection of the goals of Anteater and the tasks necessary to support those goals.

Audience The audience dimension defines who will use the visualization. Anteater aims to help python programmers understand their programs and diagnose misbehavior's in the programs they are running. While our prototype currently supports the visualization of program traces of a moderate size (around 225,000 recorded function calls and variable assignments), we believe that the design of Anteater is appropriate for general programming tasks in Python.

Target The target dimensions defines what aspects of the program are visualized. Anteater creates a trace as the program executes. Anteater focuses on collecting internal program values, such as variables and expressions, throughout the entire execution of the program. Additionally, these traces capture the calling and looping structure of the execution. The details of the tracing infrastructure of Anteater are discussed in the section "Tracing Infrastructure and Data Organization".

Representation This dimension defines how to convey the target information to the user. Anteater leverages well understood visualizations of each type of data collected to present the data to the programmer in an easily understandable way. It then pairs these visualizations with interactions that allow people to filter down to areas of interest in their program values and view details as desired. The visual design is discussed in depth in the section "Anteater's Visualization Design".

Medium The medium dimension defines where this information is displayed. We intend Anteater to be displayed in color on a laptop screen or an external monitor.

2.4 Task Analysis

In this section, we discuss Anteater's goals. The original inspiration for our goals came from Omnicode [64] and the Coz profiler [35]. We further refined our goals after exploring additional related work, characterizing the problem with Munzer's framework, and reflecting on our own experiences with respect to program debugging and understanding. The final goals below were derived after several iterations of system design and goal refinement.

G1: Identifying the source of unexpected execution behavior When programmers write and execute programs, they have some expectation of how their program should be behaving, e.g. what functions should be called and when. As a result, one goal of debugging is to identify what is causing an execution to deviate from what the programmer expected. To support this goal, debugging tools need to provide a view of the execution structure (see Features column of Table 2.1). Furthermore, this goal encompasses the subset of the **search** actions identified in the previous section that correspond to understanding the execution structure of a program. For example, a programmer may want to *lookup* a specific function call, *locate* an erroneous function call, *browse* a specific area of the execution structure, or *explore* the overall structure of the execution.

G2: Identifying the source of unexpected values and trends Similar to G1, programmers typically have a general ideas about what variable values they should observe during the execution of a program and thus desire to identify the root cause of unexpected values in the execution. To help programmers identify patterns and trends in the values of a variable, tools need to provide views of variables over the entire execution of the program. This corresponds to the "Single Variable, Whole Time" column in Table 2.1. In addition, keeping these values in context of the execution structure,

allows programmers to isolate areas of interest in the execution. Whereas G1 encompasses the subset of the **search** actions corresponding to understanding the execution structure of a program, this goal encompasses those corresponding to understanding the internal variable values of a program. For example, a programmer may want to *lookup* a specific instance of a variable, *locate* an erroneous variable calculation, *browse* instances of variables at a particular point in the execution, or *explore* the overall trends of a variable throughout the execution.

G3: Explore the behavior of an unfamiliar and/or complex piece of code
This goal encompasses a wide range of exploratory debugging and understanding tasks. We designed it to be general enough cover any programming situation that did not fit into the first two goals. For example, programmers are often tasked with understanding code written by someone else. Typically, this is no easy task and requires a significant amount of effort on the part of the programmer. Viewing the structure of the execution along with trends of variables throughout the entire execution serves as a starting point for understanding the behavior unfamiliar code. Similarly, programmers use well known but complex analysis algorithms that they write but do not fully understand how the algorithm operates. Understanding these algorithms is a difficult task that requires effort similar to understanding code written by someone else. This goal aims to encompass programming tasks like these. All debugging and understanding tools attempt to support this goal and as a result, all views and features described in Table 2.1 support this goal. This goal encompasses the subset of the **search** actions corresponding to understanding the general behavior of a program. This goal often corresponds to the *explore* and *browse* actions where the target is not concretely defined.

Under the framework of Lam et al. [68], G3 falls into the "Discover Observation" category and G1 and G2, fall into the "Identify Main Cause" category. From these goals, we derived several sub-tasks required to support the goals.

T1: Inspect all instances of a variable or expression It is often useful to look at all of the values that a variable or expression takes on to determine if it is behaving as-expected and to identify any erroneous values (supporting G2). Additionally, in an unfamiliar or complex program, it helps create a general understanding of the variables behavior (supporting G3). This task corresponds to the low-level actions *summarize* (e.g. view the trends of a variable) and *identify* (e.g. inspect an erroneous value) as described in the previous section.

T2: Identify what functions are called at runtime Often it is not clear from the static source code which functions will execute and when. However, identifying which functions are actually called during an execution is crucial for understanding how a program is operating (supporting G3) and identifying unexpected execution behaviors (supporting G1). Providing an overview of the execution (corresponding to the *summarize* action) allows people to see which functions are called at runtime and allows them to isolate misbehavior (corresponding to the *identify* action).

T3: Identify dependencies for a variable Understanding dependencies is crucial when trying to understand the behavior of a program. Identifying how a value is calculated, including the execution path required to complete the variable's calculation, allows programmers to better understand the underlying nature of the value in question (supporting G3). Such insight can lead to finding the cause of an unexpected value (supporting G2). This task supports the *identify* action in relation to viewing the dependencies of a specific instance of a variable.

T4: Identify interesting subsets of values Given a variable or expression, it is important to be able to identify the subset of values that correspond to interesting behavior. For example, if certain values indicate a failure in the program, they need to be identified so the surrounding values can be examined to understand the cause of the

behavior. This task supports G2 and G3 as well as the *identify* action.

T5: Observe relationships between values When debugging a program, programmers often investigate relationships between variables (supporting the *compare* action). For example, if variable x changes, how does variable y change? While these relationships may not be explicitly defined by the code, i.e., y may not directly depend on x, they often provide meaningful information to the programmer. Uncovering such relationships contributes to program understanding (supporting G3).

T6: Maintain context between runtime state and static source When trying to debug and understand a program, maintaining context with the actual code is critical. If the programmer is manually instrumenting print statements, they also must codify contextual information to derive insight, e.g., representing the location of a variable's modification. This task supports G1, G2, and G3.

A system that supports all of these tasks needs to track the execution structure of the program along with variable and expression values in the context of its execution. An execution trace fits this need as it naturally tracks the execution structure of a program and can be modified to also collect values. Once a system collects this data, it must present it in a way that allows for easy navigation through the data while supporting the defined tasks. We argue that visualization best way presents this information because it is known for providing overviews and context, highlighting relationships, and facilitating the filtering down to subsets of interesting information, all of which are needed to support these tasks. Anteater takes a visualization approach to program debugging and understanding that satisfies these goals through execution traces and visualizations. Currently, Anteater deals solely with single-threaded programs but we expect that this task analysis would need to be extended to satisfy our goals for multi-threaded programs.

2.5 Tracing Infrastructure and Data Organization

To support the goals and tasks defined in above, an execution trace with accompanying variable and expression values must be collected. Anteater implements a tracer that automatically instruments source code to collect its execution trace. Implemented in Python, the tracer relies solely on the Abstract Syntax Trees (AST) to facilitate the transformation of the source code. While Anteater currently only works with Python programs, the same principles can be implemented in any language that has the ability to transform source code in a similar way. After transforming the source code, Anteater runs the program, generates the trace file, and organizes the data in a way that allows for easy creation of interactive visualizations. Fig. 2.3 illustrates how the system operates.

2.5.1 Tracing Programs

This section goes into depth on part (a) and (b) of Fig. 2.3. First, it discusses how people can specify traces through the Anteater front-end. Then, it discusses how Anteater turns this trace specification in to program trace.

Specifying a Program Trace To fulfill T1 (inspect all instances of a variable or expression), Anteater allows programmers to define which variables and expressions to track, through interactions with the source code. Additionally, to eliminate unimportant functions form the trace, people may specify functions and libraries to exclude from the trace. Together, these two pieces create a trace specification. This corresponds to part (a) of Fig 2.3. Anteater also allows people to define additional custom expressions associated with their chosen variables that it evaluates and records each time it records the corresponding variable.

Anteater best supports numerical values but has limited support for strings and boolean values. While it cannot directly visualize lists and matrices, information about

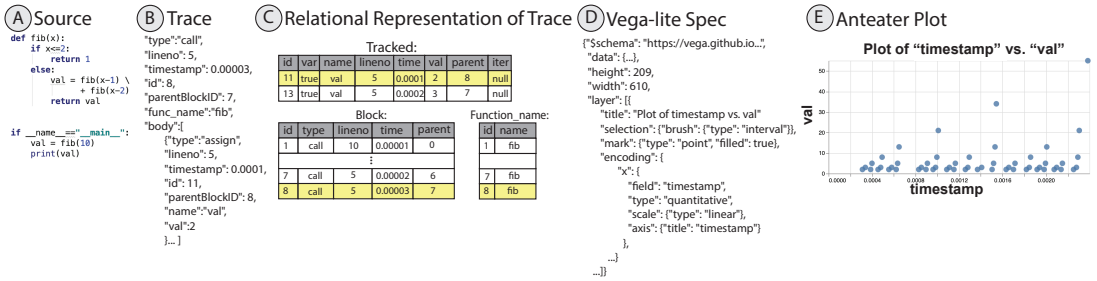

Figure 2.4: An overview of how Anteater goes from source code to visualization. (A) shows the initial source code. We are going to track the variable “val” After instrumenting the source code, as demonstrated in Fig. 2.3. The instrumented program creates a trace cell as shown in (B). Anteater then puts the JSON into a SQL table as shown in (C). From there, Anteater queries the table to select all points from “Tracked” that have the name “val” and passes them to Anteater’s Vega-lite generator which generates a Vega-lite 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)).

either structure can be tracked using custom expressions (see section ”How to Handle Objects”). Once the programmers complete the trace specification, Anteater passes it to the tracer in the backend for processing.

Note, the tracer will only collect the variables and expressions defined in the trace specification. We explicitly chose to do this because collecting the entirety of data associated with every variable in the program leads to the collection of massive traces filled with a significant amount of irrelevant/unnecessary data. Many variables residing in code have little importance in describing the program’s behavior. Thus, Anteater allows the user to select the important variables to track. This decision discussed more in the Discussion section.

Anteater’s Tracer When a user chooses to create a trace, the Anteater back-end is passed a trace specification containing a list of variables and expressions to track and a list of functions and libraries to exclude from the trace. The tracer indexes through these lists and determines the scope in which each item resides to ensure that it only

tracks/excludes the specified items. For example, if two disjoint functions both define variable x, the tracer will only track the one the user selected.

Once Anteater determines the scope of each item, the tracer uses the Python *ast* library to parse the source code into its AST. It then performs a series of traversals of the AST to collect information about the source code and transform the program to trace the execution and desired values.

In the first traversal through the AST, no transformations occur. Rather, Anteater collects information about functions, loops, and dependencies. For functions and loops, it collects the lines at which the function definition or loop begins and ends. This information enables more detailed linking between visualizations and source code. For dependencies, the tracer traverses through the code and, for each variable, stores functions and variables on which it directly depends in the source text.

Once all of the static data has been retrieved from the source code, Anteater begins transforming it. A second traversal through the AST transforms the code to isolate all function calls from their respective expression statements and expand list comprehensions into `for` loops. Anteater pulls all function calls that do not stand alone out of their expressions and assigns them to a temporary variable that replaces the call in the original expression (e.g., $x = 2 * f()$ becomes $tempF = f(); x = 2 * tempF$). This allows Anteater to easily capture when and in what order functions are called.

Next, the tracer performs the main transformations to insert the instrumentation that collects the trace. As the tracer traverses the AST, it always pauses at assignment, call, and loop nodes. When it reaches an assignment node, it checks the trace specification to determine if the target variable needs to be tracked. If so, it inserts new nodes into the AST that record the value of the variable after assignment.

When the tracer reaches a call node, it first checks if the trace specification excludes the function. If not, the tracer wraps the call with AST nodes to record the entry into and exit from the call. A simplified example of this transformation is shown in Fig. 2.3

(c).

When the tracer reaches a loop, it creates a counter to track the iteration of that loop and inserts new instrumentation to record the start of the loop. As it traverses the body of the loop, any time the tracer creates a new record, it records the iteration in which that record occurred. Tracking the iteration binds together groups of records in the trace that occurred in the same part of the execution (i.e. records that occurred in the same iteration).

Lastly, the tracer transforms the program to record expressions. Unlike variables, expressions occur in a variety of AST nodes. As the tracer visits each node, it checks if the line containing the node also contains a tracked expression. If it does, the tracer isolates the expression from the line, assigns it to a temporary variable, and then replaces the expression in the original line with the temporary variable. This ensures that the expression only executes once and that the trace records its exact behavior during the execution of the program.

Once Anteater completes the instrumentation, it compiles the AST into an executable program, which generates the trace as it executes.

2.5.2 Data Organization

Fig 2.4 illustrates how we go from the source code to visualizations. After Anteater instruments the source code, it runs the modified program and creates the trace file. Anteater writes the raw trace as a simple JSON file, shown in Fig. 2.4 (B). This allows it to easily capture the hierarchical structure of the execution as well as record data about program blocks as attributes in the corresponding JSON block. Anteater then passes the trace to the front-end. While convenient for collecting the trace, JSON is less convenient and flexible for querying the trace which limits the range of possible visualizations and interactions. To support more complex visualizations and interactions, Anteater converts the JSON 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 2.2: 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.

As shown on the right side of Fig. 2.3 (d), Anteater converts the JSON trace into SQL tables. The primary two tables store (1) the attributes of the nodes in the execution tree (the "block" table), and (2) the attributes of all instances of tracked variables and values (the "tracked" table). Fig 2.4 B-C demonstrates how to convert form JSON into the corresponding SQL tables.

Additional tables exist, such as "function_name" and "for_loop" that store additional information about certain types of blocks. The "custom" table stores the values of custom expressions that are collected alongside the variables and expressions selected in the source code.

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 2.2 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 datapoints for Anteater to use in the visualization. Because of this, we easily adapted Anteater to use Vega-Lite [95] specifications to generate visualizations. Furthermore, new visualization implementations can be plugged in with minimal effort to adapt them to fit into Anteater. This further increases the extensibility and flexibility of Anteater.

2.5.3 Generating Vega-lite Specifications

As mentioned previously, a SQL query simply returns a list of datapoints. Anteater then simply needs to generate a Vega-lite specification appropriate for the specified data (the final step of Fig. 2.3(d)). A snippet of a generated specification is shown in Fig. 2.4 (D) with the corresponding plot in (E). Leveraging the power of Vega-lite allowed us to easily create clean, interactive visualizations that are customized to best present the data selected by the programmer.

2.6 Anteater's Visualization Design

Anteater presents a new way of exploring and interacting with program executions helping users to gain a deeper understanding of the inner-workings of their programs that they cannot get from traditional tools. In the previous section, we discussed how Anteater creates the execution trace. Here, we describe the visualization design of Anteater and the features that facilitate the exploration of the execution trace. As we walk through the design, we will describe the features in context of a simple Python program that runs a recursive Fibonacci function. In addition, we use Yi et al.'s categories of interactions [121] to classify our interactions and further validate our design. Anteater uses Vega-lite to generate all visualizations, with the exception of the generalized context tree.

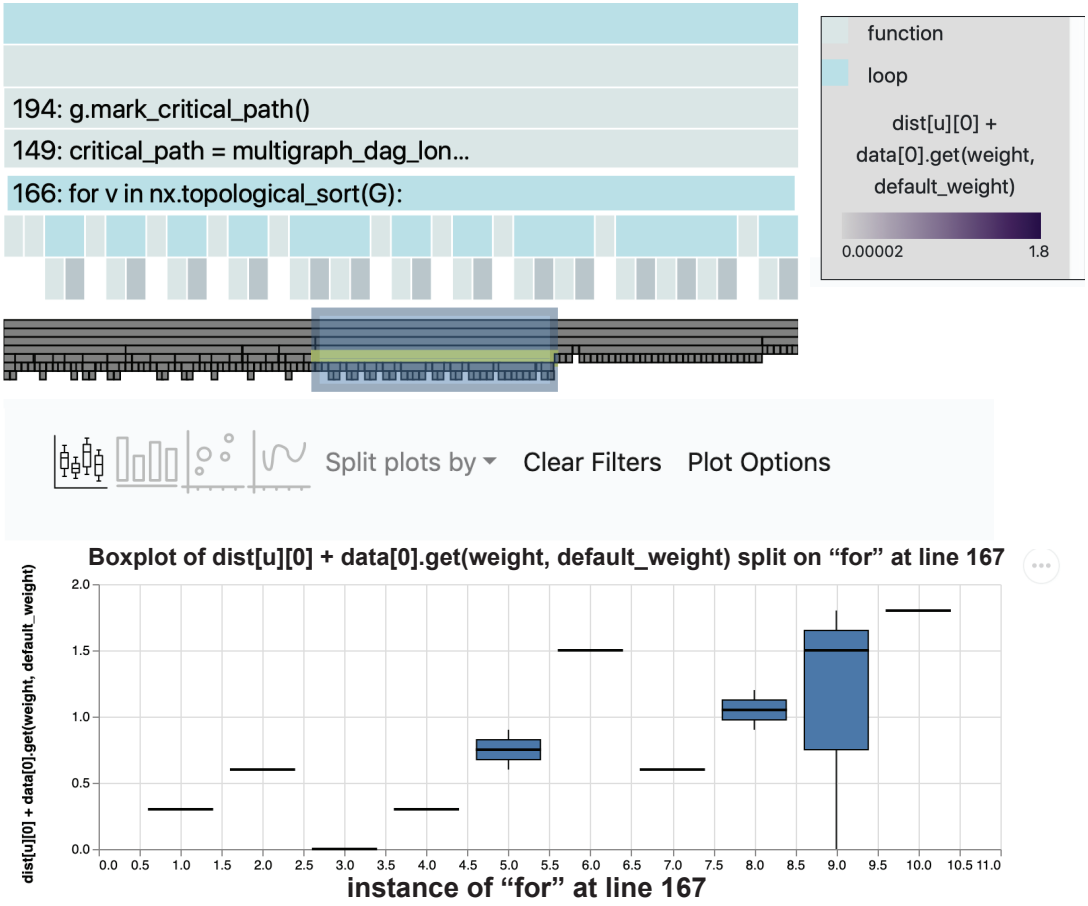

Figure 2.5: An example of Anteater splitting the data by a structural element. Anteater splits the data by instances of a for loop at line 167, which corresponds to iterations of the loop at line 166 (the selected block in the generalized context tree). The plot shows one boxplot per loop instance.

2.6.1 Visualizing Program Data

Once the tracer returns the execution trace, Anteater generates 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/custom visualizations.

Generalized Context Tree

The generalized context tree (GCT), shown on the right side of Fig. 2.2-A and in Fig. 2.2-B, provides an overview of the execution structure. 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 traces. In our setting, each rectangular block in the plot represents one of three things: a function call, a loop, or a variable assignment. The icicle plot shows the hierarchy so that, for a given block, everything that is within that blocks bounds below it, is a child which means it executed within the code of the parent block (i.e. in that call or loop iteration). For example, in Fig. 2.2-A, the block in the second row labeled "10: val = ..." is the initial call into the Fibonacci function and everything below that happens within that call. The generalized context tree can be used to determine which functions executed and when, fulfilling T2 (identify which functions are called at runtime).

As we move from left to right in the plot, we are increasing in time; everything to the left of a block was fully executed before that block. This allows users to easily read the visualization and understand when blocks are executed relative to other blocks.

The GCT highlights a single variable corresponding to the variable on the x-axis of the plot. When the user 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). In Fig. 2.2-A, Anteater colors the leaf nodes representing the variable "val" with varying shades of purple. Deeper leaves are shaded much lighter, which indicates small values at those instances; this corresponds to the deepest Fibonacci calls returning the smallest values. Coloring blocks in this way shows the behavior of values in the context of the whole execution. Every other variable or expression that appears in the trace still appears in the generalized context tree but Anteater colors them gray to keep focus on the selected variable.

Before creating the GCT, Anteater must organize the data into a hierarchy that it then passes to the D3 library to generate the visualization. To organize the data into the hierarchy, Anteater starts at the root block that represents the whole module and queries the database for all of its child blocks. It then adds these blocks as its children to the hierarchical data structure and repeats this process for each child block. In essence, this re-builds the tree in a manner similar to depth first search.

Variable Value Plots

The second visualization provided by Anteater, is a plot of tracked variables. Similar to when creating the trace specification, programmers add tracked variables and expressions to the plot by right clicking and selecting to add it. Anteater queries the database to retrieve the specified variables. When Anteater initially reads in the trace, Anteater checks each tracked variable and expression to determine its type (quantitative or nominal). Thus, 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 2.2). Once Anteater determines the correct plot type, it begins generating the Vega-lite specification. Initially, it creates the base layer that sets the mark for the plot (bar, point, line, etc.) and plots the initial data. In this layer, Anteater performs any

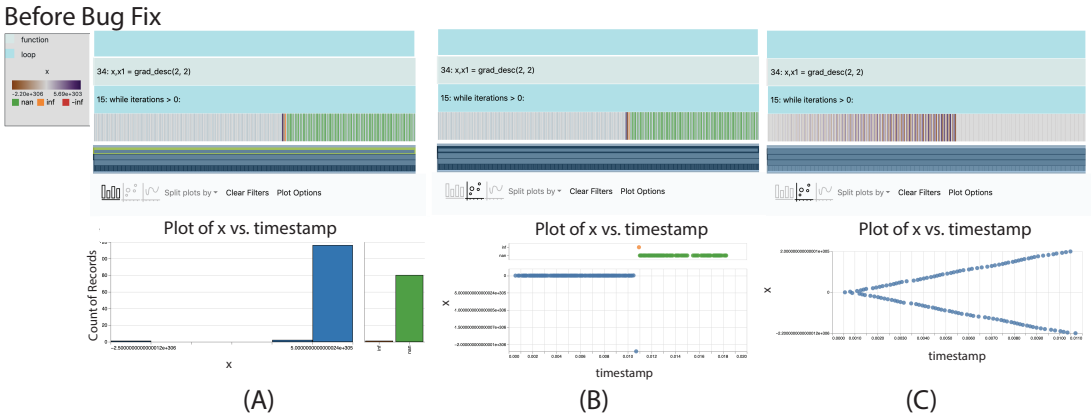

Figure 2.6: Debugging Gradient Descent with Anteater. In (A) it is immediately apparent in both the generalized context tree and the histogram that there is a bug causing NaN's, shown in green in both the histogram and GCT (NaN means "Not A Number", special floating-point values that indicate numerical failures). In (B), we switch to the scatterplot view to see how the values behave before they become NaN. The values are mostly centered around zero before becoming an extremely small negative, then going to infinity and becoming NaN. We suspect that the values centered around zero are not actually zeros so we filter the values in the scatterplot to allow us to zoom in on them and switch to a symmetric log scale, shown in (C). Now we see that the values are oscillating which suggests the problem of exploding gradients caused by a training rate that is too large. Fig. 2.7 shows the Anteater visualizations after correcting the bug.

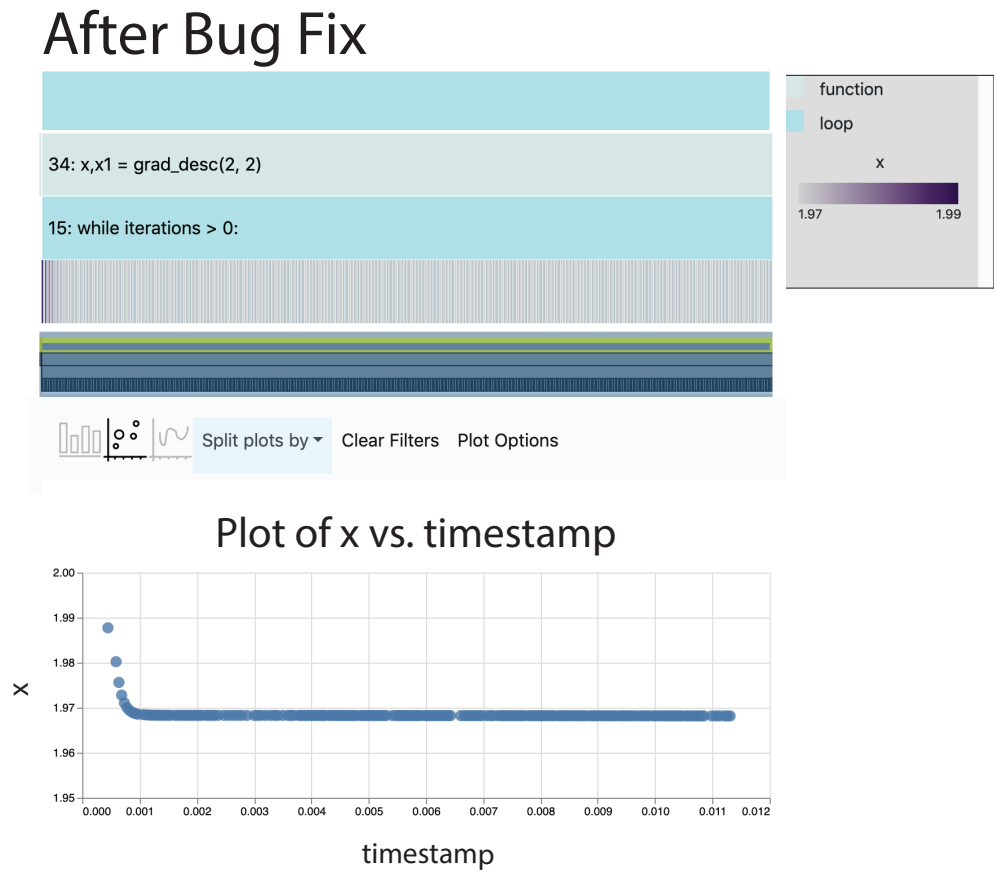

Figure 2.7: Debugging Gradient Descent with Anteater. The plot and generalized context tree after we correct the bug from Fig. 2.6. To correct the bug, we reduce the training rate and can see that the value quickly converges as expected.

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 variable contains the values) to show these values. Vega-lite allows Anteater to sync the axes of the subplots with the main plot in the base layer, as in Fig. 2.6 A and B. This builds the base visualization for the specified variables.

2.6.2 Interacting with the Trace Visualizations

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

Select and Connect Anteater provides interactions that connect related views in the following way: interactions to link the generalized context tree and the plot view (in both directions) and interactions to link the visualizations to the source code. Additionally, the interactions linking the generalized context tree to the plot view also serve to select portions of the execution data that are of interest.

Anteater provides interactions on the plots and the GCT to link the two together. When a user 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. In addition, to provide global context, the plot shows the values from the subtree rooted at the parent of the selected block. As shown in the histogram in Fig. 2.9-B, Anteater colors the bar representing the selected instance(s) blue while the coloring rest of the bars gray for context. In the scatterplot, it colors the points representing selected instances while leaving the rest gray. Anteater also provides linking from the plot back to the GCT. In the histogram, selecting a bar highlights the corresponding blocks in the tree, as

shown in Fig. 2.9-A. In the scatterplot, brushing over a set of points highlights the corresponding blocks in the trees, as shown in Fig. 2.2-B where the red blocks in the tree correspond to the brushed points. Anteater enables these selections by adding specific parameters to the Vega-lite specification. These parameters specify the type of selections available (e.g. brushing or clicking) and the visual effects of the selections (e.g. changing color or opacity of unselected points). Furthermore, Vega-lite's data listeners allow Anteater to monitor these selections and update linked views accordingly. These interactions support T4 - identify interesting subsets of values - by allowing the user to pinpoint interesting values in the plots and locate them in the execution.

Additionally, when exploring the execution, it is important to connect back to the source code to maintain the context of the execution. On its own, the generalized context tree is fairly abstract. To provide necessary context, when the user selects a block in the generalized context tree, the source code jumps to, and highlights, the corresponding section of the code. If it corresponds to a function call whose definition resides in the source file, it also highlights the corresponding function. This interaction, paired with a preview of the corresponding source code on the blocks, supports T6 - maintain context between runtime state and static source - by allowing users to navigate the execution trace without forgetting their place in the source code.

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 the user 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. 2.5, 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. To support T3 (identify dependencies for a variable), Anteater determines what dependencies could exist for any instance of a variable. To find all dependencies for a variable, Anteater accesses the variables dependency list generated during tracing, and then, for each dependency in that list, it accesses their dependency lists. This continues until Anteater builds a comprehensive list of all possible dependencies.

After creating the list of dependencies, Anteater uses context from the execution trace to eliminate some possibilities and present the remainder to the user. When a user selects a block in the generalized context tree that represents a variable, Anteater checks 2 sets of blocks: (1) any siblings of the selected block that were fully executed before it and (2) the siblings of all ancestor blocks of the selected block that were fully executed prior to the selected block. From these sets of block, Anteater finds any blocks that are on the list of possible dependencies. For any block that is on the list, it is highlighted in the generalized context tree to show the user the user on which parts of the context tree that selected block depends. This allows the user to quickly get an idea of which entities may contribute to that specific instance. In Fig. 2.2-B, the selected instance of "val" depends on the prior two calls to "fib".

Reconfigure : Anteater supports reconfiguration by allowing users to add multiple variables to a plot (supporting T5 - observe relationships between values). 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 the user to observe their relationship. Compatible variables share a common ancestor and have 1-1 instances within that ancestor. Anteater provides an options menu that allows

programmers to swap or change the scales on axes using the "Plot Options" menu.

Encode Depending on the type of data presented, Anteater allows people to encode the data in a multiple ways. People can click on the icons above the plot to 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, which helps support T4 (identify interesting subsets of values). The first type of filtering was mentioned above where clicking on deeper nodes in the context tree filters the value plots. Through this interaction, users can filter down the plot to interesting subsets of the data.

In the scatterplot, users 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. 2.6-C and Fig. 2.9-C. Similarly, in a bar plot or histogram, users can select bars and filter down to the corresponding values in the same manner.

One last way users 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, users might lose context of where they are exploring with respect to the execution. To retain this context, we add a smaller, grayscale version of the generalized context tree with a highlighter bar over it. When the user 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 users to see highlighted blocks even if they are outside of the visible portion of the generalized context tree. In Fig. 2.5, we zoomed in on the loop at line 166, but we see our location with respect to the whole generalized context tree in the context bar.

2.6.3 How to Handle Objects

While Anteater will not directly collect objects, it provides a way for users to collect the information that interests them from the object. To do this, the user 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 record that accesses the data in the object that interests them. Each time the execution reaches this point, Anteater will evaluate and record the value of the expression. This enables users to indirectly gather all of the information from objects that they wish to inspect without directly collecting the entire object.

The central challenges with collecting entire objects are the detection of every modification to the object 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 state of the object. 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 would require additional input from the user on how to design the visualization of the object given the information it contains. Rather than have users create their own visualizations, Anteater has them select the data they want to visualize from objects ahead of time and then creates the visualizations for them.

2.7 Evaluation

We evaluated the efficacy of Anteater's framework through a preliminary user study, a comparative study and a series ofusage scenarios.

2.7.1 Preliminary Pair Analytics User Study

User affordances offered by and the development status of a visualization prototype are key factors to steer the design of a user evaluation study [40]. In the case of Anteater, we do not intend to validate the scalability or usability of its interface and architecture (see Discussion). Similarly, we do not evaluate users ability to complete the tasks defined earlier using Anteater. Rather, we found it more appropriate to validate Anteater's visualization first approach to debugging and the exploration processes that Anteater facilitates. In particular, we wanted to observe the use and utility of global views of program values offered by Anteater in the program exploration process. Hence, we chose *pair analytics* [6] an appropriate user evaluation protocol.

Pair analytics offers a "think-aloud" protocol that helps generate verbal data by capturing the natural interaction between study participants and the proctor using the visualization interface as a communication anchor. Using the pair analytics method, a team is formed between a study proctor (or a visualization expert) who helps navigate Anteater and a subject matter expert who drives the exploration/debugging efforts.

We chose this evaluation over other methods for multiple reasons. First, this approach allows the subject matter expert to focus less on the nuances of the visualization interface (e.g., interaction types, loading data, etc) and more on exploration and question-answering processes. Other methods require participants to thoroughly learn an entirely new system before completing any tasks. The overhead of learning the nuances of a new system requires a significantly longer study session. Additionally, Anteater is a prototype implementation. Having a proctor to assist in the navigation

of the tool provides immediate assistance on how to proceed in the event that a system problem arises in the prototype. Second, a comparative study where experienced programmers complete debugging tasks with Anteater as well as with existing methods not only requires a significant overhead for learning the new system but also must mitigate the bias introduced by participants predisposition towards their current debugging practices. We discuss this more in the next study.

The exploratory nature of this study combined with the pair analytics protocol allows us to mitigate the bias of a participants predisposition to their current practices and reduce the overhead of learning and using a prototype system while still evaluating the utility of Anteater in exploratory debugging/understanding tasks.

Methodology

Participants Participants were recruited from a graduate level "Principles of Machine Learning" course. All participants are actively involved in computer science research, use Python as their primary programming language, and consider themselves experts in Python. We believe that the debugging and understanding tasks of programs written by graduate students in an upper level machine learning course or their research are comparable to those in real world data analysis programs.

We recruited a total of 5 participants from the class, which had a total of 20 students. However, only 3 of the studies were carried out to completion. We discarded one of these studies because the participant provided a program with a known bug that they thought might be interesting to re-discover with Anteater. While the subject matter expert's program was appropriate for the user study, we thought the prior knowledge of the participant would bias the study's outcome. As a result, we promoted this program to a usage scenario and discuss it in a later section. We discarded another study because the programs presented by the participant were not a good fit for the study. They do, however, highlight some of the limitations of Anteater and are discussed in more detail

later.

Study Session Process For each study, we recorded screen capture data along with audio recordings of each interview. Participants were asked to bring their own program to the study. All participants brought a program that performs some form of data analysis. Allowing them to choose their program helped alleviated some of the mental overhead of the study by not requiring them to learn a new program, along with a new debugging tool. Furthermore, this kept participants in their domain which enabled them to perform more meaningful explorations of their programs with Anteater.

Participants engaged in two sessions with the proctor. Due to the current policies in place in the U.S. at the time the study was conducted, all sessions were held online rather than in person, as would typically be done. Anteater's primary developer served as the study proctor to assist participants with navigating the nuances of Anteater and prompting them with questions to describe their exploration process.

The first session, was a brief meeting to introduce the participant to Anteater and discuss the participants program. The proctor and participants discussed what the participant wanted to see from within their program. After the first session, the proctor ensured that the program was suitable for the study and that the participant could view what they desired by testing it in Anteater. If the program was suitable, participants were asked to meet for a subsequent session.

In the second session, the proctor walked participants through the various features of Anteater. Afterwards, participants began exploring their programs with Anteater. The study gave participants free reign of their exploration, they were not given specific tasks to accomplish. In doing this, their behavior with Anteater exemplified more precisely how they would use a system like Anteater in their actual program debugging and understanding practices.

During the second session, the proctor served two primary purposes. First, to mit-

igate the overhead of learning a new system, the proctor assisted participants in navigating the features of the tool. Prompted by verbal cues from participants, the proctor would remind participants how to accomplish tasks within Anteater. Second, akin to other pair analytics evaluation studies, the proctor freely asked questions to promote exploratory thinking. In effect, participants' answering of such questions helped distill internal cognitive processes that were qualitatively analyzed.

For the participants who completed the study, the second session lasted between 60 and 90 minutes. Approximately the first 30 minutes of each session was spent introducing the subject matter experts to Anteater and getting Anteater set up to run properly on their machines.

Results

From this study, we found that, even in its imperfect prototype state, Anteater was useful to participants for debugging and achieving a better understanding of their programs. All participants were able to learn something new about their program that they previously had not understood. For the sake of confidentiality, we cannot give specifics about the programs used by participants. However, we try to give some context in the form of general concepts found in data analysis programs.

The first participant (P1) knew a bug existed in their program causing it to run incorrectly, but had yet to find it. With the proctor's guidance, P1 leveraged Anteater to identify and fix the bug (which aligns with G2 above). Through the use of the timeline plot and the ability to track custom expressions on more complex data structures (which corresponds to T1 - track a variable or expression), P1 found the bug, fixed it, and then verified that the revised program ran properly. During the exploration process, P1 discovered that there was something unusual about the training dataset, denoted as the *whole* dataset, which is split into *left* and *right*, vital to the proper execution of the program. P1 correctly noticed the problem since "the right dataset and the [whole]

dataset cannot be the same" even though the scatter plot showed them as identical (T5 - observe relationships between values). Upon further investigation of the captured values in each dataset, P1 explained that the "right dataset ... points to [the] class dataset" which causes them to overwrite the whole training set with only the *right* one. After modifying the code, a new trace was run and P1 validated the proper behavior of the code. After being asked if they were "able to gain new insight into [their] program using Anteater," P1 answered that "the scatter graph and also the tracking values [were] very helpful."

The other two participants (P2 and P3) presented more open-ended cases. P2 and P3 did not have known bugs, but rather non-trivial data analysis programs whose execution was not fully understood (which corresponds to G3). In both cases, the timeline view of certain variables over time was crucial. P2 heavily relied on the timeline and filtering capabilities of Anteater to verify that their program was converging as expected. P2 also used the timeline and filtering feature to inspect if their program was reaching the extremes of its search space. Using the visualizations provided by Anteater, P2 discovered that the program did not search the entire space in one direction and searched beyond the bounds in the other direction. After completing their exploration, P2 commented that understanding "why the values are so far off from the [search space] is a good next thing to look at."

P3 also heavily relied on the timeline view. They used it to understand the behavior of a set of weights in their analysis program. Before their use of Anteater, P3 had little idea of how the weights behaved throughout their program's execution. Anteater allowed them to track and visualize the weights over time to see how they evolved as the program ran. After they inspected the weights, the participant commented that "[Anteater] completely helped [them] understand sort of the underlying domain thing of what was going on with the weights." P3 further explained that Anteater was able to show that the program "is converging on one particular feature as an important weight

and the rest [are seen as] super unimportant." Through the use of Anteater, P3 was able to understand the behavior of the weights relative to the domain for which the program addressed, specifically through the use of the visualizations. The ability to visualize the variables over time was key to P3 understanding this behavior.

We believe our observations in this preliminary study provide promising evidence towards the utility of a visualization first approach to exploratory program debugging and understanding. P2 and P3 both performed exploratory tasks for understanding their programs and heavily relied on the global plots of values from within their program and interactions with those plots to improve their understanding of the programs behavior. In a post exploration interview, all participants indicate that they were able to gain new insight into their programs: P1 by finding their bug and P2 and P3 through understanding the behavior of certain values. Similarly, all participants expressed that, if a polished and optimized version were available, they would like to use a system like Anteater for future programming tasks.

As mentioned earlier, we discarded one evaluation study, because the programs provided were not ideally suited for the objective of the evaluation. The participant initially brought a large, machine learning program that took approximately a week to run. This program was not a good fit since we do not aim to study the interaction between trace size and applicability of our approach, but rather the utility of our approach to real Python programmers. A program that takes a week to run will generate a trace too large to handle by the current implementation of Anteater. Admittedly, this does limit the generalizability of Anteater, but we consider the study of how to scale a system like Anteater for future work. The participant then provided several small-scale programs that we were also not ideally fit for the study. The first of the programs was a small multi-threaded program. However, Anteater does not currently support multithreading. The other two programs were linear programs (no loops) with only 20-40 function calls and variable assignments where the code was broken into several small,

independent parts. These programs are similar to those found in an introductory CS course. This study aimed to use programs similar to those that would be found in a real data analysis setting. The small-scale programs were simply not sufficiently realistic for the study. As a result, rather than asking them to provide additional programs, we omitted their case from the study.

Threats to Validity

Because our sample size was small, the results, while promising, can only be considered preliminary. We designed the study to keep participants in their domains as much as possible to preserve the ecological validity of the study. In doing so, we could get deeper insights into users exploration processes and the results would better reflect real world utility of Anteater's design. However, a full scale study to further validate the design and utility of Anteater needs to be conducted in future work.

In addition, the pair analytics protocol could potentially introduce bias into the study if the proctor becomes too heavy handed in driving the exploration. In this study, the goal of the proctor is not to drive the exploration, but rather to aid the user in understanding the nuances of the system. Their primary role was to observe the participants as they explored their programs, point them to the features of Anteater that would help them answer their questions, and prompt them with additional questions to provoke thought about the findings presented by Anteater.

2.7.2 Comparative Evaluation with an IDE

Our preliminary study aimed at evaluating how programmers use Anteater in their exploratory debugging and understanding tasks. We found that the approach of providing global views of program values at the forefront was useful for our participants in completing exploratory program understanding tasks. In a second study, we attempt to compare how participants debug with Anteater vs a more traditional IDE debugger.

In this study, we asked people to complete two debugging tasks, one with Anteater and one with an IDE. We then prompted them with questions about their experiences. The goal of this study was not to evaluate how efficiently or accurately participants debugged programs. Rather, we aimed to evaluate how people interact with each system and contrast their experiences. However, over the course of the study we encountered several challenges to conducting an evaluation of this type. These will be discussed more in the following sections.

Methodology

Using recent technology that brings the Python library into the browser [88], we developed a version of Anteater that runs entirely in the browser and does not require a python server back-end as in the prior version. Thus, in contrast to our preliminary study, participants were able to complete this study entirely in the browser, without any assistance from a proctor or system expert. Participants were asked to complete 2 debugging tasks: one with Anteater and one with an online Python IDE.

Participants Throughout the course of the study, we conducted two rounds of recruiting participants.

The first round of recruiting was conducted by advertising the study to relevant groups of research and data analysis professionals. As an incentive for completing the study, participants were entered into a drawing for a $100 Amazon giftcard. We chose this method of recruitment because we believed that these groups of professionals would have the experience to provide focused and meaningful feedback. We kept the study open for 4 weeks. However, we were only able to recruit 4 participants in this first round. As a result we conducted a second round of recruitment.

In the second round of recruiting, we recruited participants through the recruitment service Prolific. To ensure that our participants had the proper experience, Prolifc al-

lowed us to specify that participants must have programming experience. Additionally, in the description of the study, we specified that they must have Python programming experience. However, despite this specification, several participants indicated that they were novices with Python, which may have impacted their ability to complete the given debugging tasks. These participants tended to provide the least meaningful responses. Through prolific, we recruited 9 participants. Participants were paid $10/hour to complete the study.

Of the 13 participants, 4 rated themselves as Python novices (have never used Python before or are currently learning it), 6 as intermediate (use Python sometimes, but not as a primary language), and 4 as experts (use Python regularly as a primary language).

Participants were asked to provide the purpose of their primary programming activities. 4 of the participants indicated that they primarily program for coursework, 7 indicated that the program for software development and 2 indicated that they program for data science/analysis.

Participants were also asked about their current debugging practices. 8 participants indicated that print statements were part of their debugging process, 8 use breakpoint debuggers, and 6 participants use a mix of the two. 3 participants did not provide descriptive responses.

Study Setup During recruitment, we gave participants a link to a webpage describing the study purpose and format. If participants chose to start the study, they were taken to a Google form to where they were they were shown an instructional video on how to use Anteater and then given two debugging tasks to complete, one with Anteater and one with an IDE. Participants were asked to debug the program either until they found the bug or until 10 minutes lapsed. We wanted participants to try to debug the program but, in the event that they could not find the bug, we did not want to ask

them to spend more than 10 minutes trying. After each debugging task, participants were asked a series of questions about their experiences. Upon completing both tasks, they were asked a series of questions to compare their experiences and evaluate the design of Anteater.

Debugging Tasks The study asked participants to complete two debugging tasks. Both tasks involved computing the number of polling places per capita for every state in the US from a dataset containing the population and number of polling places for each county in the country. One of the debugging tasks (D1) generated erroneous (negative) values due to an off by one error when indexing a list that separated each state in the data by inserting a value of -999. The second debugging task (D1) generated similarly erroneous values due to an unclean data file that represented missing values with -999. Participants were presented the two tasks in random order and each task was randomly paired with either Anteater or the IDE.

Results

Overall, the reception of Anteater was generally positive, especially since this was participants first exposure to the system.

Of the 13 participants, using either tool, only 5 were able to confidently find D1 (4 were unsure and 5 did not believe they found it) and only 2 were able to confidently find D2 (7 were unsure and 4 did not believe they found it). This indicates that the bugs we introduced, while seemingly simple, were likely too complex for this type of study. This exemplifies the first challenge we encountered in this study: introducing sufficiently small but realistic and meaningful bugs that are identifiable in a short period of time. We carefully developed these bugs and tested them in a pilot study to ensure that they were not too complex. However, despite this, it seems our bugs were still to complex.

Additionally, several participants cited that the recommended time limit of 10 min-

utes for each task was not enough to learn Anteater and complete the debugging task. One participant noted *"I felt like anteater was aiming towards a feature I would find useful, but also only having 10 minutes of time with it, it certainly wasn't effortless to figure out and I suspect I missed some possible chances to take better advantage of it."*. This highlights the second challenge we encountered: overcoming the overhead of teaching participants an entirely new system while still keeping the study a reasonable length. Most participants took at least an hour to complete the study with the 10 minute time limit. Given the offered compensation, we did not feel that would could ask for more time than that. However, it seems that this was not enough time for participants to learn the nuances of the system.

After completing their debugging tasks, around half of the participants (7 out of 13) indicated that they would prefer a production ready version of a system like Anteater over an IDE. One participant cited *"Anteater seems like it would be more effective for the targeted debugging style that I tend to follow, i.e. focus on a few specific variables or expressions, and investigate them deeply (and being able to do it visually isn't something supported in anything I've seen!), as opposed to the way the IDE just gives me cluttered lists of numbers for everything, even if I already know they're not important."* Another participant cite that Anteater *"removes the careless of skipping the thing that we wanted to find (odd values)"* that happens when stepping in IDE debuggers.

Of the 6 who preferred traditional IDE's, 4 of them cited that they preferred the IDE and print statement debugging simply because they are accustomed to it. One participant cited *"I'm just more used to the traditional IDE, it's debugging paradigm makes sense in my head. Anteater introduces a whole new paradigm of debugging, which I could imagine to be useful, but I'm just not used to it"*. Another stated that Anteater *"provides a lot more information about what's happening with the problem, but I think it does not replace the phase of using convenient prints and proper testing to check what's exactly the problem."*. This exemplifies one of the largest challenges that we encountered

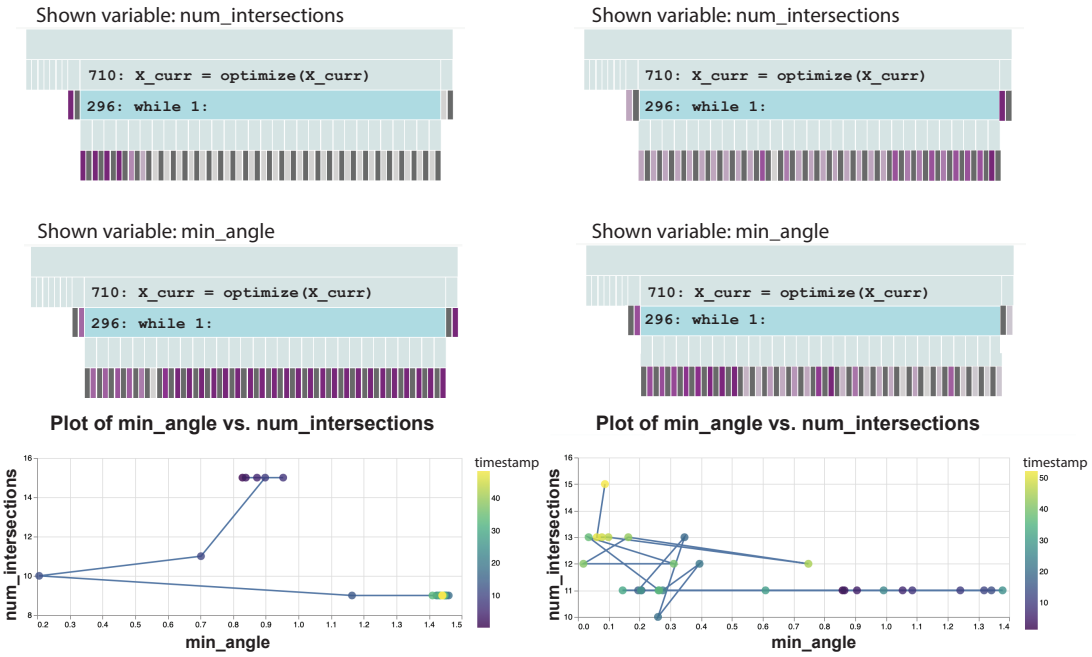

Figure 2.8: Using Anteater to compare two runs of gradient descent that should maximize the minimum crossing angle while minimizing edge crossings. The generalized context trees in (A) show that the number of intersections rapidly decreases (the color changes from dark purple to white) while the minimum angle increases. The scatterplot shows that the descent spends its first few steps at a bad solution and takes approximately three big steps before converging on a good solution. In contrast, in (B) the number of intersections increases throughout the descent while the minimum angle decreases. The scatterplot shows that, in general, as the number of intersections grows, the minimum angle shrinks and lands at a bad solution.

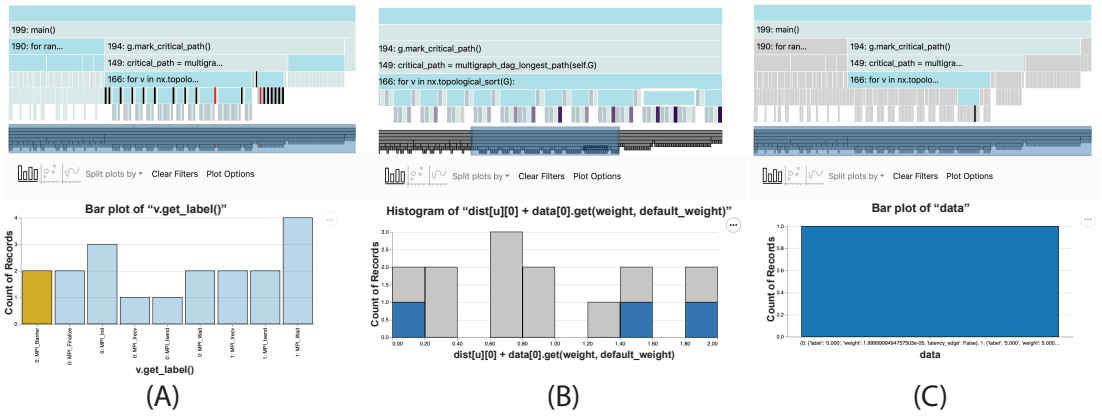

Figure 2.9: (A) shows how we can use the bar plot to find where the barrier node occurs in the execution plot. (B) shows the weights of the paths to the barrier node calculated in the program. We see that none of them are 5, as we expect from looking at the MPI call graph. (C) shows the data used to calculate the path weight that should include the edge of weight 5. We see that the data does include this edge but the algorithm does not look at it. For more details, refer to the supplemental video.

during this evaluation: overcoming experienced programmers predisposition to their current debugging practices. One participant strongly stated that *"I find prints the way of God"*. Several participants felt that Anteater would be useful but they did not want to change their current debugging practices. However, two of these participants commented that they think the features of Anteater would be useful if integrated into an IDE. One participant stated ''*The best combination would be to integrate Anteater in a classical IDE in order to have the best of both worlds"*. We found this a very encouraging comment for validating Anteater's visualization first approach to debugging and will explore this possibility in future work.

Threats to Validity

While the first round of targeted recruitment produced more meaningful results, there is the potential for bias due to the participants potential familiarity with the authors. Although the study was anonymous, we recruited participants from groups with which the authors are affiliated. It is possible that participants were more generous with their

responses than they would have been if they did not know the authors but we have no way of detecting this.

In the second round of recruiting, we were unable to properly screen participants through Prolific. In order to maintain the validity of the study, we did not modify the protocol when extending the recruitment to Prolific. However, it seems that the study would have benefited from additional efforts to screen participants Python and professional experience. In Prolific, we were only able to specify that participants have programming experience. However, Python experience was important for the study. One participant was unable to complete either task, citing that both tools were not accessible, possibly due to the fact that they were a novice in Python.

2.7.3 Usage Scenarios

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

Gradient Descent

The first usage scenario we present inspects a program performing gradient descent. This program was collected from a question on Stack Overflow [52]. The programmer struggled to figure out why the resulting values of the variables "x" and "$x1$" were NaNs. We will walk through how to use Anteater to understand the bug and correct it.

First, we run the program with Anteater to track one of the misbehaving variables, "x." Fig. 2.6-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 the plot type to "scatterplot" which shows a plot of the variable "x" over time, shown in Fig. 2.6-B. Now,

Program	# Tracked Variables	# Lines	# Recorded Calls	# Recorded Assignments	Original Execution Time (s)	Instrumented Execution Time (s)	Trace Size (MB)
Gradient Descent	5	36	402	801	0.0041	0.0691	0.3098
Longest Weighted Path	3	199	149	55	0.0023	0.0343	0.2351
Recursive Fibonacci	1	11	150,049	75,024	0.0292	5.8952	116.2

Table 2.3: Trace information and execution impact for traces of three of the programs discussed in this paper. The trace size and performance impact depends on the amount of information recorded in the trace

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. 2.6-C shows the resulting visualizations. We see that 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. 2.7, 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 see its behavior throughout the execution. In a traditional debugger, detecting this behavior requires stepping through several 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.

Graph Edge Crossing Angle Maximization

In this usage scenario, we investigate a program that tries to balance the number of edge crossings in a graph with the size of the minimum crossing angle. The program

searches for the layout that minimizes the number of edge crossings while maximizing the size of the minimum crossing angle. In this usage scenario, we inspect the stability of the gradient descent method on this problem.

To inspect the stability, we ran the gradient descent multiple times, tracking the minimum angle and number of intersections at each iteration of the gradient descent. We found that in most cases, the gradient descent returns a good solution, as demonstrated in Fig. 2.8-A, where it immediately begins moving toward a good solution and never turns back. However, instances occur, as shown in Fig. 2.8-B, where the gradient descent starts moving towards a bad solution, and never recovers. Therefore, we can conclude that although the majority of the time it produces a good solution, this method suffers from stability issues.

Longest Weighted Path Calculation

This usage scenario was presented to us by a prospective participant in the user study. While the program was not a good fit for the study, because the participant already knew where the bug was, it presents a good example of the utility of Anteater on real problems. This program aims to find the critical path, i.e., the longest weighted path, from the "Init" to "Finalize" nodes in an MPI call graph. It uses the networkx library to build a multiDAG and calculate the longest (weighted) path. We were given this program with the knowledge that this bug existed and which methods were affected but no other information on how to fix it. We then found and fixed the bug using only Anteater. Below, we explain how we found the bug.

To begin, we loaded the program and data files into Anteater. We know from inspecting the test graph manually that the algorithm overlooks one of the edges (of weight 5) from the "Init" node into the "Barrier" node. To build the longest path, the algorithm topologically sorts the nodes and iterates over them. For each node, it iterates over all of the predecessor nodes. To find the bug, we first need to find where the barrier

node occurs. We do this by collecting the node label in each iteration. Inspecting the node labels in the bar plot, as shown in Fig 2.9-A, shows us the point in the execution tree where the loop reaches the "barrier" node. Once we find the barrier node, we select it to view the other values in that specific iteration. We then switch variables to look at the path weights for each predecessor, as shown in Fig. 2.9-B. We see (from the x-axis of the barplot) that none of the path weights reach 5, which indicates that the algorithm misses the edge of weight 5 into the "Barrier" node. Next, we look at the data used to calculate the path weights. We notice that one of the "Init" predecessors has two weights associated with it, as shown in the filtered bar in Fig. 2.9-C. There are two keys in the dictionary, one for each edge from "Init" to "Barrier". Looking back at the algorithm, we see that it only looks at the first key which causes it to miss the edge of weight 5 and report an incorrect longest weighted path. To fix this, we simply find the edge with the highest weight over all of the edges from the predecessor to current node.

2.8 Discussion and Limitations

Omnicode vs. Anteater While Omnicode and Anteater both intend to help programmers debug and understand their programs, the two systems differ in their target audience. Omnicode aims to help novice users create mental-models to reason about their program's execution and debug unexpected behavior. The size and complexity of programs it needs to support for this audience is quite small. Thus, Omnicode only supports programs of around 10 variables and 100 execution steps. Anteater aims to help programmers in general. Therefore it needs to support different types of programs.

While Anteater cannot support large scale software-systems as they produce an unmanageable amount of data, it can support much larger programs than those written by novices, such as those programs written by data scientists. Anteater's ability to support

a program largely depends on the number of function calls and variable assignments that are recorded in the trace. We do not know the exact limits of these parameters (as they are very interdependent) but know that Anteater certainly supports programs with up to 225,000 function calls and variable assignments. Each function calls and variable assignment is comparable an execution step as described by Omnicode. Thus, in contrast to Omnicode which supports programs of around 100 execution steps, Anteater can support programs of at least 225,000 comparable steps.

Most of the differences between Omnicode and Anteater stem from the fact that they are geared toward different audiences. Omnicode supports a live programming environment because it targets small programs whereas a static environment makes more sense for Anteater. Similarly, Omnicode tracks every variable in the program which is infeasible for the larger programs Anteater supports.

Goal 3 As mentioned earlier, G3's serves as a catch all for debugging tasks that do not fit into the first two goals. We acknowledge that a different, more specific, version of G3 may exist that, when evaluated, would allow us to learn more specific information about how programmers use Anteater. However, keeping this goal general allowed us to support any exploratory tasks presented to us by potential participants in the preliminary study. This study focused more on observing how people would use Anteater to validate its design rather than evaluating their ability to complete tasks with Anteater.

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 to the programmer. 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. The problem of automatically suggesting variables and efficiently

tracing all program variables remains for future work. One possible approach could be to still have the programmer specify variables, but then automatically collect all values that the variables depend on.

Limitations Anteater will not scale to programs the generate 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. Table 2.3 shows the performance impact for several programs traced with Anteater. Research exists on collecting the entire trace of large programs [85]; future work is needed to evaluate if Anteater works well with this method. We note that our visualizations operate on relational data, and there is a growing number of techniques to support interactive visualizations on large relational datasets [50], [79]. A full investigation of their impact on program visualization, however, is out of present scope. 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. Finally, Anteater assumes a sequential programming model and does not support parallel programs. Work exists in automatic tracing of parallel programs in the traditional sense (without values) but applying and extending these traces to Anteater is left for future work.

Chapter 3

Trace Diffing

3.1 Introduction

In the previous chapter, we introduced Anteater. Anteater supports the exploratory debugging of single instances of a program's execution. However, people commonly perform comparative debugging tasks where they run the program, evaluate the program execution values, make a small change, re-run the program, and re-evaluate the execution values to understand the effects of the change on the program. ProgDiff extends Anteater to support the comparison of two program executions.

Past research on debugging strategies and processes identifies strategies that employ this comparative process **grigoreanu2009males**, [44], [117]. For example, a study of debugging processes found that people often use comparative debugging methods at multiple stages of the debugging including "determining the problem" and "repairing the error" [117] . In "determining the problem", people use it to compare correct outputs with incorrect outputs. When "repairing the error", people use it to confirm the effects of their repair.

This type of program inspection is not limited to traditional debugging tasks but also applies to more exploratory tasks. Consider a program that requires people to set hyper-parameters that define how the analysis runs. Often, these programs provide little guidance on choosing these hyper-parameters and people often use trial and error to determine a "good" value. As a result, people often desire to compare the results of different parameter settings. Typically, a person guesses a parameter value, runs the program, and inspects the results. After inspecting the results, they adjust the parameter, re-run the program and inspect the effects on the results. They often repeat this process many times until they achieve the desired results.

Current practices do not adequately support comparative debugging and under-

standing tasks. While comparison of final results may seem simple enough, not all comparative debugging tasks operate solely on final results and, in a program where no "correct" output exists, viewing the results of a single execution in isolation does not necessarily illustrate their quality or improvement over past results. When debugging with print statements, people must save the previous version of the text output and manually compare textual values with the output of the current version. Side-by-side comparison of individual values can even give a misleading impression of the differences in the overall trends of the data. Step-through debugging provides an even greater challenge for comparative debugging practices, as it is difficult to save the previous state without running simultaneous step through debugger instances, one on the past version and one on the new version. Similarly, Anteater also requires two instances to view both traces simultaneously. As a result, debugging methods need to support the direct comparison of consecutive program executions.

ProgDiff creates comparative visualizations to illustrate the effects of source code changes on program values. It extends the tracing infrastructure of Anteater to find the differences between two consecutive program traces. Rather than requiring people to contrast two instances of their debugging methods, ProgDiff provides a single view to illustrate the effects of source code change.

3.2 Related Work

Source Code Diffing Several tools exists for diffing source code. The Unix diff utility and Git diff command support the diffing of two textual source files by finding the minimum number of line additions and removals to produce the new file from the old one . This method provides very course grained differences in source code files. Canfora et. al. [21] expand on the Unix diff utility to provide more fine grained diffing of textual source files that supports changes and moves, as well as additions and deletions.

Horwitz [58] expands their definition of diffing to differentiate between semantic and textual changes when identifying changes to the source code. Semantic Diff [61] provides a semantic report of the differences between two versions of a procedure through the inspection of input-output behavior of the procedure.

Rather than diffing text source files, other methods first parse source code into abstract syntax trees (AST's) and then find diff the resulting trees **dotzler2016move**, [42], [46], [55]. Similar to textual diffing methods, these methods look for the minimum number of edits to move from the original AST to the new one, where an edit can be an addition, removal, change, or move. These methods build off of Chawathe et al.'s algorithm for detecting change in hierarchically structured information [28] by leveraging structure and information specific to AST's to more accurately identify differences. As part of ProgDiff, we use gumtree [42] for source code differencing to highlight the textual program changes. However, ProgDiffrequires additional methods for diffing the resulting traces (discussed more in Section 3.4.1).

Trace Diffing and Comparison Suzuki et al.'s TraceDiff [109] relates most closely to ProgDiff. TraceDiff provides automatic feedback and hints for students completing introductory programming assignments. It uses program traces to illustrate differences between an incorrect version of a program and a synthesized correct version. Leveraging these differences, it provides hints to guide students through their mistakes. While similar to ProgDiff, this tool focuses on assisting students in guiding students in their assignments and utilizes minimal visualization.

Taheri et al.'s DiffTrace [110] provides diffs of program traces for debugging high-performance computing code. It collects whole program function call traces per process/thread and uses concept lattices to build the traces and identify the differences between a normal trace and a fault laden trace. In contrast, ProgDiff operates only on sequential programs and presents differences in program values, such as variables and

expressions, in addition to calling structure.

Other methods exist for comparing traces without formally diffing them. Miranksyy et. al. use trace comparison to find sources of errors in software systems [78]. They take an existing trace with correct behavior and compare it against a trace in which the software misbehaves to calculate the entropy between them.

Visual Comparison of Traces Cornelissen and Moonen create visualizations to highlight repetitive behavior and execution phases in a single program identified by matching the trace to itself [34]. Intel's Trace Analyzer provides timeline visualizations to compare traces of MPI applications [1]. Trümper et. al. create multiscale visualizations for comparing large traces using icicle plots and edge bundles [115]. Voigt et al. [118] create trace visualizations of method calls and object accesses for large scale traces. While all of these tools use visualization for comparing traces, they all address a specific type of problem: detecting similarities inside a single program, comparing executions of MPI programs, and comparing executions of large scale programs. All of these problems differ from the programs ProgDiff supports: understanding the impact of small changes in general, medium-scale, single-threaded programs.

Comparative Visualization Munzner's framework for visual design identifies comparison as a low-level user goal when analyzing data. [81]. Many visualization applications support this goal in volume rendering **woodring2006multi**, and more areas TBD . While there exist many applications supporting comparison, there exist few taxonomies and frameworks for creating comparative visualizations. Pagendarm and Post describe approaches for comparative visualization in images **pagendarm1995comparative**. Graham and Kennedy surveyed methods for visualizing (and thus comparing) multiple trees **graham2010survey**. In 2011, Gleicher et al. **gleicher2011visual** developed a taxonomy for designing comparative visualizations for information visualization. This taxonomy remains as the primary taxonomy for designing comparative visualizations.

It defines three types of comparative visualization: juxtaposition, superposition and explicit encoding. We employ this taxonomy in ProgDiff's visual design and discuss these types in more detail in Section 3.4.2.

3.3 Classification of Program Changes

Several works exist on classifying types of changes to programs, for several types of program representations.

Purushothaman and Perry classify changes based on the textual changes to the source code [87]. They present 4 types of changes: (1) modifications to existing lines, (2) insertions of new statements between existing lines, (3) deletions of existing lines, and (4) modifications of lines accompanied by an insertion and/or deletion of lines. The fourth type combines the first three types.

Fluri and Gall [45] define a similar classification for changes to the AST. They describe four program modification operations on an AST: insert a new leaf node, delete a node from its parent, move a node to a new parent, update the value of an existing node. Additionally, they provide several higher level changes for object oriented programs stem from one or more of the base modification operations. We use this classification when describing the types of source code changes.

Lehnert et al. [71] have a similar taxonomy for classifying change, based on that of Fluri and Gall [45]. Rather than classifying changes to AST's, they define software as a graph where nodes are artifacts such as UML diagrams or C++ classes and edges are dependencies between the artifacts. Lehnert et al. present two tiers of changes: atomic and composite. Atomic changes include the addition, deletion, and modification of both nodes and edges. Composite changes require multiple atomic changes and include: move, replace, split, merge, swap.

While all of these classifications operate on slightly different program representa-

tions, there seems to be a core set of operations that for classifying change: add, update, and delete, with more complex changes consisting of combinations of these operations. We use these core change types to classify the changes in both the source code and the traces. We differentiate between changes in the source and changes in the trace because one type of change in the source may cause a different type of change in the trace (or no change at all). In the remainder of this section, we discuss in more detail each type and how ProgDiff supports it.

3.3.1 Source Code Changes

Fluri and Gall's classification of change types includes a variety of higher level changes that depend on additions, deletions and updates. ProgDiff only supports a subset of these changes, primarily those that modify the execution and functionality of a program.

Addition Fluri and Gall present a variety of changes that rely on the addition operation, including additional functionality, statement insert, parameter insert, and else-part insert [45] . ProgDiff inherently supports all of these when diffing the source, however some of them may not directly appear in the trace and each type may cause updates, additions or have no effect in the trace (discussed in the next section). For example, the addition of a new parameter to a function will appear in a source code diff, however unless that parameter affects the calculation of a value or the execution of a function call or loop, the change will not appear in the trace.

Deletion Fluri and Gall present a complementary set of changes that rely on the deletion operation, including removed functionality, statement deletion, parameter deletion, and else-part deletion. Again, ProgDiff supports all of these when diffing the source but they may not directly appear in the trace. Deletions to the source may propagate to the trace as deletions, updates, or not at all.

Update The majority of update changes that ProgDiff supports fall under the "statement update" type (e.g. updated parameter/variable values) with the rest falling under "condition expression change". The remaining type These modifications do not explicitly show up in the diffed trace but may result in changes that show up in the trace. Thus, source code updates may result in any type of trace change.

Unsupported changes Some changes defined by Fluri and Gall do not have any affect on the trace and, as a result, ProgDiff does not support them. The renaming of any program components, e.g. parameters, functions, variables, etc., will not have an effect on the behavior of the program. The source code diff highlights this change but, in the back-end, the originally named component still maps to the newly named component and they are considered the same when running the trace diffing algorithm.

Additionally, we do not support changes to the accessibility of a component (e.g. a private or public variable), the type of variables, modifications to object state, or changes to the inheritance structure of classes. While these changes have significant affects in object oriented programs, they do not inherently have a significant effects in Python programs or on execution traces.

Last, we do not precisely support modifications that alter the depth of components. For example, if, in the original program, function A calls function B and we modify the program so that function C calls function A calls which then calls function B, the call to A is now at a new depth and will not be matched with the call to A in the original trace. ProgDiff will mark the calls to A and B in the original trace as deleted and the calls to A and B in the new trace as added. We show an example of this in Fig. 3.1.

3.3.2 Trace Changes

Addition Additions to the trace result from a variety of source changes including: added function calls, increased iterations in a loop, update of conditional statement,

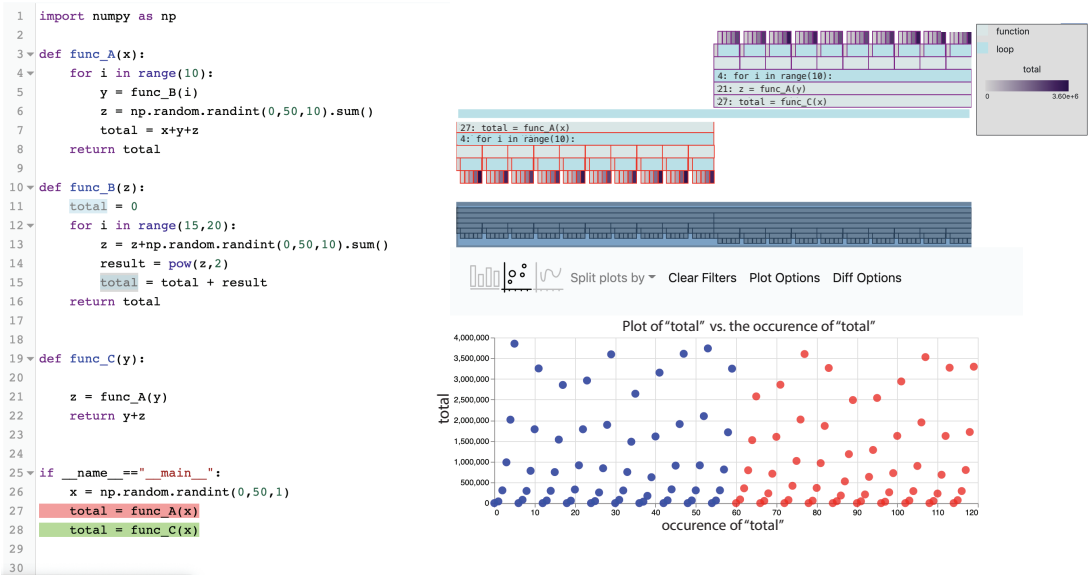

Figure 3.1: The execution diff of a program after we wrap the initial function call to func_A in a call to func_C. Doing this changes the depth of the call to func_A, causing this call and all subsequent children to be marked as additions.

etc. We define an addition to a trace as any node (function call, loop, assignment, etc.) that does not have a corresponding node at the same depth in the previous execution. Note, if a function call exists in the prior version but the call appears at a different depth or from a different parent than in the original execution, it will be marked as an addition.

Deletion Deletions in the trace result from a complementary set of source changes to those that cause additions in the trace including: a removed function call, fewer loop iterations, etc. Consequently, our definition of a deletion complements that of an addition. We define a deletion in the trace as any node in the original execution that does not have a corresponding node at the same depth in new version of the trace.

Update In traces, we limit the definition of update to only include updated variable/expression values. When two matched instances of a variable differ in value, we

mark that value as updated. These updated values stem from program changes such as the modification of a calculation statement or parameter, the insertion/deletion of a calculation step or simply the use of a different randomly generated value.

We do not consider "updates" to function calls or loops in our traces. A clear definition of what it means to update a function call does not exist in this setting. Likewise, the definition of "updating" a loop could mean a variety of things, such as changing the number of iterations, changing the values iterated over, etc. However, those updates would likely spawn subsequent trace changes that our definitions easily encompass. .

3.4 ProgDiff's Design

3.4.1 Source and Trace Diffing

ProgDiff incorporates two forms of program diffing: diffing the source code and diffing the resulting trace. In this section, we discuss how ProgDiff performs each of these diffs.

Source Diffing As previously mentioned, ProgDiff uses Gumtree [42] to create the source code diff by diffing the AST's of the original and modified version of the program. This allows ProgDiff to highlight the textual differences between the two versions of the source code.

Because ProgDiff already parses and traverses the AST to transform the source code to generate the trace, it may seem that the AST diff would provide all necessary information for diffing the traces. It already identifies any new, changed, or deleted nodes in the AST so we would only need to mark occurrences of those nodes in the resulting trace. However, two primary problems arise when only diffing the AST's.

First, not all changes in a trace lead result from changes to the corresponding node

in the AST. For example, consider a variable that sets how many iterations a loop runs. Changing the value of that specific variable will cause the loop to execute more or fewer times. However, the loop node of the AST will remain unchanged. Thus, simply diffing the AST's and marking the changed nodes would not mark the loop nodes as new/changed, just the iteration variable. Thus, the AST diff to mark changes in the trace would not mark all changes.

Second, in order to use the AST diff to match the old trace with the new trace, we would have to re-run the original trace to mark the nodes that we know will change or be deleted, thus allowing us to match them with the nodes in the new trace. However, we do not want to re-run the original trace because if the program contains any randomness, this may change the tracked values from the original trace. As a result, AST diffing would only allow us to mark nodes in the new trace and would not provide means to link the two traces together. Thus, while AST diffing makes sense for identifying the differences in the two versions of the source code, we need an additional algorithm for diffing the resulting traces.

Trace Diffing Execution traces inherently have a tree structure, where each child node was executed from within the parent node. Because of this, to identify the differences in the traces, we can use a tree diffing algorithm. ProgDiff employs a basic tree diffing algorithm, adapted from [20]. The algorithm operates as follows. First, the algorithm takes in two traces (V1 and V2) and traverses them simultaneously. Starting at their roots, it inspects gather the children of both nodes and matches the children of the V1 node to the children of the V2 node as possible. We consider two nodes to be a match if they are of the same type and name (e.g. they are both variable assignments and they assign to the same variable). We consider matching nodes to be "unchanged" with the possible exception of variables and expressions. For variable and expression nodes, we also consider their recorded values. If the nodes match in type and name

but not value, we identify them as "updated", rather than "unchanged". We identify the reaming nodes in V1 as "deleted", because they do not correspond to any nodes in V2, and the remaining nodes in V2 as "added" for the same reason. After pairing the children together the algorithm recurses on each pair and repeats this process. The algorithm also recurses on the children of the unpaired nodes to classify all of their children as "added" or "deleted" as well. For a pair of matched nodes, once their children have all been recursed on, the two nodes are combined into one and added to a combined trace. Unmatched nodes are added to the combined trace with empty attributes as placeholders for their would-be match. Algorithm 1 presents psuedocode of the algorithm.

While this algorithm provides informative diffs for many types of changes, it does not fully leverage all of the information specific to program traces. For example, in the Fibonacci program in Figure 3.2 , we know that execution tree and values from the original program (where we evaluated the 10th Fibonacci number) are a subtree to of the evaluation of the new version (where we evaluate the 12th fibonacci number). As a result, an ideal diffing algorithm would identify the recursive nature of this program and identify that the original execution is just a subtree of the new one and mark the parent nodes as added rather than marking leaves as new. As such, more sophisticated algorithms need to be explored and can easily be swapped in. We discuss this more in Section 3.6.

3.4.2 Comparative Visualizations

ProgDiff extends the visualizations in Anteater to support comparative visualizations and interactions. Gleicher et al. **gleicher2011visual** provides a taxonomy of different comparative visualizations. They present three types of comparative visualization for two datasets: juxtaposition, superposition, and explicit encoding of relationships. Juxtaposition provides separate but adjacent plots in the same space that allow people

Algorithm 1 Trace Diffing Algorithm: This algorithm describes how ProgDiff performs trace diffing. Note, the method MATCH_CHILDREN is described in Algorithm 2. The method COMBINE_NODES is trivial and thus is not described here.

Input: T1: The trace of the original program, T2: The trace of the modified program

Output: CT: A trace combining T1 and T2

```
1: function DIFF(T1,T2)
2:     new_children ← Empty List
3:     if T1 is unmatched (T2 is NULL) then
4:         Mark T1 as "deleted"
5:         for Each child N of T1 do
6:             childCT ←DIFF(N, NULL)
7:             Add childCT to newChildren
8:         end for
9:     else if T2 is unmatched (T1 is NULL) then
10:        Mark T2 as "added"
11:        for Each child N of T2 do
12:            childCT ← DIFF(NULL, N)
13:            Add childCT to newChildren
14:        end for
15:    else
16:        pairedChildren ←MATCH_CHILDREN(T1.children, T2.children)
17:        if The T1 & T2 are variables or expressions and the values differ then
18:            Mark the T1 & T2 as "updated"
19:        else
20:            Mark the T1 & T2 as "unchanged"
21:        end if
22:        for each pair in pairedChildren do
23:            childCT ← DIFF(pair[0], pair[1])
24:            Add childCT to newChildren
25:        end for
26:    end if
27:    CT ← COMBINE_NODES(T1,T2, newChildren)
28:    return CT
29: end function
```

Algorithm 2 Node Matching

Input: C1: list of children from T1, C2: list of children from T2

Output: pairedChildren: list of paired of nodes with the first node from C1 and the second from C2

```
1:  function MATCH_CHILDREN(C1, C2)
2:      pairedChildren ← Empty List
3:      for Each node C in C1 do
4:          pair ← [C,NULL]
5:          Look for matching node in C2
6:          if matching node is found then
7:              pair[1] ← matching node
8:          end if
9:          Add pair to pairedChildren
10:     end for
11:     for Each unmatched node C in V2Children do
12:         pair ← [NULL,C]
13:         Insert pair into pairedChildren
14:     end for
15:     return pairedChildren
16: end function
```

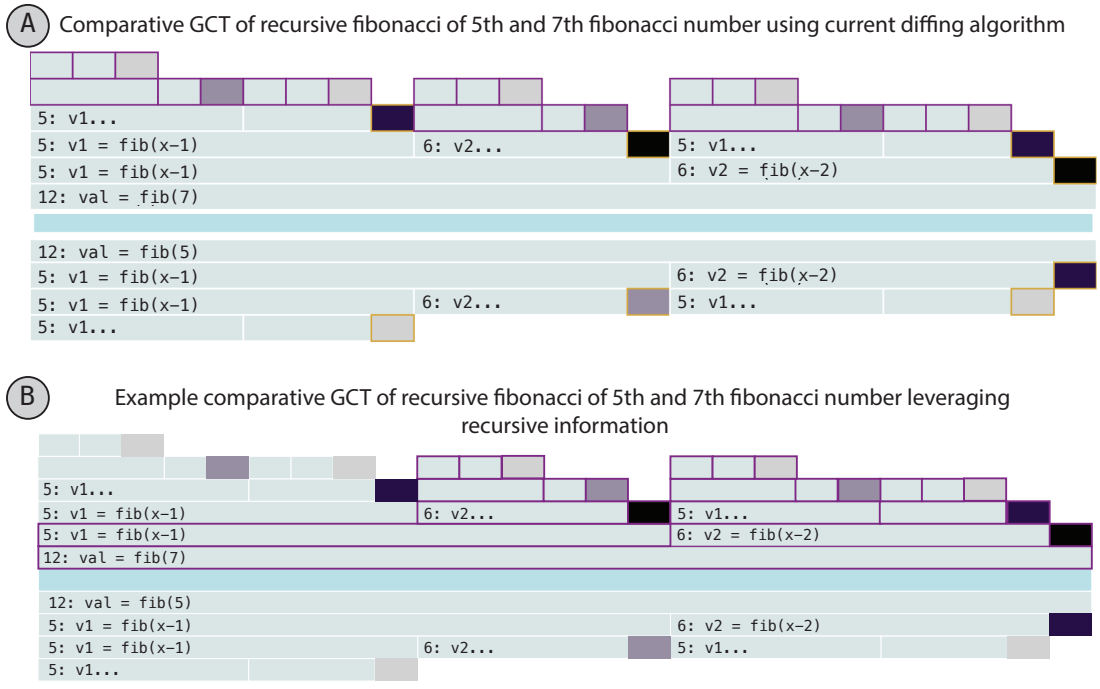

Figure 3.2: Caption

to view each individually as well as facilitate comparison of the two datasets. superposition combines the two dataset into a single plot, fully displaying both datasets in the same space. Explicit encoding directly encodes the relationship between the two datasets, rather than presenting them as two disjoint datasets. For each type of comparative visualization supported, ProgDiff provides a view using each of the three types with controls to toggle between them.

Generalized Context Tree The comparative generalized context tree (GCT) needs to show the two execution structures, while highlighting the differences between the two executions. When creating the comparative GCT we must ensure that corresponding parts of the execution align horizontally in the plot. The diffing algorithm combines the two traces into a single one. We combine each pair of matched nodes into a single node, maintaining the relative order for each trace. When creating the comparative GCT, we use this combined trace to generate the tree. We present three comparative GCT views based on Gleicher et al.'s taxonomy **gleicher2011visual**, two juxtaposition views and a superposition view.

The first juxtaposition view presents side by side GCT's, one for each trace. Each GCT uses the combined trace to create the structure. In the GCT for the original version, we draw all nodes that are updated, deleted, or unchanged. These nodes correspond to those that existed in the original trace. However, we leave gaps for nodes added in the second trace. This ensures that the two GCT's align correctly. We draw the GCT for the modified version in the same manner, leaving gaps for deletions from the first trace. While this view gives the entire view of both traces, it suffers when locating corresponding positions in the two plots.

The second juxtaposition view presents a single GCT visualization for both traces. The single view does not combine both traces, but rather, roots the GCT's at the same block and builds the original trace downwards and the trace from the modified version

upwards. Doing this instead of presenting two separate views enables faster comparison of the two traces. Again, we use the combined trace to build the GCT's, leaving gaps for additions and deletions as necessary. People no longer have to shift back and forth between two plots and instead only need to find nodes at the same horizontal position and vertical depth.

The superposition view presents a single GCT containing both traces. Unlike the second juxtaposition view, this view only builds downward. It builds the entire combined trace, highlighting the additions, deletions, and updates. While this view presents all of changes to the trace, it does not as easily illustrate the individual GCT's.

Histogram ProgDiffprovides three histogram views. When creating comparative histograms, we must ensure that the histogram bars align for both datasets. If the range of the data changes significantly, then keeping the original bars and extending the axis as necessary could result in an excessive number of bars. Instead, we bin both versions of the data together to create the bins and use those bins to create the comparative histograms.

We developed three comparative histogram views. First, the juxtaposition view places a histogram of each dataset adjacent to each other. We ensure that the bins and y-axis align in both plots to enable easier identification of similarities and differences. This view offers the advantage of allowing people to view the whole dataset, without interrupting visual clutter. However, it requires mental overhead of matching positions for comparison in two disjoint views.

The superposition view shows the two histograms side by side on the same axis. As shown in Fig. , there are two bars for each bin, one for the original version and one for the modified version. In contrast to the juxtaposition view, people can easily compare matching bars without mental overhead. However, the combined histogram interrupts the global view of each individual dataset, making it more difficult to get the entire

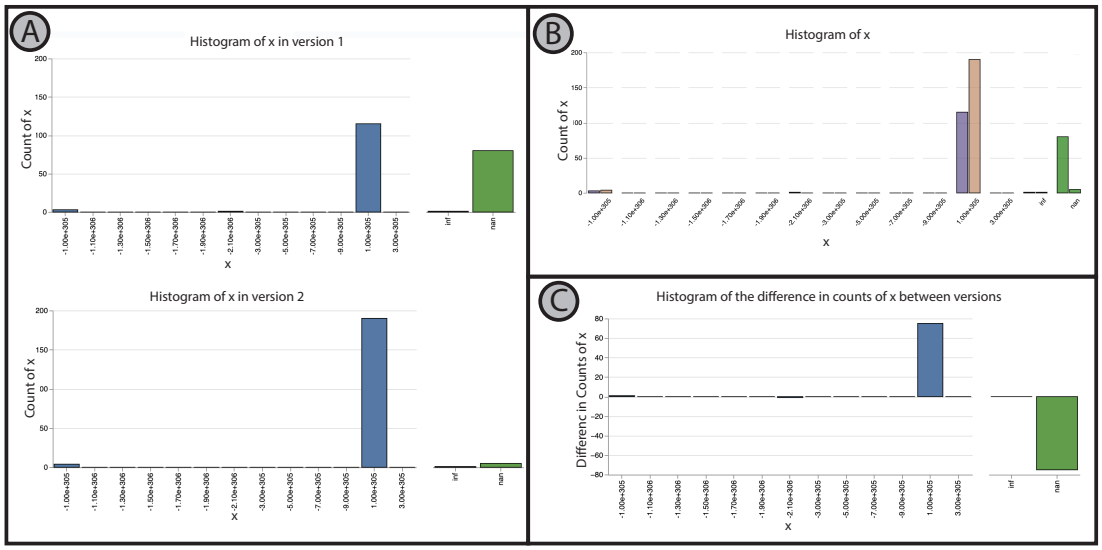

Figure 3.3: The comparative histogram views. (A) shows the juxtaposition view. The top plot shows the histogram from the original trace and the bottom plot shows the histogram from the trace of the modified program. (B) shows the superposition view. The left bars (colored purple for all quantitative values) represent the count from the original version and the right bar (colored orange) represent the count from the modified version of the program. (C) shows the explicit encoding of the difference of counts between the two traces. The count represents the count of the original version subtracted from the count of the modified version.

view.

Last, the explicit encoding view directly encodes the difference in the frequencies between the new trace and the original trace. Each bar represents the change from the original frequency to the new frequency. As a result, some differences may end up negative, requiring the histogram to account for negative values. This view strays the farthest from a traditional histogram, as values cannot have negative frequencies. This view allows people to easily and quickly see where the frequencies changed between the two versions. On the other hand, people lose the context of the actual distributions of the data involved.

Scatterplot ProgDiffalso provides three scatterplot views, complementary to those in the comparative histograms. For scatterplots we must ensure that all corresponding

instances are aligned on the x-axis. The diffing algorithm, described previously in Section 3.4.1 matches the instances from the new trace with those in the old when possible and marks the rest as new or deleted accordingly.

Much like the juxtaposition histogram view, the juxtaposition scatterplot view simply plots two adjacent scatterplots, one of each version of the data. While these give a clear view of each dataset, they require additional effort to match instances together and inspect individual differences, particularly if there exist unmatched instances. , .

The superposition view plots both sets of points on the same axis and colors the points depending on which version they belong to. ProgDiff plots matching instances, as identified during diffing, at the same x value. This plot allows easy comparison of matching instances across the traces by inspecting their horizontal positions. However, in some cases, it suffers from clutter and overlapping of values that do not significantly, making some comparisons difficult.

The explicit encoding view directly plots the difference between the two versions for each instance. For each paired instance, it calculates the difference between the new version and the original version, and plots that point. For points from the original version that do not have a matching instance, we simply treat their matching instance as 0. Thus for an unmatched value x from the original version, we plot it as $-x$. Similarly, we plot unmatched instances from the new version as their true value (i.e. if x is in the new trace and does not have a matching instance, we plot it as x). While this plot reduces the clutter fo the superposition plot, it loses the context of the actual values from the traces. To bring back some context, when someone selects a point we plot the corresponding points from both traces to give them context of the original values.

Unlike Anteater, ProgDiff currently only supports scatterplots with a single value, plotted by occurrence in the execution. Two variable scatterplots do not have an inherent ordering of the points, making it difficult to visually cue people towards matching pairs of points. Only the juxtaposition view, with accompanying interactions that link

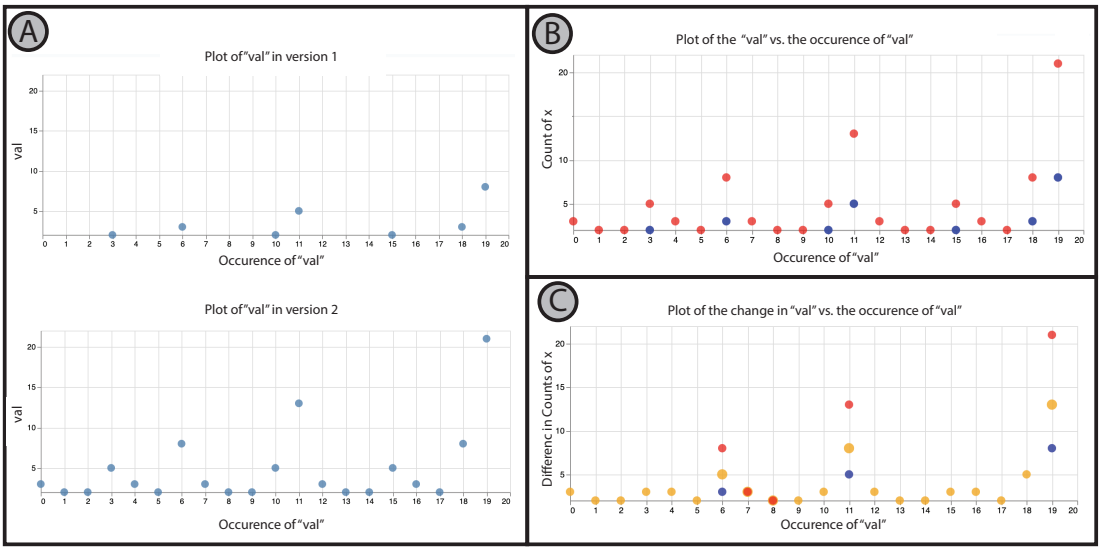

Figure 3.4: The comparative scatterplot views. The x-axis represents the occurrence (or instance) in the execution of the shown value. (A) shows the juxtaposition view. The top plot shows the scatterplot from the original trace and the bottom plot shows the scatterplot from the trace of the modified program. (B) shows the superposition view. We color points from the original version blue and points from the modified version red. Occurrences with only one point indicate an addition or deletion of an occurrence. (C) shows the explicit encoding of the difference between the occurrence in the modified execution and the original occurrence. Gold points represent the difference between the two occurrences. Selecting a point shows the actual values form both versions (in red and blue). When clicking a point only shows one point, this indicates that the point was added or deleted.

the two plots could facilitate the comparison of two variable scatterplots. However, even that view does not make the differences visually salient. .

3.5 Usage Scenarios

3.5.1 Gradient Descent

Recall the gradient descent example from . We saw how, by using Anteater, we could visualize the progression of the descent for various training rates. Using ProgDiff, we see the actual differences between two different training rates. shows the comparative

views for two runs at different training rates. We clearly and immediately see the effects of lowering the training rate. We see that the number of NaN's greatly decreased, but in the scatterplot, we see that the descent still oscillates, just not as rapidly. Viewing these two executions disjointly it may not be as apparent that the oscillation rate decreased that significantly or how many fewer NaN's actually occurred.

3.6 Discussion and Limitations

Tree Diffing Algorithm The diffing algorithm has some limitations, its design assumes more simplistic nodes where all nodes are the same type but may differ in value. However, this a program trace contains a variety of node types, each of which still have a value. Therefore, when looking for diffing the execution trees, the algorithm may overlook certain commonalities due to slight structural differences. However, the diffing algorithm used in this work is only one possible example of a diffing algorithm. Due to the modular nature of the system, a different diffing algorithm can easily be inserted into the system. The exploration of more sophisticated diffing algorithms is left for future work.

Scale of Program Changes ProgDiff is designed to support minor changes. Minor changes include those that only alter program values (and not the execution structure) or only minorly alter the execution structure. ProgDiff will still work on more significant changes, however as the changes become larger, the visualizations become more complex and less readable. .

3.7 Conclusion

In this chapter, we presented ProgDiff, an extension to Anteater for visualizing the execution effects of program changes. ProgDiff builds on the tracing infrastructure

of Anteater to trace two versions of a general Python program, identify the changes between the two traces, and present these changes through interactive visualizations.

ProgDiff assumes nothing about a program other than that it is written in Python. It does not assume anything about the structure of the program or even the type of changes made to the program (although it performs better on smaller changes). As a result, ProgDiff limits the types of questions we can ask when preforming comparative tasks. Specifically, ProgDiff only allows us to ask general comparative questions, such as "how does the behavior of this version of value x compare to the the behavior of this other version of value x." It does not enable us to ask deeper, more program dependent questions such as "what is the influence of value x on dependent variable y". To address these questions, we need to explore the types of assumptions we can make about programs and the types of comparative questions enabled by these assumptions. In the next chapter, we explore one application of this with DimReader where we assume that a program (1) takes a dataset as input and (2) performs differentiable calculations on the dataset to produce an output which enables the question "if the data were slightly different, how would the output change".

Chapter 4

DimReader

4.1 Introduction

One of the central promises of data visualization is that its techniques will help users and analysts make sense of large, complicated datasets. Data visualization, and specifically techniques in dimensionality reduction, are routinely used in practice during exploratory data analysis of challenging datasets.

Classical linear methods such as Principal Components Analysis have existed for more than a century, but recent advances from *non-linear* methods that started with Tenenbaum et al's Isomap [111] have revolutionized the practice of dimensionality reduction. The potential to understand high dimensional data via low-dimensional representations is clearly attractive. But just what, exactly, are these non-linear dimensionality reduction (NDR) methods showing? This is the fundamental question that drives the work we report here. Data scientists and analysts use NDR's in an attempt to create a nice 2-dimensional representations for their data in hopes of learning some of the underlying structure of the data. The NDR's often result in nice pictures but give no indication *why* the NDR placed things the way it did and no context to the input.

Consider van der Maaten and Hinton's t-SNE, arguably the most powerful and currently most popular method for NDR [76]. Although practical experience attests to t-SNE's power to uncover cluster relationships in very challenging datasets, its sensitivity to the hyper-parameters is remarkable [120]. If small changes in parameter settings produce plots that are fundamentally different, we must ask ourselves: are some results generated by NDR methods just bad? Do different parameter settings show different features of the data? More importantly, how do we even answer these questions?

In this work, we design data transformations, which induce transformations on the visualization itself, elucidating the behavior of the NDR method (this is the perspective

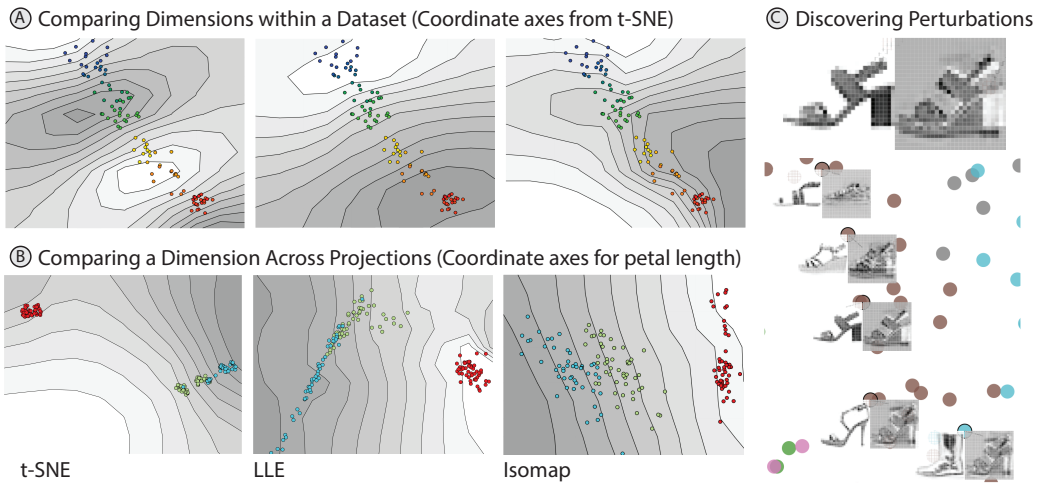

Figure 4.1: DimReader explains non-linear dimensionality reduction methods by illustrating the effects of user-designed perturbations of the input dataset. It provides an answer to the question "if the input data had been slightly different in a particular way, how would the plot have changed?". In the case of traditional scatterplots, it recovers exactly the axis lines being displayed. In the case of non-linear methods, DimReader recovers *generalized axes*, which indicate how dimensions of interest behave. Examples of these axes are shown in (A) for the x, y, and z dimensions of the S Curve (an S shaped 3 dimensional manifold). These axes also allow for the comparison of different projection methods. This is exemplified in (B), where the petal length axis of the iris dataset is shown for three projections. Petal length is well behaved in t-SNE but not in the other projections . We also provide a technique for discovering good perturbations of the input (perturbations that change the projection the most). The top of (C) shows an example of a discovered perturbation. In context, shown at the bottom of (C), this perturbation shows us that t-SNE is sensitive to flat shoes v.s. heels. The perturbation wants to change the original image from a heel to a flat by filling in the arch.

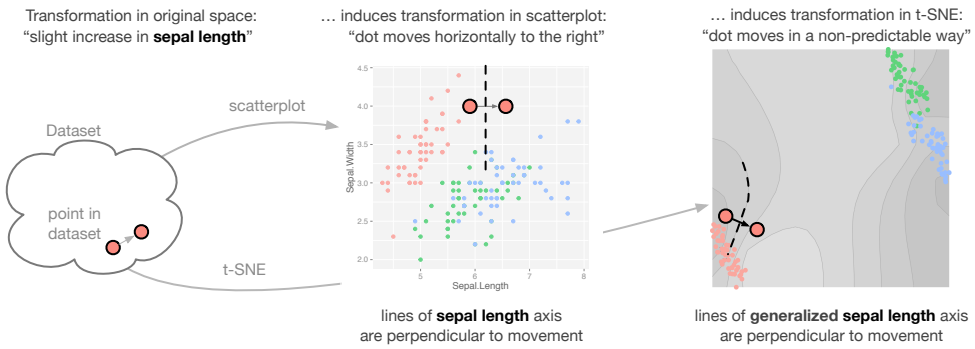

Figure 4.2: In traditional scatterplots, the grid lines (or axes lines) exist to explain what the plot is showing. Equivalently, they capture *infinitesimal perturbations* of the dataset in specific directions, because they are always perpendicular to the directions of movement. DimReader extends the same principle to non-linear dimensionality reduction (NDR) methods, and recovers *generalized axis lines*, which help explain NDR methods in terms of interpretable data transformations.

introduced by Kindlmann and Scheidegger's algebraic design process [66]). Specifically, we use infinitesimal perturbations — small changes of the data in its original space — to produce infinitesimal changes of the visualization. We then show how these visualization changes can be interpreted as producing *effective, non-linear axis legends*. In this way, our non-linear axes *explain* the NDR plot in the same way that axis legends explain the positional encoding in scatterplots. As a result, analysts can understand and evaluate dimensionality reduction plots similarly to how they evaluate *linear* methods. In fact, we show in Section 4.3 that our methods exactly recovers the gridlines of typical scatterplots. DimReader is quite general, and can be applied to many different NDR techniques, only requiring access to the source code of its implementation. Specifically, we use a method known as *automatic differentiation* to produce the necessary gradients for calculating the infinitesimal changes of the visualization [54]. An overview of the process is given in Figure 4.2.

In summary, our contributions are:

- A general framework to explain plots generated by non-linear dimensionality re-

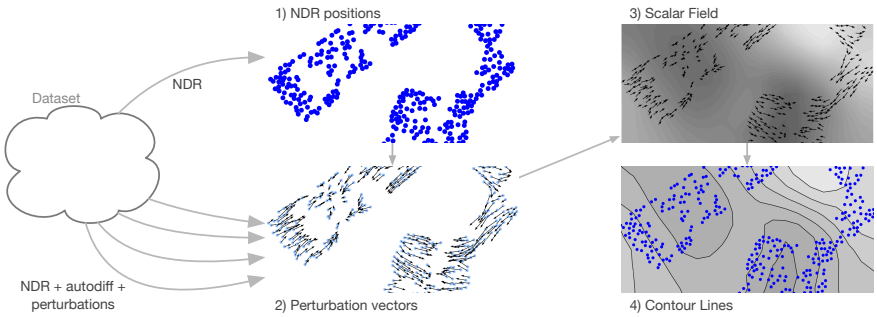

Figure 4.3: An overview of DimReader. For a given NDR method, we 1) compute its position using the original implementation; 2) compute perturbation directions for the input points with the transformed version of the implementation which uses dual numbers (We discuss how to choose appropriate perturbations in Section 4.3.2); 3) compute the scalar field whose gradient best matches the perturbation vectors in a least-squares sense; and finally 4) compute its isocontours. Section 4.3 explains these steps in detail.

duction, using infinitesimal perturbations

- A practical implementation of the framework using automatic differentiation
- A method for discovering good perturbations for a given dataset, useful when the input lacks easily interpretable dimensions (and hence, lacks easily-defined perturbations)
- An experimental study of the effectiveness and efficiency of DimReader using three well-known NDR methods: Isomap [111], LLE [92], and t-SNE [76].

4.2 Related Work

Projection methods have received a considerable amount of attention in information visualization. In this section, we review the work that is most directly related to our research, but cannot hope to cover the entirety of the field. For a comprehensive view on multidimensional scaling and dimensionality reduction, we recommend Born and Grönen's textbook [13], and Fodor's survey [47].

Projection methods in information visualization The observation that pairwise similarities (or distances) can be converted into low-dimensional representations by a mathematical formulation comes from Torgerson and his now-classical theory of multidimensional scaling [113]. In information visualization, force-directed methods have long been used as a dimensionality reduction technique, from fully-automatic methods [25], [60], [80], to methods which take some amount of interaction, either through placement of exemplar points [41], [63], [83] or through direct interaction with projection parameters [62]. Although interactive methods offer a better hope for understandability because the perturbation analysis we discuss can happen "in the analyst's head" during interaction, we argue that the visual encoding these techniques provide can still be unclear. The technique we propose here can be applied to essentially all of the methods above, and offers an attractive complement to both automated and interactive projection methods.

Perturbation Analysis for data science The idea of understanding a system by examining its behavior under perturbations is well-established in the engineering and statistics literature. In the 1970's, Cook introduced the notion we now know as Cook's distance [32], which measures the influence of a point on the parameters of linear regression models. In the context of visualization, Bergner et al. point to sensitivity analysis as one of the requirements in understanding computer simulations [9]. In this paper, we use perturbation analysis as a central tool to recover readable axes from NDR methods, in a sense incorporating sensitivity analyses into familiar visual metaphors.

Automatic Differentiation Perturbation analysis is clearly an important tool for understanding systems, but the issue of how to implement it in existing computer systems is crucial. Automatic differentiation (which we explain in detail in Section 4.3) provides a way to compute derivatives of arbitrary functions in a computer program, provided access to the source code (or similar structural information about the com-

putation) is available [54]. To the best of our knowledge, the most mature software library employing automatic differentiation is Ceres, written in C++ and employing template metaprogramming [3]. DimReader is implemented in Python for simplicity and terseness, but could easily be redesigned in C++.

Guidance and validation of projection results One of the issues with NDR is that it's hard to know what a plot is actually showing [116]. This has resulted in a variety of papers which offer guidance and design principles on how to interpret projections, based on a combination of real-world experience, synthetic examples, and theoretical arguments [18], [70], [94], [98], [99]. This work is essential to the current practice, we argue, because current NDR methods do not offer explanations of their own results — there are much fewer research papers offering guidance for understanding and interpreting traditional scatterplots. As we show in Section 4.4, our technique provides a way for a projection method to explain itself. Although analyst guidance and validation will always be a part of a well-designed analysis infrastructure, our technique could mitigate some of the problems that have been observed in deployed systems, where projection methods are ultimately discarded because of readability issues [16], [59].

Lee et. al. review measures for assessing the quality of NDR techniques .

Augmented visual representations There is another avenue of attack on the readability problem of NDR methods. Often, researchers will *augment* the results of the projections with visual diagnostics that pinpoint potential problems. Seifert et al. augment the projection by showing how the projection's stress (roughly the discrepancy between source-space distances and target-space distances) varies spatially in the NDR plot [100]. In [7] and [72], the projection is augmented to show uncertainty measures and distortions in NDR's respectively. Cutura et. al's VisCoDeR allows users to compare and explore different dimensionality reductions by augmenting the projection to allow users to explore how dimensions are mapped in the dimensionality reduction re-

sults as well as the high-dimensional proximity of projected points to a selected point in the projection [36]. Stahnke and co-authors described methods to *probe* a projection, through carefully designed user interactions and custom visual encodings [105]. Our method for extracting effective axes can be seen as a way to allow *any* NDR method to augment itself with metaphors that have a well-defined analogy in the linear case, as can be seen in Section 4.4, and Figure 4.7 specifically. In Section 4.4.3, we provide a more direct comparison to some of the methods used in Stahnke et al.'s work.

Explainable visualizations Every plot assumes an audience that can read it, and visualization literacy remains an active area of research [15]. Often, novel metaphors are necessary because of the data or task complexity [17]. We argue that generalizing well-established techniques such as axis legends to NDR can help explain those techniques. Gleicher's Explainers take user interaction to design specific projections for input data [49]. In contrast, our technique extracts axes inherent in the non-linear projections. Coimbra et al. explain projections through enhanced biplots [31]. Similar to our technique, they show axes for the dimensions of the input data in the low dimensional plot. Because of the non-linearity of these dimensionality reductions, the biplot axis will change based on the projected position of the sampled point whereas our technique captures the axis lines for the entire projection. A similar approach is proposed by Cavallo and Demiralp in Prolines, a technique for interacting with data points in both low- and high-dimensional spaces [24]. Prolines allow efficient, direct manipulation of the output points, but require access to efficient forward and backward projection, limits its applicability. Flow-based scatterplots [26] and Generalized Sensitivity Scatterplots [27] show the sensitivity of a dimension in a scatterplot with respect to other dimensions in the dataset. Similar to our technique, these methods use derivatives to determine the sensitivities. Our technique differs by showing sensitivity of the projection with respect to the original data rather than sensitivity between

dimensions in the original data. The data context map from Cheng et. al. provides a way to simultaneously look at clusters of data points and the location of the most dominant values of each attribute with the assumption that the attribute values always decrease as points move farther away from it. [29]. Our technique differs by showing the behavior of a dimension throughout the entire projection, not just the location of the most dominant value.

4.3 Technique

Our technique is broken into two parts: (1) explaining an NDR method using a known perturbation (DimReader) and (2) searching for good perturbations when there is no known perturbation (after which DimReader can be applied).

In principle, all that DimReader requires is the ability to compute derivatives of the projection coordinates with respect to each of the input points. For extremely simple techniques (such as scatterplots and other fixed linear projections), these derivatives can easily be evaluated in closed form. However, more sophisticated methods such as Isomap, LLE, and t-SNE involve long computation chains, for which the evaluation of the derivative would introduce significant development overhead. Instead of trying to solve them in closed form, we take central advantage of automatic differentiation. [54]. As we describe next, automatic differentiation allows us to calculate the derivative of a projection with minimal implementation effort.

4.3.1 Automatic Differentiation

In this paper, we use a particular form of automatic differentiation known as forward-mode automatic differentiation. In what follows, we will refer to it as "autodiff".

In forward-mode autodiff, the program's derivative with respect to a specified variable is computed alongside the function value, by using an *extended number system*. In

this system, we replace numbers in the program with dual numbers that have the form $x = (a, b)$ where a holds the original value of the number and b carries the derivative of x with respect to our variable of interest. When we initialize a variable y, we set b to one if that is the variable we want to differentiate with respect to (since $dy/dy = 1$) and zero otherwise. When the projection is run with dual numbers in place of regular numbers, in addition to calculating the projected points, it calculates their derivatives through applications of the chain rule and derivative rules (product rule, quotient rule, etc.).

Note that autodiff is always performed at a specific value, and with respect to a specific variable. It produces two numbers as a result: the function value and the partial derivative with respect to the chosen variable. This has two important consequences for our design. First, we need to decide over exactly which variables we will take derivatives. Second, we need to execute the program many times in order to evaluate many different derivatives. This will become important in Section 4.4.4.

4.3.2 DimReader Process

Overview of the process

To apply DimReader to an NDR method, there are four steps. Each of these steps is discussed in a subsection below.

- A user chooses a perturbation of interest, which defines an infinitesimal change for each data point (possibly in different directions).
- The NDR method is executed many times using dual numbers, from which we obtain the perturbation vectors, one for each input point.
- From the perturbation vectors, a scalar field whose gradient matches the perturbation vectors is computed.

```
# Basic method, O(numPoints) runs
for i in range(0, numPoints):
    points = copy(inputPoints)
    points[i] = perturb(points[i], perturbation)
    projection = project(points) # project uses autodiff
    dx, dy = projection.derivative[i]
    projectionVectors[i] = vector(dx, dy)
return projectionVectors

# Improved method, O(log(numPoints)) runs
counts = zero_array(numPoints)
projectionVectors = zero_matrix(numPoints, 2)
while any(counts < 1):
    points = copy(inputPoints)
    for i in range(numPoints):
        if random() < 0.5: # perturb each point with probability 0.5
            perturbed[i] = true
            points[i] = perturb(inputPoints[i], perturbation)
    projection = project(points) # project uses autodiff
    for i in range(numPoints):
        if perturbed[i]: # only store vectors of perturbed points
            dx, dy = projection.derivative[i]
            projectionVectors[i] += vector(dx, dy)
            counts[i] += 1
for i in range(numPoints): # average all perturbations performed
    projectionVectors[i] /= counts[i]
return projectionVectors
```

Figure 4.4: Although a basic implementation of DimReader is easy to understand (top), it only extracts one perturbation vector at a time. A more efficient implementation (bottom) extracts half of the perturbation vectors from the input at once. To remove possible correlations between the outputs, we choose which points to include at random, and iterate until all points have been included. The expected time in this case is logarithmic on the size of the input point dataset.

- The isolines of this scalar field, which are perpendicular to the gradient, are extracted using Marching Squares. They form the effective axes.

Choosing which perturbation to use

The first step of our method involves a choice of the perturbation of the dataset. A **perturbation** is a small change to a specific dimension (or set of dimensions) for each data point in the original, high-dimensional space. Thus, the choice of perturbation corresponds, effectively, to an analyst answering the following question: "if each data point were slightly different in this specific way, what would happen to the visualization?" In order to recover different features of the NDR method and its effect on the dataset of interest, different perturbations can be designed. In the following, we discuss choosing a perturbation in a dataset with interpretable dimensions. We discuss discov-

ering perturbations for other datasets in Section 4.3.4. In automatic differentiation, perturbations are represented by the derivative part of the dual number for the original data points. The perturbation of a data point with d dimensions has the form of a unit vector with d elements where the value of each element specifies how much the corresponding dimension is perturbed relative to the rest of the dimensions.

Datasets with interpretable dimensions Some datasets have interpretable columns. Take the iris dataset, for example, which is used in Figure 4.2. In that case, a perturbation that changes each of the input points in the direction of a given dimension will reconstruct, for an NDR method, curved axes lines that correspond, roughly to the linear grid lines in scatterplots. Concretely speaking, we evaluate each input point p_i as $(p_i, (0, \cdots, 0, 1, 0, \cdots, 0))$, where the value 1 is positioned at the dimension of interest.

Extracting derivatives from NDR methods

In this section, we describe two techniques used in DimReader to extract the **perturbation vectors** ,the changes to the projected coordinates resulting from a perturbation on the input, for a given projection. The first technique is simple, straightforward, and provides a good intuition for the overall strategy. Unfortunately, this technique requires as many executions of the NDR method as there are input points in the dataset, which often means the overall performance can suffer. The second technique, on the other hand, only requires as many runs as the *logarithm* of the number of input points. We give pseudo-code for the two approaches in Figure 4.4.

DimReader needs access to the source code for the NDR method at this step so the method can be executed with dual numbers. In principle, the source code can be executed without any modifications aside from converting the input points into dual numbers. In practice, some issues arise because of efficiency concerns and library limitations. We discuss these issues at length in Section 4.4.

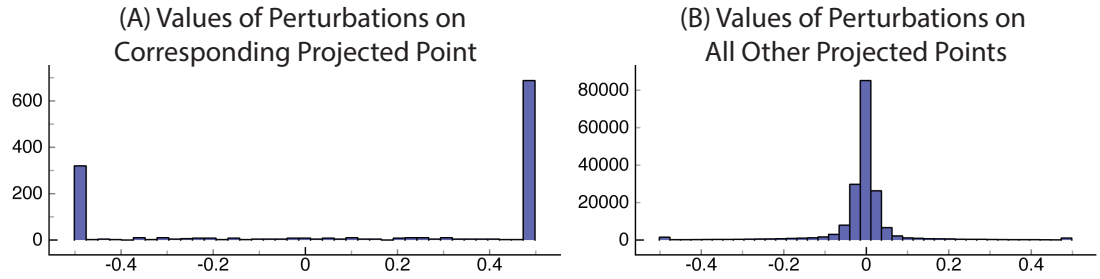

Figure 4.5: (A) shows the effect sizes of perturbing point p_i on its corresponding projected point, v_i. DimReader uses these values in its computations: note their large magnitude (values outside of the displayed range are clamped at -.5 and .5). (B) shows the effects sizes of perturbing point p_i on all projected points aside from v_i. DimReader assumes these values are zero and discards them: note their small magnitude.

Perturbing one point at a time After a perturbation is chosen, the NDR technique is executed with automatic differentiation for every point in the dataset. On execution i, the point p_i is perturbed (that is, we replace p_i with $(p_i, \bar{p_i})$ where $\bar{p_i}$ is the specified perturbation of p_i). The NDR technique will return the projection coordinates, v, for all points, along with the derivative of the projection coordinates with respect to the perturbation of p_i, $\frac{dv}{dp_i}$. We use the derivative of each coordinate in the reduced point v_i as the vector that describes the change in the coordinate, and discard the rest of the information of the run. The pseudocode for this is given on the top half of Figure 4.4.

Perturbing many points at a time The method described above is inefficient, requiring $O(n)$ evaluations of the NDR method. A naive attempt to optimize the method would evaluate the projection derivatives with respect to all of the points (and hence all of the per-point perturbations) at once, and only run the autodiff version of the code once. Unfortunately, this does not work for many perturbations, because most DR methods are *invariant to dataset translations*. The perturbation of only one input point at a time offers interesting insight into the NDR method, but if we move all of the points at once in the same direction, NDR methods such as Isomap, LLE, and t-SNE will produce exactly the same projection (the perturbation vectors will be all zeros).

We solve this problem by adding a small amount of randomization. Instead of perturbing one point at a time, we can choose half of the points at random to perturb, while the other half does not change. We then store the projection vectors for the points we chose to perturb, and repeat the process until we have actually perturbed all of the input points. After each round, we expect to halve the number of unperturbed points, which gives an expected number of repeated runs which is logarithmic on the number of input points. The pseudocode for this is given on the bottom half of Figure 4.4. We found that the DimReader plots produced by perturbing many points at a time are indistinguishable from the plots produced by perturbing one point at a time.

Ignoring changes in unperturbed points In some cases, perturbing a point p_i has an effect on points other than its corresponding projected point v_i. However, the effects on other points are small enough that we can effectively ignore them. In Figure 4.5, we show that the effect of perturbing each point p_i on the other points, v_j $(i \neq j)$ tends to be near zero and very rarely is significant. We show this result for the iris dataset, but have found it to be true in general for t-SNE in all datasets we checked. Intuitively, we expect a good dimensionality reduction to be robust to a small change in a single point, and thus the residual effect on the rest of the points to be insignificant.

Reconstructing the direction field

Once we have the projected points and their derivatives (that is, the perturbation vectors), we need to reconstruct the direction field, in order to extract perpendicular lines. We achieve this by computing a scalar field whose gradient best matches the vectors. We use a simple least-squares reconstruction technique, adapted from Ferreira et al.'s vector-field clustering work [43], which we illustrate in Figure 4.6. We first decompose the output plane in a rectangular grid, and split each grid square into two triangles, giving a triangular mesh of the output space. The resolution of this grid needs

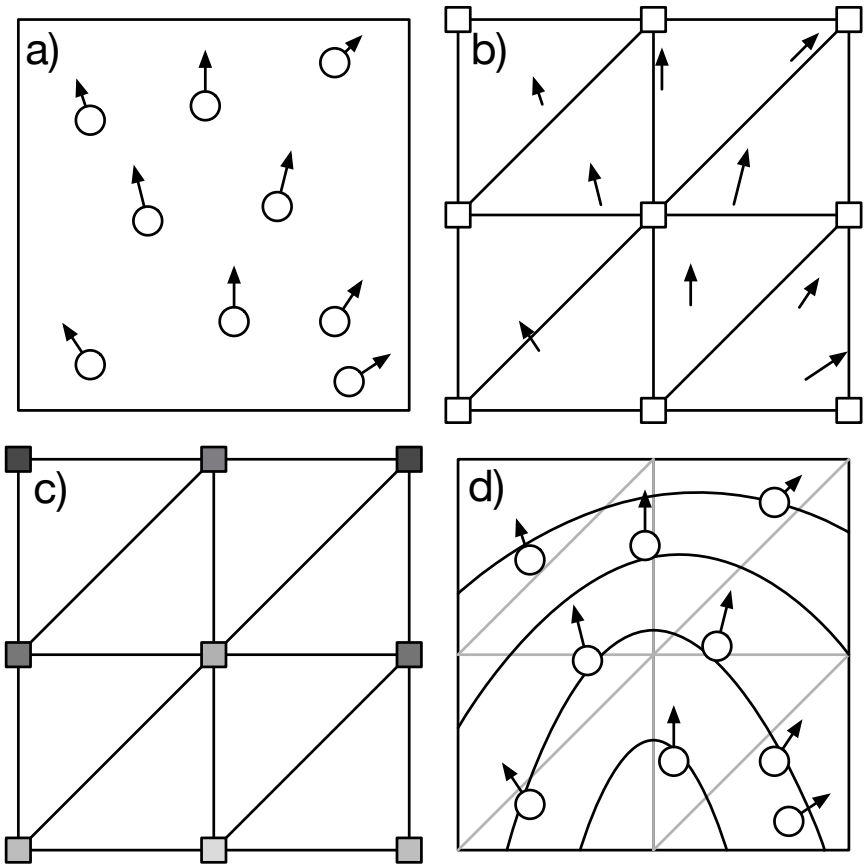

Figure 4.6: An illustration of the process to recover generalized axes. Given the point positions and perturbation vectors (a), we construct a triangular mesh and interpret each vector as a linear constraint on the gradient of a function (b), which gives values on each of the vertices (c). From these values, we can extract lines perpendicular to the perturbation vectors using marching squares.

to be decided ahead of time, and we use a 10x10 grid in our examples for this paper. We model a scalar field on the output plane as a piecewise-linear function on the grid values. Each point and its perturbation vector is interpreted as a linear constraint on the vertices of its corresponding triangle. To find the best-fitting scalar field, we solve it in a least-squares sense, regularizing the system to ensure a unique solution [43].

Extracting perpendicular lines

The final step is quite simple. With the scalar field expressed as values in a triangular mesh, we can use marching squares to extract isocontours [5]. By construction, the gradient of this scalar field matches the perturbations. Since isolines are perpendicular to a function's gradient [96], the resulting curves will tend to be perpendicular to the perturbations. As we show in Figure 4.2, these isoline can be thought of as *generalized axes lines*.

4.3.3 Interpreting DimReader Plots

In our plots, the projected points would move perpendicular to the isolines nearest to them if the input were perturbed in the specified way. An increase in the corresponding dimension would move the point from light to dark. The relative density of the isocontours can be interpreted similarly to the behavior in scalar fields. Narrowly-spaced isocontours indicate a high sensitivity to changes in the *independent* variable, (in our case, projection coordinates). Widely-spaced isocontours indicate *low spatial sensitivity*: a change in the projection coordinates is not expected to change the outcome variable by much. Curved isolines indicate that the same perturbation has a different effect on different points. Isolines that fan out (go from narrowly-spaced to widely-spaced as in Figure 4.8) indicate that the sensitivity of the plot is changing from more sensitive on one side to less sensitive on the other.

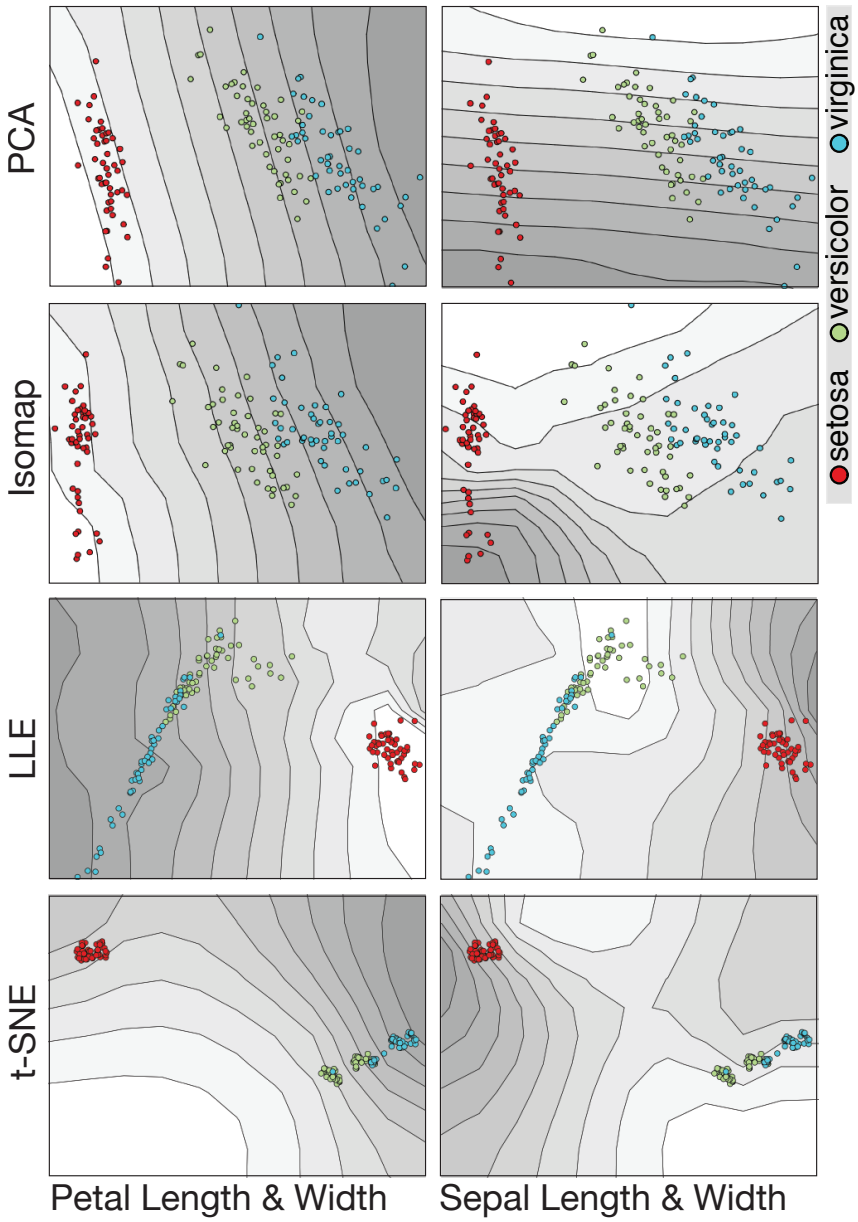

Figure 4.7: Extracting axes from the Iris dataset with four projections: PCA, Isomap, LLE, and t-SNE. We only show petal length and sepal width because petal width is extremely similar to petal length and sepal length is very similar to sepal width. We discuss how to interpret these plots in Section 4.4.1

4.3.4 Discovering Good Perturbations

We may not always know good perturbations for a dataset, such as the MNIST digits where it is not clear what the best way to perturb each image would be. To help solve this problem, we offer two methods to recover good perturbations. We define good perturbations as perturbations on the input that change the projection the most under the given constraints. Both of the methods require that we have a tangent map, M, for the projection. The tangent map allows for efficient calculation of the perturbation vector, $\bar{v}$, for a given projection, without running the projection itself. Given a perturbation on the input $\bar{p}$, $M \cdot \bar{p}$ results in the perturbation vector $\bar{v}$, i.e. $\bar{v} = dv/dp$ where v is the projected coordinates. The vector $\bar{p}$ consists of the perturbation of each input point concatenated into a single vector (end to end) and the perturbation vector $\bar{v}$ consists of the perturbation vector $\bar{v_i}$ of each projected point (v_i) concatenated into a single vector. A single column of M can be recovered by a perturbation vector that has a single entry of 1 and the rest zeros (i.e. perturbing a single dimension of a single point). By doing this for each dimension of every point, the entire tangent map can be recovered.

One observation about the tangent map is that the values we need for calculating the perturbation vectors lie in $k \times d$ blocks along the diagonal, where k is the dimension of the projection (typically 2) and d is the dimension of the input data. Because we ignore the effect a given perturbation on all other points (as discussed in 4.3.2), we set the rest of the matrix to zero. We exploit the block structure of the tangent map in both of our methods for finding the best perturbation.

Perturb all points in the same direction

The first method for recovering the best perturbation is to find the single perturbation that when applied to all points changes the projection the most. The formulation for

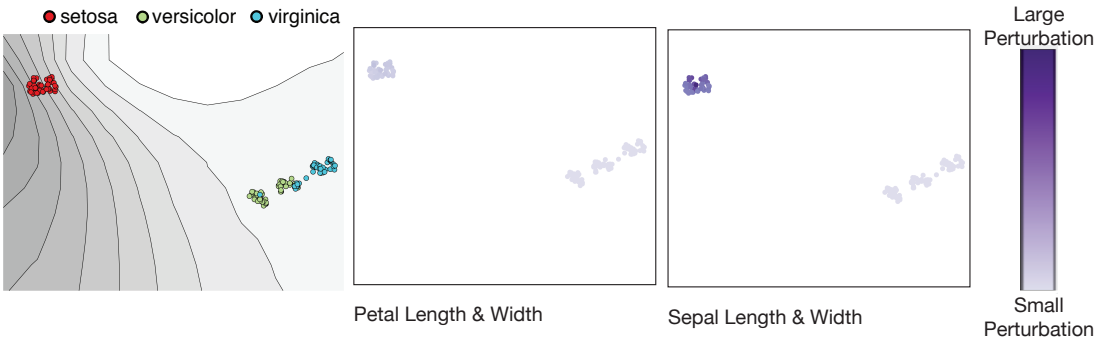

Figure 4.8: The best perturbation for the iris dataset. The left plot is the DimReader plot for this perturbation. In the two plots on the right, the color of each point shows how much the point is perturbed for the specified dimension. We see that the sepal width and sepal length are perturbed more in the Red cluster (the Setosa cluster) than the petal dimensions which means that, for this cluster, the projection is sensitive to changes in the Sepal dimensions. The perturbations for the points in the other cluster are insignificant. This tells us that perturbing only the Setosa points will change the projection the most.

this method is:

$$\underset{\bar{v}\in\mathbb{R}^d}{argmax} \sum ||B_i\bar{v}||^2 \quad s.t \ \ ||\bar{v}^2|| = 1$$

where B_i is the block on the diagonal of M for point i. This can be rewritten as $\sum_i \bar{v}^T B_i^T B_i \bar{v} + \lambda(\bar{v}^T\bar{v} - 1)$. To find the maximum, we take the derivative with respect to $\bar{v}$ and set it to zero: $\frac{d}{d\bar{v}} \sum_i \bar{v}^T B_i^T B_i \bar{v} + \lambda(\bar{v}^T\bar{v} - 1) = \sum 2B_i^T B_i\bar{v} - \lambda 2\bar{v} = 0$. The best perturbation vector is the eigenvector of the matrix $\sum_i B_i^T B_i$ with the largest eigenvalue. This gives us a single perturbation, $\bar{v}$, that when applied to all points maximizes the change in the projection. $\bar{v}$ is constrained to have unit length to prevent the method from choosing an arbitrarily large perturbation.

Perturb each point individually

The second method for recovering the best perturbation is to find different perturbations for each point that collectively change the projection the most constrained so that points

that are projected to similar places have similar perturbations. The formulation is as follows:

$$\underset{\bar{v}\in\mathbb{R}^d}{argmax} \sum_i ||B_i\bar{v}_i||^2 - \lambda \sum_i \sum_j ||\bar{v}_i - \bar{v}_j||^2 S(i,j) \quad s.t \ \ ||\bar{v}^2|| = 1$$

where B_i is the block on the diagonal for point i, $\bar{v}_i$ is the perturbation for the point i, λ is a free parameter for how much smoothing we want, and $S(i,j)$ is the similarity between the projection of points i and j, p_i and p_j. This similarity is defined as $S(i,j) = e^{-||p_i - p_j||^2/\sigma^2}$. σ^2 is a free parameter set by the user that determines how close points have to be in the projection to be considered similar. We can rewrite the above equation as follows:

$$\underset{\bar{v}\in\mathbb{R}^d}{argmax} \sum_i \bar{v}_i B_i^T B_i \bar{v}_i - \lambda \sum_i \sum_j \langle \bar{v}_i - \bar{v}_j, \bar{v}_i - \bar{v}_j \rangle S(i,j) \quad s.t \ \ ||\bar{v}^2|| = 1$$

We observe that $\sum_i \sum_j \langle \bar{v}_i - \bar{v}_j, \bar{v}_i - \bar{v}_j \rangle S(i,j)$ takes form similar to a Laplacian matrix, L_s multiplied by the entire perturbation vector v (the concatenation of all of the individual perturbations, $\bar{v}_i$) on both sides: $\bar{v}^T L_s \bar{v}$. L_s differs from a standard Laplacian matrix in that rather than having diagonal values $\sum_{j\neq i} S(i,j)$ and off diagonal values $-S(i,j)$, it has diagonal values $I * \sum_{j\neq i} S(i,j)$ and off diagonal values $I * -S(i,j)$ where I is $d \times d$ identity matrix.

Furthermore, the equation can be rewritten in terms of the whole matrix, M, and the entire perturbation vector, $\bar{v}$ giving us the following equation which incorporates the constraint on the length of $\bar{v}$:

$$\underset{\bar{v}\in\mathbb{R}^d}{argmax} \ \ \bar{v}^T M^T M \bar{v} - \lambda \bar{v}^T L_s \bar{v} - \lambda_2 \bar{v}^T \bar{v} - 1$$

Taking the derivative with respect to $\bar{v}$ and setting it to zero, we get

$$\bar{v}^T(M^TM - \lambda L_s) - \lambda_2\bar{v} = 0$$

The entire perturbation vector, $\bar{v}$, is the eigenvector of the matrix $M^TM - \lambda L_s$ with the largest eigenvalue.

Choosing λ and σ. σ controls the width of a gaussian centered on each point. Examining the results of the projection itself gives some information about plausible values for σ. For example, points outside further than three standard deviations from each other are essentially treated independently, but at the same time, we don't want a σ that creates a gaussian which covers the entire projection. For choosing λ, we should be looking at the resulting perturbations. If a single point is heavily dominating the perturbation (i.e. it moves much more than the rest of the points) then λ is likely too small. In contrast, if all points are perturbed in almost exactly the same way, this is an indication that λ may be too large.

4.4 Implementation and Experiments

In this section, we discuss the implementation of our techniques along with a suite of experiments designed to explore the capabilities, performance, and limitations of DimReader. We will show how DimReader directly addresses the following gaps identified in Sedlmair et al.'s interview study about gaps between theory and practice in dimensionality reduction (DR) [98]. These include the *interpretation gap*: "what do the results mean?"; *guidance gap*, "what algorithm to use?", and the *non-linear unmapping gap*: "how do projection dimensions relate to input dimensions?".

Our current prototype for DimReader is implemented in Python and numpy [119]. Our t-SNE implementation is closely based on van der Maaten's Python code [75], while the LLE and Isomap implementations are from-scratch. The entire method takes about

3,500 lines of Python, including implementations of Marching Squares, the classes for autodiff, and the linear solvers described below.

4.4.1 DimReader

In the following we look at a well known dataset, the iris dataset, with known perturbations, and simple projection algorithms, in order to better understand the behavior of the technique [67].

Linear projections

We start with showing results of linear projections as a basic sanity check on the behavior of DimReader. Figure 4.7 shows a typical example of the axes reconstructed by DimReader when using linear projections. Since linear projections can be exactly represented by a matrix multiplication, the derivatives of input points position with respect to one direction will always be constant vectors. As a result, the reconstructed scalar field is almost (except for the influence of the regularization terms) a linear ramp, and so the contour lines are evenly spaced and parallel, which indicate that changes in the input variable will behave identically across the entire field. Despite their limited power, this property is one of the main advantages of linear projections.

Isomap

Isomap was one of the first NDR techniques to recover curved manifolds well in practice [111]. Isomap builds a weighted graph which approximates the manifold, where edges have weight equal to the distance between points, and each point has edges to its k nearest neighbors (k is specified by the user). The global distance between two points is defined to be the shortest-path metric on the graph. The low-dimensional projection is constructed from the shortest-path metric using classical MDS [13].

We implemented Isomap not only because of its historical significance and relatively

high-quality results, but also because it highlights an interesting property of automatic differentiation: it works over code bases that we tend to not think of as differentiable. Specifically, the operations in Dijkstra's algorithm for shortest paths are all well defined for dual numbers, and so we naturally can extract the sensitivity of shortest-path distances with respect to changes in the input points [33].

Interaction with numerical linear algebra routines The final step of Isomap is Classical MDS, and this presents unique challenges for our autodiff implementation based on operator overloading. Specifically, Classical MDS requires the computation of eigenvectors, and since Python libraries for numerical linear algebra are implemented through high-performance libraries like Lapack, the operator-overloading functionality is not present. To solve this issue, we implement the eigenvalue computation through power iterations [51], since matrix-vector multiplication of dual numbers has efficient dual-number implementations in terms of matrices of values and ϵ terms.

Isomap Experiments Because Isomap uses classical MDS (which is essentially a linear projection), we should expect that, to some degree, Isomap would behave much like linear projections. This is indeed the case with simpler datasets, such as the Iris dataset, shown in Figure 4.7. However, there are some interesting differences. Consider the generalized axis for the "sepal width" variable which DimReader recovers. Even though the point positions generated by Isomap are quite similar to that of PCA, the sensitivity of the projection differs dramatically from the cluster of Setosa samples to that of Virginica and Versicolor samples. Even more interestingly, it seems that the sensitivity is caused by only some of the Setosa samples. This differentiation is not present in the linear projection, and would not be clear from the Isomap plot alone. In this example, DimReader helps overcome Sedlmair's *interpretation gap* by providing an explanation for why Isomap spread the points in the Setosa cluster (Isomap is sensitive to differences in the Sepal Width in this cluster) that would otherwise be unknown.

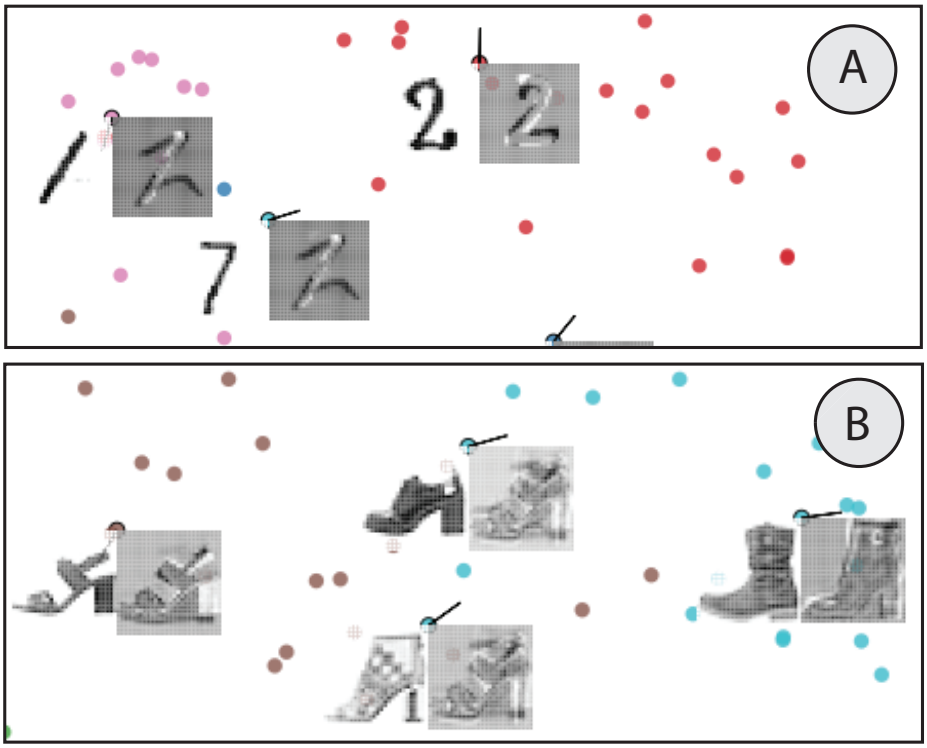

Figure 4.9: An overview of perturbations for points in the(A) MNIST digits and (B) MNIST fashion. There is structure in the perturbations that our technique discovers. They often resemble their true digit (or clothing article) but with some variation. Darker areas are perturbed more than lighter areas.

Locally Linear Embedding

The next algorithm we highlight is Roweis and Saul's Locally Linear Embedding [92] (LLE). Like Isomap, LLE uses a nearest-neighbor graph to recover a global view of the dataset. LLE computes edge weights for the nearest neighbor graph, such that each vertex can be best reconstructed by a linear combination of its neighbors using those weights. On a second step, the projection coordinates are recovered by finding positions on the plane that best respect the weights.

Interaction with numerical linear algebra routines Similarly to Isomap, our autodiff implementation of LLE involves a small degree of adaptation. In the case of Isomap, we required the computation of the largest eigenvalues of a matrix. In the case of LLE, we need to compute the *smallest non-zero* eigenvalues. Our implementation uses *inverse power iteration* [51]. Inverse power iteration, in turn, requires a linear system solver, which presents similar issues for dual number implementations. Our solution is to implement a black-box linear system solver using conjugate gradients [102].

LLE Experiments Locally Linear Embedding is a popular method due to its performance [92], but is known to produce distorted projections [39]. In this section, we illustrate how DimReader might help pinpoint such problems. Consider the projection of the iris dataset in Figure 4.7. Note that neither of the recovered axes quite cross the projection perpendicularly on the left side of the arc (the Versicolor and Virginica cluster): no direction of perturbation on the input moves the points diagonally along that cluster. This suggests that the shape of the cluster is an artifact of the projection method. Compare this with the Isomap projection in Figure 4.7: Isomap has perturbations which cross each of the clusters perpendicularly (Sepal Width for Setosa and Sepal Length for Versicolor and Virginica). Thus, Isomap is more faithful to the underlying data than LLE. This is evidence that DimReader helps bridge Sedlmair et. al's

guidance gap [98], giving an indication for which NDR algorithm performs better for this data.

t-SNE

t-SNE is among the most powerful techniques for dimensionality reduction, and also one of the hardest to interpret appropriately [76], [120]. As such, it is a natural target for DimReader. In addition, t-SNE is significantly different from Isomap and LLE in both formulation and implementation. This provides us with an opportunity to explore practical issues of using DimReader to explain its results.

We highlight two separate issues to discuss: the presence of multiple local minima, and its formulation in terms of the gradient of an energy function. While the first issue presents challenges for implementations that depend on repeated executions, the second issue allows us to achieve a significant speedup.

Multiple minima The energy function that t-SNE minimizes has more than one local minimum. This means that any source of randomness in the implementation will cause multiple runs to possibly diverge, presenting a challenge for our approach. Most implementations of t-SNE require an initial guess for the projection, and we take central advantage of this. Specifically, in our first execution of t-SNE we use a random initial guess and regular floating-point numbers to calculate a local minimum that is then used as the initial guess for subsequent runs. In the initial run we also capture variables that serve as parameters for subsequent runs to ensure that they reach the same local minimum.

Gradient descent t-SNE is implemented as an explicit gradient descent formulation through an additive update of the parameters. Specifically, the main loop of t-SNE is roughly as follows:

```
pos = initial_guess
g = gradient(energy(pos), pos)
while mag(g) > epsilon:
    pos = pos - rate * g
    g = gradient(energy(pos), pos)
```

As a result, when the loop exits, we know that the gradient of the energy with respect to the position will be close to zero. This means that to recover any one perturbation of the t-SNE formulation with respect to an input point, all that is required is to run one single iteration of t-SNE with dual numbers. By providing the dual-number implementation the result of the execution of the floating-point implementation (as explained in the previous paragraph), the loop will execute at most once before exiting – in fact, in order for the sensitivity of the positions with respect to the input to be recorded in the `pos` variable, we must force the loop to execute at least once. Still, since t-SNE typically executes between 100 and 1000 iterations in this loop, this simple optimization achieves a significant speedup.

t-SNE Experiments T-SNE is often considered to be the state of the art in NDR methods, but one of the main objections to its use in practice is the opaque nature of its optimization criteria [120]. It is unclear how effectively the projection recovers high dimensional information. Consider the t-SNE axes in Figure 4.7. t-SNE is detecting variation in the petal length and petal width of the Virginica and Versicolor cluster and subsequently spreading the cluster based on these dimensions. This helps explain how petal length behaves in the projection, providing evidence that DimReader helps bridge the *non-linear unmapping gap* [98].

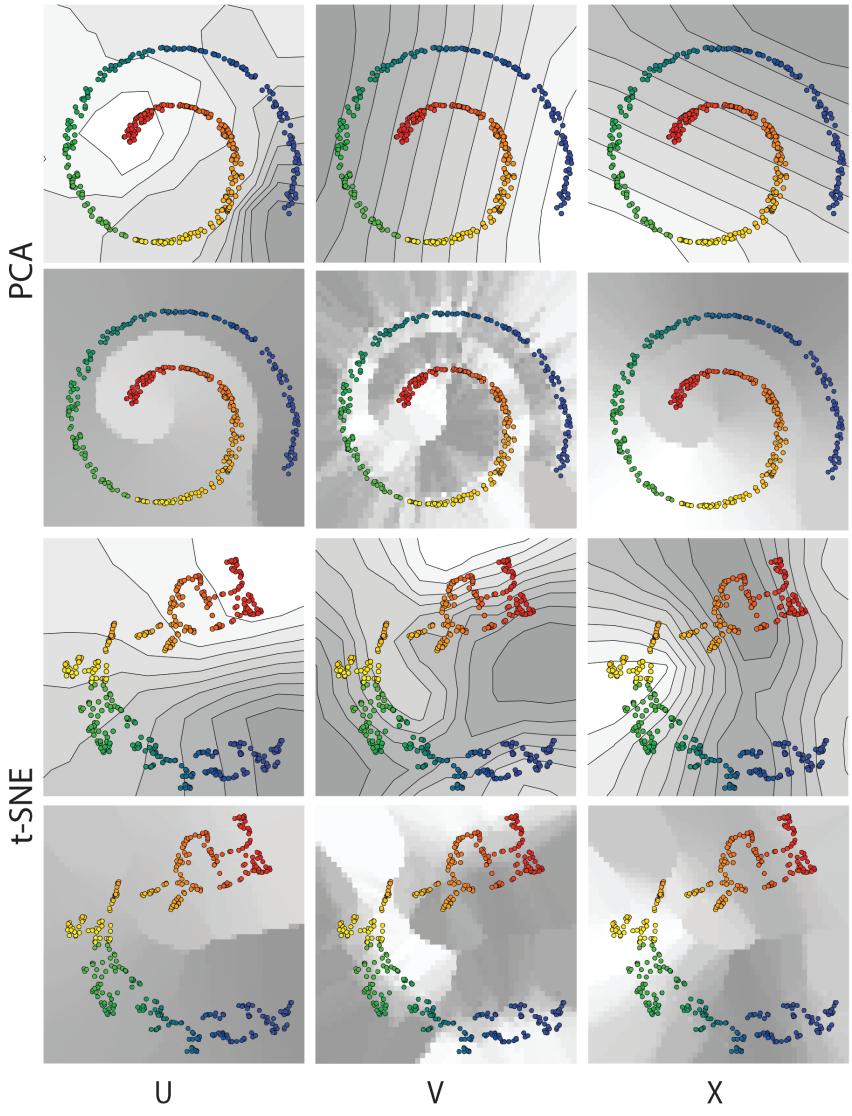

Figure 4.10: DimReader axes and value heatmaps for the u,v, and x dimensions of the swiss roll. A discussion of these plots is in Section 4.4.3.

4.4.2 Discovering Perturbations

The implementation of the equations in 4.3.4 for discovering perturbations is straightforward as long as the machine has sufficient memory to hold the expanded laplacian matrix, L_s. This matrix becomes very large for very high-dimensional data and thus requires significant memory. To solve this problem, we were again able to exploit the block structure of the tangent map as well as the structure of the Laplacian matrix: the diagonal values are $\sum_{j \neq i} S_{i,j}$ and the off diagonal values are $-S_{i,j}$. We implemented a version of power iteration that does not require access to the matrix $M^T M - \lambda L_s$ but instead requires a function that, when given a vector v, returns $(M^T M - \lambda L_s)v$. The multiplication function calculates elements of the output vector individually and thus does not require the entire L_s matrix.

Using this method, we uncovered perturbations for several datasets projected with t-SNE. In the following experiments, we use the method in Section 4.3.4 to find individual perturbations for each point.

Iris

We first look at the best perturbation for the Iris dataset. Figure 4.8 shows the DimReader plot for this perturbation as well as a plot for each dimension that shows how much that dimension was perturbed in each point through the color (the darker purple a point is, the more it was perturbed). The DimReader plot shows that the best perturbation only perturbs points in the Setosa (red) cluster. In the individual dimension plots, the Setosa cluster is perturbed primarily in the Sepal length and Sepal width dimensions which tells us that in this projection, points in the Setosa cluster are sensitive to changes in the Sepal dimensions. Comparing 4.8 to the t-SNE plots in Figure 4.7, the movement of the Setosa cluster with the discovered perturbation is similar to the movement when the sepal width or sepal length is perturbed.

MNIST Digits

Figure 4.9 (A) shows an sample of discovered perturbations in the projection. The perturbation images often resemble a variation of their corresponding digit or a nearby digit (due to the constraints defined in Section 4.3.4). These perturbations show us that t-SNE is capturing meaningful information about the dataset. In Figure 4.9 (A), the perturbation that moves the "seven" the most turns the "seven" into a "two" and moves it toward the cluster of "two"s. Thus, DimReader, is showing evidence that t-SNE is capturing information about what constitutes a "two" and is using that information to separate out the two's into their own (imperfect) cluster.

MNIST Fashion

The MNIST Fashion dataset is similar to the digits dataset in that each point represents a 28×28 pixel image and there are 10 different classes of images (articles of clothing) but is more complicated than the digits dataset. A t-SNE plot with selected perturbations found by our technique is shown in Figure 4.9 (B). Just as in the digits perturbations, there is structure in these perturbations. In the example in Figure 4.1 (C), our technique is finding perturbations that capture information about how t-SNE is projecting the data. DimReadertells us, that for the heel in the middle, the perturbation that moves this point the most, changes it from a heel into a flat shoe. This also shows us that t-SNE understand the difference between flat shoes and heels and is able to separate them.

4.4.3 Synthetic Examples

In this section we will look at the DimReader plot with two synthetic examples, the swiss roll and the interlocked rings, and compare them to the value heatmaps for each dimension from Stahnke et. al's Probing Projections.

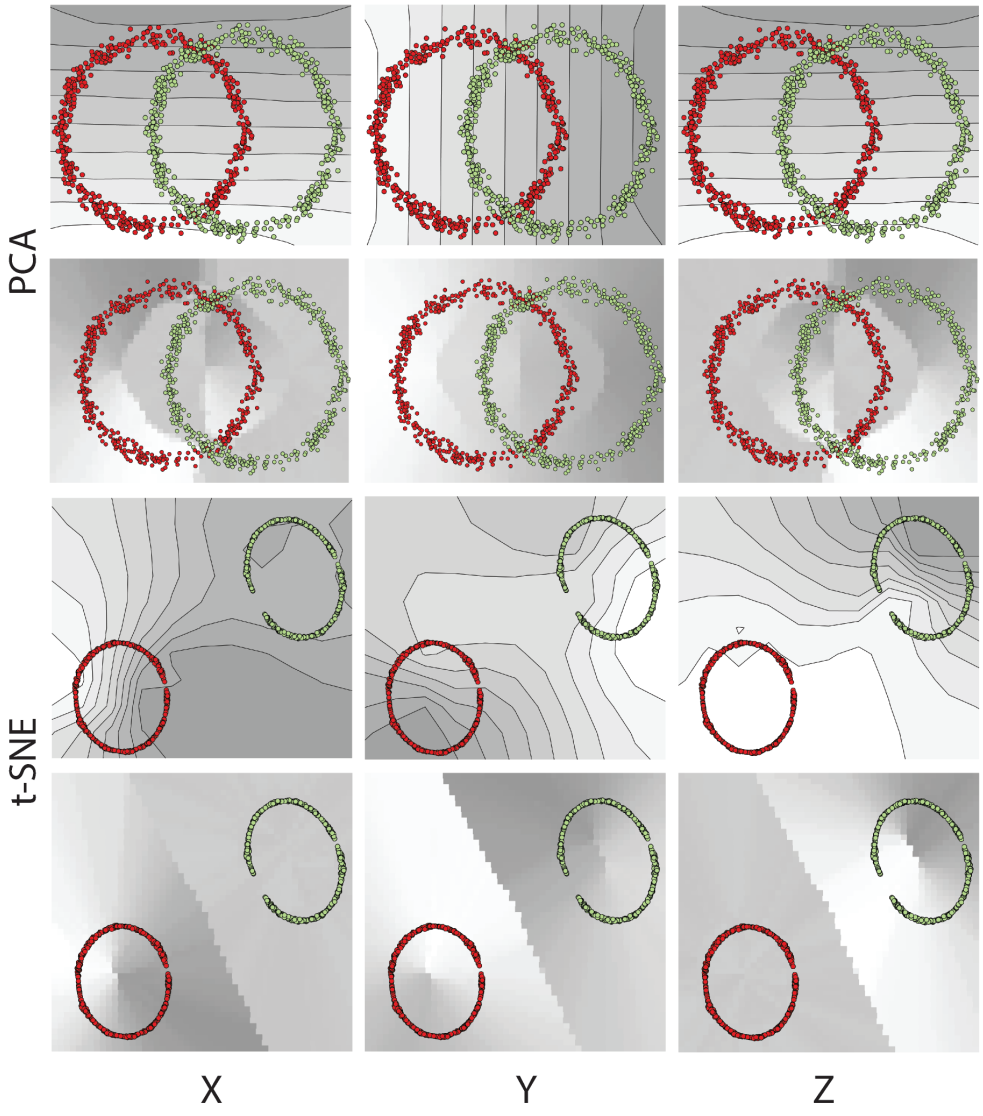

Figure 4.11: DimReader axes and value heatmaps for the x, y, and z dimension of the interlocked rings. A discussion of these plots is in Section 4.4.3.

Swiss Roll

The swiss roll dataset is calculated from the equations $x = u\cos(u)$, $y = u\sin(u)$, $z = v$ where $\frac{3\pi}{2} \leq u \leq \frac{9\pi}{2}$ and $0 \leq v \leq 15$. Figure 4.10 shows perturbations of the U,V, and X dimensions.

PCA In the v dimension, the DimReader plot shows that increasing the v in the original data moves the projected points to the left. In comparison, the value heatmap for the v dimension is difficult to read due to the high variance in v between neighboring points. This highlights a fundamental difference between DimReader and value heatmaps: DimReader is showing what the *projection* is doing while value heatmaps show the values of a dimension based on the placement of points. If we created a projection that simply mapped each point to its PCA coordinates through a table lookup, the value heatmaps would not change whereas the DimReader plots would show nothing because changing any of the dimensions would not change the projection.

The DimReader plot for the u dimension shows that changing the u dimension would move the point along the spiral, from red to blue. The isolines, however, are irregular from green to orange. These irregularities could be due to the resolution of the grid or the regularization. One direction for future work is to automatically determine the appropriate grid resolution and regularization for a projection.

t-SNE In the DimReader plot for t-SNE, the u dimension behaves exactly as we would expect, increasing as we move from red to blue. The spacing of the lines around the curve, specifically how the curves are wider on the outside than on the inside, indicates the bending does not reflect the underlying data but rather is caused by the projection.

In the DimReader plot for v, t-SNE has flipped the green and yellow segment (the points move to the bottom left rather than the upper right). This appears less clear from the value heatmap alone.

In the plots for x, the highest values in the heatmap do not match the area in the DimReader plot where the biggest change occurs.

Interlocked Rings

The DimReader plots and value heatmaps generated for the interlocked rings dataset are shown in Figure 4.11.

PCA The PCA plots from DimReader show that changing X or Z would move the points upward and changing the Y dimension move points to the right. Again, the plots here highlight the difference between our technique and the value heatmaps: the heatmaps of x and z tell us that the x and z values are only changing over one ring each whereas our technique is showing that PCA will move the rings vertically if the x or z dimension is changed. DimReader shows what the projection does, whereas heatmaps shows where the values go.

t-SNE T-SNE has separated the two rings well. In the DimReader plot of the X dimension, the red ring will move from left to right when the X dimension is changed. Comparing the X dimension to the Y dimension, for the red ring, the two axes are nearly perpendicular to one another. This suggests that t-SNE is primarily using these two dimensions for projecting the red ring. Furthermore, in the red ring the X dimension changes much quicker than the Y dimension (the lines are closer together) which indicates that t-SNE is distorting the shape of the red ring. Similar observations can be made about the green ring with the Y and Z dimensions. Again, the Y and Z are nearly perpendicular in the green ring. In the Y dimension, the axes change their behavior when they reach the gap in the green ring. Points in this region move more slowly when changed which in turn tells us that this is likely a tear in the ring caused by t-SNE that does not reflect the structure of the underlying data. It is not as clear from the value heatmaps that the gap is a tear in the data caused by the projection.

		MNIST Digits				S Curve			
		100	200	500	1000	100	200	500	1000
Regular	tSNE	3.3	6.9	21.2	88.4	2.8	6.0	21.9	90.3
DimReader (4 cores)		4.6	12.2	53.9	314.4	3.8	10.8	54.8	907.9
Regular	Isomap	0.5	2.1	14.9	79.6	0.1	0.6	6.4	46.5
DimReader (4 cores)		3.7	22.5	221.5	1845.1	2.5	17.1	210.4	1890.6
Regular	LLE	4.8	15.9	91.1	357.4	0.0	0.1	0.8	3.7
DimReader (4 cores)		6.5	20.4	103.7	391.4	0.3	0.6	2.3	8.3

Figure 4.12: Performance figures for the MNIST digits dataset and the S-Curve dataset, for progressively larger samples and three different NDR methods. All figures are reported in seconds.

4.4.4 Performance

Known Perturbations

In this section, we report performance figures for the prototype implementation of DimReader. Although we were reasonably careful with algorithmic and high-level design decisions that impact performance, we did not make a significant effort to make DimReader fast. We expect carefully-implemented versions of our proposal in high-performance languages such as C++ or Java to be significantly faster, possibly by an order of magnitude (typically the performance difference between Python and aggressively optimized, compiled languages).

A table showcasing typical results is included in Figure 4.12. The performance of DimReader for a given NDR method is dependent on two main factors: the number of input points and the overhead incurred by dual numbers. We need to execute a number of repeated runs proportional to the base-2 logarithm of the number of input points, and that is essentially unavoidable. We note that for the case of LLE and t-SNE, the optimizations we described in the previous section make the execution of the

dual-number version of the projection much faster than that of the regular numbers. As a result, DimReader can extract axes with a relatively small performance overhead.

For cases such as Isomap, on the other hand, where we performed no such optimizations, the performance of our method suffers a bit. We argue that this is an acceptable tradeoff: DimReader still works in an acceptable amount of time in the general case, but more careful implementations can be significantly more efficient.

Discovering Perturbations

The most expensive part of searching for a perturbation is calculating the tangent map. The tangent map is $n*d \times n*d$ and requires d executions of DimReader to build it. For datasets with a large number of data points and dimensions, this quickly becomes slow. Once we have the matrix, the performance for finding the best perturbation largely depends on whether or not we can the expanded Laplacian matrix, L_s, (described in Section 4.4.2) in memory. If we can't and have to use power iteration, the performance depends on the speed of our multiplication function as well as the amount of time it takes power iteration to converge. We did not make a significant effort to increase the performance for calculating the matrix or searching for perturbations; this remains for future work.

4.5 Discussion

Can we trust DimReader plots? While we have shown that DimReader can help determining how NDR plots can be trusted, a natural question to ask is: can the DimReader plots themselves be trusted? One natural scenario in which this comes up is when perturbation vectors of nearby projections disagree with one another. It's always possible to show the vectors themselves as a diagnostic of the quality of the reconstructed axis lines, but a proper, user-centric evaluation of the settings in which

DimReader's axes are more informative than naked NDR plots is clearly necessary, and will be the subject of future work.

Inverse readings DimReader enables interpretation of forward transformations: given a perturbation of an input and a visualization, DimReader provides an answer. But a different natural reading is the *inverse*: given a projected point and a direction of movement in the projection, what changes in the data could generate such movement? In principle, the derivative information obtained by autodiff also captures this inverse relationship [22], but the fact that we are dealing with *projections* makes the problem fundamentally harder. A full investigation is beyond the scope of this work.

More algorithms, better infrastructure While DimReader shows that it is possible to adapt a large number of existing NDR methods to run within an autodiff framework, one goal is to provide DimReader axes to as much existing visualization infrastructure as practically possible. In such scenarios, reducing the implementation effort even further would be desirable. The majority of our difficulties porting algorithms to automatic differentiation arose due to difficulties in evaluating derivatives of linear-algebraic concepts, such as solutions of a linear system and eigenvectors. Some of these have explicit formulas [84], but incorporating them in an autodiff system effectively and efficiently is a fundamental challenge beyond the scope of our work. We note, in addition, that our choice of automatic differentiation is not strictly necessary. Other methods exist to evaluate function derivatives, including manual derivation of the expressions. When using DimReader with a specific NDR method, these alternatives might be more attractive. This might be particularly true whenever approximations of the derivative can be computed more efficiently than autodiff.

4.6 Conclusion

In this chapter, we identified *infinitesimal perturbations* as a tool to enable interpretation of NDR plots, and presented DimReader, a technique that produces generalized axes for studying such perturbations. While much work remains to be done, DimReader strikes a favorable balance between generality and power, highlighting strengths and weaknesses of a variety of NDR methods, and providing a novel perspective into what NDR methods are actually visualizing.

Acknowledgements We acknowledge fruitful discussions of dimensionality reduction with Sean Stephens, Luiz Gustavo Nonato, and Joshua Levine on the energy formulation of t-SNE. This work has been partially supported by the NSF under the TRIPODS program, award number CCF-1740858, and IIS-1513651.

Chapter 5

Conclusion

In this dissertation, we reframed the problem of program debugging and understanding as a data analysis problem and explored how to employ visualization principles to create descriptive visualizations of program data. We explored two opposing applications of this perspective: (1) visualization of general programs and (2) visualization of a specific class of programs.

We presented Anteater which gathers data from general Python programs via tracing and applies visualization principles to present this data through interactive visualizations. Taking this data analysis perspective allows us to break away from traditional methods and facilitate exploratory debugging and understanding tasks. ProgDiff extends this work to facilitate comparison between consecutive executions of a program. It provides a variety of comparative visualizations to enable people to understand the effects of program changes. Anteater and ProgDiff make no assumptions about the content of the programs, only that they can be traced. As such, these methods broadly apply to understanding and debugging any program. On the other hand, because we make no assumptions about the programs, these methods only answer a limited scope of general questions for understanding programs. For example, they show us the behavior of this specific variable value, but cannot answer questions such as "how much does this variable value influence the value of this other value"?

In contrast, DimReader makes strict assumptions that enable more specific questions. DimReader assumes that a program takes in a datasest as input and performs differentiable computations to produce an output. These assumptions allow us to support a much more specific question of "if the input points changed in this specific way, how would the output change". DimReader only supports a very specific type of change, the infinitesimal perturbation to the input data points, on a specific type of program, dimensionality reductions. By constraining the programs in this way, DimReader enables

the design of richer visualizations for understanding the behavior of these projections that Anteater could never support in a general way. However, the strict constraints of DimReader means that other programs cannot employ DimReader and benefit from the rich, explanatory information it generates.

Each of these applications sacrifices either generality of applicable programs or specificity of supported tasks. These two present two extreme points in the range of possibilities in this design space. A such, the question remains, what intermediate design points exist that provide more generality than DimReader while enabling more specific tasks than Anteater? Deep learning systems provide a natural next target application. They encompass a much wider range of programs than DimReader, while readily facilitating more complex questions than Anteater. As a result, we can enable more specific questions about a broader class of programs.

In future work, we will explore how to employ the perturbation analysis from DimReader to create explanatory visualizations of interactions with deep learning systems. We will facilitate interaction inject human expertise into deep learning systems. Using automatic differentiation, we will create explanatory visualizations of the effects of these interactions on the system.

References

[1] Admin, *Intel trace analyzer and collector*, Oct. 2019. [Online]. Available: `https://software.intel.com/en-us/trace-analyzer`.

[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*, ser. SOFTVIS, Salt Lake City, Utah, USA: ACM, 2010, pp. 53–62, ISBN: 978-1-4503-0028-5. DOI: `10.1145/1879211.1879222`.

[3] S. Agarwal, K. Mierle, *et al.*, *Ceres solver*, `http://ceres-solver.org`.

[4] 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*, IEEE, 2012, pp. 545–548.

[5] E. Angel, *Interactive computer graphics: A top-down approach using OpenGL*, 2003.

[6] R. Arias-Hernandez, L. T. Kaastra, T. M. Green, and B. Fisher, "Pair analytics: Capturing reasoning processes in collaborative visual analytics," in *2011 44th Hawaii international conference on system sciences*, IEEE, 2011, pp. 1–10.

[7] M. Aupetit, "Visualizing distortions and recovering topology in continuous projection techniques," *Neurocomputing*, vol. 70, no. 7-9, pp. 1304–1330, 2007.

[8] 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)*, IEEE, 2013, pp. 1–4.

[9] S. Bergner, M. Sedlmair, T. Moller, S. N. Abdolyousefi, and A. Saad, "Paraglide: Interactive parameter space partitioning for computer simulations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 9, pp. 1499–1512, 2013.

[10] J. Bertin, *Semiology of Graphics: Diagrams, Networks, Maps*. UMI Research Press, 1983, ISBN: 9780835735322. [Online]. Available: https://books.google.com/books?id=0Ju8bwAACAAJ.

[11] 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*, ser. SANER, Mar. 2015, pp. 535–539. DOI: 10.1109/SANER.2015.7081872.

[12] 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*, ser. SC '16, Salt Lake City, Utah: IEEE Press, 2016, 47:1–47:11, ISBN: 978-1-4673-8815-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014967.

[13] I. Borg and P. J. Groenen, *Modern multidimensional scaling: Theory and applications*. Springer Science & Business Media, 2005.

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

[15] J. Boy, R. A. Rensink, E. Bertini, and J.-D. Fekete, "A principled way of assessing visualization literacy," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 1963–1972, 2014.

[16] M. Brehmer, S. Ingram, J. Stray, and T. Munzner, "Overview: The design, adoption, and analysis of a visual document mining tool for investigative journalists," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2271–2280, 2014.

[17] M. Brehmer and T. Munzner, "A multi-level typology of abstract visualization tasks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2376–2385, 2013.

[18] A. Buja and D. F. Swayne, "Visualization methodology for multidimensional scaling," *Journal of Classification*, vol. 19, no. 1, pp. 7–43, 2002.

[19] 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*, ser. UIST '13, St. Andrews, Scotland, United Kingdom: ACM, 2013, pp. 473–484, ISBN: 978-1-4503-2268-3. DOI: 10.1145/2501988.2502050. [Online]. Available: http://doi.acm.org/10.1145/2501988.2502050.

[20] L. Burgess-Yeo, *F# tree diff algorithm*, Jan. 2020. [Online]. Available: https://medium.com/f-ing-about/f-tree-diff-algorithm-5b7f9c85beac.

[21] G. Canfora, L. Cerulo, and M. Di Penta, "Tracking your changes: A language-independent approach," *IEEE software*, vol. 26, no. 1, pp. 50–57, 2008.

[22] M. P. Carmo, "Differential geometry of surfaces," *Differential Forms and Applications*, pp. 77–98, 1994.

[23] M. S. T. Carpendale, *Considering visual variables as a basis for information visualisation*, 2003. DOI: 10.11575/PRISM/10182. [Online]. Available: https://prism.ucalgary.ca/handle/1880/45758.

[24] M. Cavallo and Ç. Demiralp, "A visual interaction framework for dimensionality reduction based data exploration," *ACM Human Factors in Computing Systems (CHI)*, 2018.

[25] M. Chalmers, "A linear iteration time layout algorithm for visualising high-dimensional data," in *Proceedings of the 7th conference on Visualization'96*, IEEE Computer Society Press, 1996, 127–ff.

[26] Y.-H. Chan, C. D. Correa, and K.-L. Ma, "Flow-based scatterplots for sensitivity analysis," in *Visual Analytics Science and Technology (VAST), 2010 IEEE Symposium on*, IEEE, 2010, pp. 43–50.

[27] ——, "The generalized sensitivity scatterplot," *IEEE transactions on visualization and computer graphics*, vol. 19, no. 10, pp. 1768–1781, 2013.

[28] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *Acm Sigmod Record*, vol. 25, no. 2, pp. 493–504, 1996.

[29] S. Cheng and K. Mueller, "The data context map: Fusing data and attributes into a unified display," *IEEE transactions on visualization and computer graphics*, vol. 22, no. 1, pp. 121–130, 2016.

[30] 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*, ser. ICSE '16, Austin, Texas: ACM, 2016, pp. 605–608, ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889167. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889167.

[31] D. B. Coimbra, R. M. Martins, T. T. Neves, A. C. Telea, and F. V. Paulovich, "Explaining three-dimensional dimensionality reduction plots," *Information Visualization*, vol. 15, no. 2, pp. 154–172, 2016.

[32] R. D. Cook, "Detection of influential observation in linear regression," *Technometrics*, vol. 19, no. 1, pp. 15–18, 1977.

[33] T. H. Cormen, *Introduction to algorithms*. MIT press, 2009.

[34] B. Cornelissen and L. Moonen, "Visualizing similarities in execution traces," in *Proceedings of the 3rd Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, 2007, pp. 6–10.

[35] C. Curtsinger and E. D. Berger, "Coz: Finding code that counts with causal profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 184–197.

[36] R. Cutura, S. Holzer, M. Aupetit, and M. Sedlmair, "Viscoder: A tool for visually comparing dimensionality reduction algorithms,"

[37] S. Diehl, *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.

[38] L. N. Q. Do, S. Krüger, P. Hill, K. Ali, and E. Bodden, "Visuflow: A debugging environment for static analyses," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18, Gothenburg, Sweden: ACM, 2018, pp. 89–92, ISBN: 978-1-4503-5663-3. DOI: 10.1145/3183440.3183470. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183470.

[39] D. L. Donoho and C. Grimes, "Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data," *Proceedings of the National Academy of Sciences*, vol. 100, no. 10, pp. 5591–5596, 2003.

[40] N. Elmqvist and J. S. Yi, "Patterns for visualization evaluation," *Information Visualization*, vol. 14, no. 3, pp. 250–269, 2015.

[41] A. Endert, C. Han, D. Maiti, L. House, and C. North, "Observation-level interaction with statistical models for visual analytics," in *Visual Analytics Science and Technology (VAST), 2011 IEEE Conference on*, IEEE, 2011, pp. 121–130.

[42] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[43] N. Ferreira, J. T. Klosowski, C. E. Scheidegger, and C. T. Silva, "Vector field k-means: Clustering trajectories by fitting multiple vector fields," *Computer Graphics Forum*, vol. 32, no. 3pt2, pp. 201–210, 2013.

[44] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander, "Debugging from the student perspective," *IEEE Transactions on Education*, vol. 53, no. 3, pp. 390–396, 2009.

[45] B. Fluri and H. C. Gall, "Classifying change types for qualifying change couplings," in *14th IEEE International Conference on Program Comprehension (ICPC'06)*, IEEE, 2006, pp. 35–45.

[46] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, vol. 33, no. 11, pp. 725–743, 2007.

[47] I. K. Fodor, "A survey of dimension reduction techniques," *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory*, vol. 9, pp. 1–18, 2002.

[48] P. Gestwicki and B. Jayaraman, "Methodology and architecture of jive," in *Proceedings of the 2005 ACM Symposium on Software Visualization*, ser. SoftVis '05, St. Louis, Missouri: ACM, 2005, pp. 95–104, ISBN: 1-59593-073-6. DOI: 10.1145/1056018.1056032. [Online]. Available: http://doi.acm.org/10.1145/1056018.1056032.

[49] M. Gleicher, "Explainers: Expert explorations with crafted projections," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2042–2051, 2013.

[50] P. Godfrey, J. Gryz, and P. Lasek, "Interactive visualization of large data sets," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 8, pp. 2142–2157, 2016.

[51] G. H. Golub and C. F. Van Loan, *Matrix computations*. JHU Press, 2012, vol. 3.

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

[53] 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)*, IEEE, 2017, pp. 54–63.

[54] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[55] M. Hashimoto and A. Mori, "Diff/ts: A tool for fine-grained structural change analysis," in *2008 15th working conference on reverse engineering*, IEEE, 2008, pp. 279–288.

[56] 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*, ACM, 2018, p. 532.

[57] ——, "Visual debugging techniques for reactive data visualization," *Comput. Graph. Forum*, vol. 35, no. 3, 271â280, Jun. 2016, ISSN: 0167-7055.

[58] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, 1990, pp. 234–245.

[59] S. Ingram, *Personal communication*, 2017.

[60] S. Ingram, T. Munzner, and M. Olano, "Glimmer: Multilevel mds on the gpu," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 2, pp. 249–261, 2009.

[61] D. Jackson, D. A. Ladd, *et al.*, "Semantic diff: A tool for summarizing the effects of modifications.," in *ICSM*, vol. 94, 1994, pp. 243–252.

[62] D. H. Jeong, C. Ziemkiewicz, B. Fisher, W. Ribarsky, and R. Chang, "Ipca: An interactive system for pca-based visual analytics," *Computer Graphics Forum*, vol. 28, no. 3, pp. 767–774, 2009.

[63] P. Joia, D. Coimbra, J. A. Cuminato, F. V. Paulovich, and L. G. Nonato, "Local affine multidimensional projection," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2563–2571, 2011.

[64] 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*, ACM, 2017, pp. 737–745.

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

[66] G. Kindlmann and C. Scheidegger, "An algebraic process for visualization design," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2181–2190, 2014.

[67] R. M. Kirby and C. T. Silva, "The need for verifiable visualization," *IEEE Computer Graphics and Applications*, vol. 28, no. 5, 2008.

[68] H. Lam, M. Tory, and T. Munzner, "Bridging from goals to tasks with design study analysis reports," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 435–445, 2018.

[69] T. D. LaToza and B. A. Myers, "Developers ask reachability questions," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, ACM, 2010, pp. 185–194.

[70] J. Lee and M. Verleysen, "Quality assessment of nonlinear dimensionality reduction based on k-ary neighborhoods," in *New Challenges for Feature Selection in Data Mining and Knowledge Discovery*, 2008, pp. 21–35.

[71] S. Lehnert, M. Riebisch, *et al.*, "A taxonomy of change types and its application in software evolution," in *2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems*, IEEE, 2012, pp. 98–107.

[72] S. Lespinats and M. Aupetit, "Checkviz: Sanity check and topological clues for linear and non-linear mappings," in *Computer Graphics Forum*, Wiley Online Library, vol. 30, 2011, pp. 113–125.

[73] 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*, ser. CHI '95, Denver, Colorado, USA: ACM Press/Addison-Wesley Publishing Co., 1995, pp. 480–486, ISBN: 0-201-84705-1. DOI: 10.1145/223904.223969. [Online]. Available: http://dx.doi.org/10.1145/223904.223969.

[74] S. Litvinov, M. Mingazov, V. Myachikov, V. Ivanov, Y. Palamarchuk, 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.

[75] L. van der Maaten, *A Python implementation of t-SNE*, https://lvdmaaten.github.io/tsne/.

[76] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. Nov, pp. 2579–2605, 2008.

[77] J. I. Maletic, A. Marcus, and M. L. Collard, "A task oriented view of software visualization," in *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*, IEEE, 2002, pp. 32–40.

[78] A. V. Miranskyy, M. Davison, R. M. Reesor, and S. S. Murtaza, "Using entropy measures for comparison of software traces," *Information Sciences*, vol. 203, pp. 59–72, 2012.

[79] D. Moritz, D. Fisher, B. Ding, and C. Wang, "Trust, but verify: Optimistic visualizations of approximate queries for exploring big data," in *Proceedings of the 2017 CHI conference on human factors in computing systems*, 2017, pp. 2904–2915.

[80] A. Morrison, G. Ross, and M. Chalmers, "A hybrid layout algorithm for sub-quadratic multidimensional scaling," in *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium on*, IEEE, 2002, pp. 152–158.

[81] T. Munzner, *Visualization analysis and design*. AK Peters/CRC Press, 2014.

[82] 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*, ser. ICSE '16, Austin, Texas: ACM, 2016, pp. 601–604, ISBN: 978-1-4503-4205-6. DOI: 10.1145/2889160.2889166. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889166.

[83] F. V. Paulovich, L. G. Nonato, R. Minghim, and H. Levkowitz, "Least square projection: A fast high-precision multidimensional projection technique and its application to document mapping," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 3, pp. 564–575, 2008.

[84] K. B. Petersen, M. S. Pedersen, *et al.*, "The matrix cookbook," *Technical University of Denmark*, vol. 7, p. 15, 2008.

[85] G. Pothier, É. Tanter, and J. Piquer, "Scalable omniscient debugging," in *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, ser. OOPSLA '07, Montreal, Quebec, Canada: ACM, 2007, pp. 535–552, ISBN: 978-1-59593-786-5. DOI: 10.1145/

1297027.1297067. [Online]. Available: http://doi.acm.org/10.1145/1297027.1297067.

[86] B. A. Price, R. M. Baecker, and I. S. Small, "A principled taxonomy of software visualization," *Journal of Visual Languages & Computing*, vol. 4, no. 3, pp. 211–266, 1993.

[87] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 511–526, 2005.

[88] *Pyodide*. [Online]. Available: https://pyodide.org/.

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

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

[91] G.-C. Roman and K. C. Cox, "Program visualization: The art of mapping programs to pictures," in *Proceedings of the 14th International Conference on Software Engineering*, 1992, pp. 412–420.

[92] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

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

[94] D. Sacha, L. Zhang, M. Sedlmair, J. A. Lee, J. Peltonen, D. Weiskopf, S. C. North, and D. A. Keim, "Visual interaction with dimensionality reduction: A

structured literature analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 241–250, 2017.

[95] A. Satyanarayan, D. Moritz, K. Wongsuphasawat, and J. Heer, "Vega-lite: A grammar of interactive graphics," *IEEE transactions on visualization and computer graphics*, vol. 23, no. 1, pp. 341–350, 2016.

[96] H. M. Schey, *Div, grad, curl, and all that*. Norton, 1973.

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

[98] M. Sedlmair, M. Brehmer, S. Ingram, and T. Munzner, "Dimensionality reduction in the wild: Gaps and guidance," *Dept. Comput. Sci., Univ. British Columbia, Vancouver, BC, Canada, Tech. Rep. TR-2012-03*, 2012.

[99] M. Sedlmair, T. Munzner, and M. Tory, "Empirical guidance on scatterplot and dimension reduction technique choices," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2634–2643, 2013.

[100] C. Seifert, V. Sabol, and W. Kienreich, "Stress maps: Analysing local phenomena in dimensionality reduction based visualisations," in *IEEE Int. symposium on Visual Analytics Science and Technology.*, 2010.

[101] M. Sensalire, P. Ogao, and A. Telea, "Classifying desirable features of software visualization tools for corrective maintenance," in *Proceedings of the 4th ACM symposium on Software visualization*, 2008, pp. 87–90.

[102] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," Carnegie-Mellon University. Department of Computer Science, Tech. Rep., 1994.

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

[104] 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*, ser. PADD, Madison, Wisconsin, USA: ACM, 1988, pp. 206–215, ISBN: 0-89791-296-9. DOI: 10.1145/68210.69235.

[105] J. Stahnke, M. Dörk, B. Müller, and A. Thom, "Probing projections: Interaction techniques for interpreting arrangements and errors of dimensionality reductions," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 1, pp. 629–638, 2016.

[106] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, *Software Visualization*. MIT Press, 1998.

[107] J. T. Stasko and C. Patterson, "Understanding and characterizing software visualization systems," in *Proceedings IEEE Workshop on Visual Languages*, IEEE, 1992, pp. 3–10.

[108] 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*, ser. SoftVis '08, Ammersee, Germany: ACM, 2008, pp. 47–56, ISBN: 978-1-60558-112-5. DOI: 10.1145/1409720.1409729. [Online]. Available: http://doi.acm.org/10.1145/1409720.1409729.

[109] R. Suzuki, G. Soares, A. Head, E. Glassman, R. Reis, M. Mongiovi, L. D'Antoni, and B. Hartmann, "Tracediff: Debugging unexpected code behavior using trace divergences," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2017, pp. 107–115.

[110] S. Taheri, I. Briggs, M. Burtscher, and G. Gopalakrishnan, "Difftrace: Efficient whole-program trace analysis and diffing for debugging," in *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, IEEE, 2019, pp. 1–12.

[111] J. B. Tenenbaum, V. De Silva, and J. C. Langford, "A global geometric framework for nonlinear dimensionality reduction," *science*, vol. 290, no. 5500, pp. 2319–2323, 2000.

[112] R. Tiarks and T. Roehm, "Challenges in program comprehension," *Softwaretechnik-Trends*, vol. 32, no. 2, pp. 19–20, 2012.

[113] W. S. Torgerson, "Multidimensional scaling of similarity," *Psychometrika*, vol. 30, no. 4, pp. 379–393, 1965.

[114] J. Trümper, J. Bohnet, and J. Döllner, "Understanding complex multithreaded software systems by using trace visualization," in *Proceedings of the 5th International Symposium on Software Visualization*, ser. SOFTVIS, Salt Lake City, Utah, USA: ACM, 2010, pp. 133–142, ISBN: 978-1-4503-0028-5. DOI: 10.1145/1879211.1879232.

[115] J. Trümper, J. Döllner, and A. Telea, "Multiscale visual comparison of execution traces," in *2013 21st International Conference on Program Comprehension (ICPC)*, IEEE, 2013, pp. 53–62.

[116] A. Vellido, J. D. Martín-Guerrero, and P. J. Lisboa, "Making machine learning models interpretable.," in *ESANN*, Citeseer, vol. 12, 2012, pp. 163–172.

[117] I. Vessey, "Expertise in debugging computer programs: A process analysis," *International Journal of Man-Machine Studies*, vol. 23, no. 5, pp. 459–494, 1985.

[118] S. Voigt, J. Bohnet, and J. Dollner, "Object aware execution trace exploration," in *2009 IEEE International Conference on Software Maintenance*, IEEE, 2009, pp. 201–210.

[119] S. v. d. Walt, S. C. Colbert, and G. Varoquaux, "The numpy array: A structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, 2011.

[120] M. Wattenberg, F. ViÃ©gas, and I. Johnson, "How to use t-sne effectively," *Distill*, 2016, http://distill.pub/2016/misread-tsne.

[121] 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.*, vol. 13, no. 6, pp. 1224–1231, 2007.